Deriving Production Rules for Constraint Maintenance

Stefano Ceri*
Jennifer Widom

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
ceri@cs.stanford.edu, widom@ibm.com

Abstract. Traditionally, integrity constraints in database systems are maintained either by rolling back any transaction that produces an inconsistent state or by disallowing or modifying operations that may produce an inconsistent state. An alternative approach is to provide automatic "repair" of inconsistent states using production rules. For each constraint, a production rule is used to detect constraint violation and to initiate database operations that restore consistency.

We describe an SQL-based language for defining integrity constraints and a framework for translating these constraints into constraint-maintaining production rules. Some parts of the translation are automatic while other parts require user intervention. Based on the semantics of our set-oriented production rules language and under certain assumptions, we prove that at the end of each transaction the rules are guaranteed to produce a state satisfying all defined constraints. We apply our approach to a good-sized example.

1 Introduction

In database systems, an *integrity constraints* facility permits logical specification of those database states that are considered acceptable, or *consistent.* In current systems, mechanisms for defining and enforcing integrity constraints are limited. Most relational database systems support only specific types of constraints, such as uniqueness of keys and referential integrity, rather than supporting arbitrary predicates. Furthermore, when constraints are violated, "repair" of the database state usually is limited to fixed reversal actions, such as rolling back the current operation or the entire transaction. (Consequently, for increased flexibility, integrity constraints often are encoded within applications, usually in an ad-hoc manner.) An improved approach to constraint enforcement allows definition of compensating actions that correct violation of each constraint according to a well-understood, application-dependent semantics.

Most research in the area of integrity constraints has focused on efficiently determining actual or potential constraint violation, sometimes considering quite general constraints (such as arbitrary predicates). In this paper, we also consider general constraints, but we focus on the issue of constraint enforcement. A language is proposed for specifying constraints on relational databases. We then provide a framework for translating constraint specifications into *production rules* that maintain the constraints. Production rules in database systems allow specification of data manipulation operations that are automatically executed whenever certain events occur and/or certain conditions are met [DE89, Han89, KMS90, SJ*90, WF90]. The usefulness of incorporating production rules into database systems is well accepted [EC75, MD89, Mor83], particularly in the context of constraint enforcement. However, we know of no automatic (or semi-automatic) method for specifying general constraints in a high-level, non-procedural language, then deriving lower-level production rules that maintain the constraints. We describe such a method.

The constraint and rule languages we use are based on an extended version of SQL [HF*89, IBM88], although our work could easily be adapted for alternate languages. Constraints are expressed as predicates over the database state: if the predicate is true in a particular state, then the constraint is *violated* and the state is *inconsistent.*[1] Constraints may be ordered, specifying that certain constraints will be enforced earlier than (and therefore may be assumed valid by) other constraints. The production rule language is described in [WF90]. Prior familiarity with this rule language is not necessary; an overview is provided.

Production rules enforce constraints by issuing actions to correct violation. In many cases, several possible actions may correct a given constraint violation, and which action is most appropriate may depend on the application. Thus, for each constraint, the compensating actions are specified by the application designer. However, several other necessary components of the derivation can be performed automatically. We envision an interactive system for deriving rules from constraints with a structure as illustrated in Fig. 1. The automatic portions of the derivation include:

- Producing *rule templates* from constraints: Rule templates enumerate all operations that may cause constraint violation (these form the triggering com-

*On leave from University of Modena

[1] For our framework, it is more convenient for constraints to specify the inconsistent rather than the consistent states. This choice does not affect expressiveness, since the alternative semantics can be achieved simply by negating each constraint.

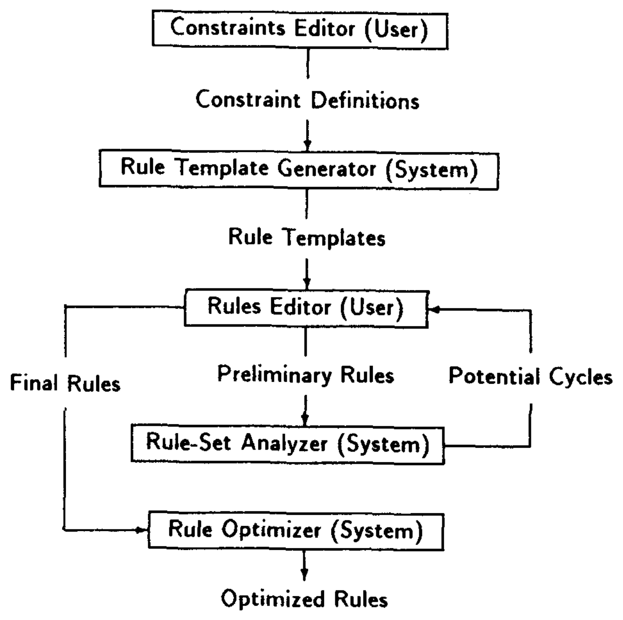

Figure 1: *Interactive System for Rule Derivation*

ponents of the constraint-enforcing rules) and include rule conditions. Rule actions are provided by the user.

- Detecting potential cycles in rule activation: As the number of rules increases and rules become more complex, there is increasing possibility of infinite triggering behavior. This component detects the potential for such behavior and provides warnings to the user.
- Rule optimization: The system automatically optimizes rules derived from constraints, preserving their constraint-maintaining semantics.

Each automatic component is described in some detail. We also describe those tasks that must be performed by the users of the system. Finally, a theorem is proven stating that under certain assumptions regarding the users' obligations (such as correctness of compensating actions and finite triggering behavior), the final set of production rules is guaranteed to maintain all defined constraints. That is, at the end of every transaction, rule execution terminates in a consistent state.

1.1 Related Work

As mentioned above, most work involving integrity constraints in database systems has addressed—in a variety of settings—the problem of efficiently detecting constraint violation [BBC80,HMN84,HI85,KP81,Nic82, QS87,Sto75]. Some of this work describes algorithms for detecting in advance that an operation may cause constraint violation; such operations are not permitted to proceed. In other work, inconsistent states are detected as they occur; consistency is restored by performing undo or rollback operations. In our approach, inconsistent states may occur and are detected, but consistency is restored by issuing corrective actions that depend on the particular constraint violation.

Approaches similar to ours are taken in [CTF88, Mor84, UD90, UD91], but in restricted settings. In [CTF88], only referential integrity and inclusion dependency constraints are considered. The user may define compensating actions (drawn from a restricted set) to be executed when constraints are violated. In [Mor84], the focus is on a very high-level language for expressing inter-relational constraints. The set of expressible constraints is a subset of those expressible using arbitrary predicates. In many cases, specific compensating actions may be derived automatically from constraints, subject to certain "hints" provided by the constraint definer. In [UD90,UD91], analysis of constraints is considered in an object-oriented environment. Constraints are represented using Horn logic (again permitting only a subset of arbitrary predicates). Constraint analysis reveals the effects of constraints on object manipulation, determines possible constraint violations, and suggests propagation actions for correcting violations.

Several other papers also extend the standard approaches to constraint definition and enforcement. In [CG88], logic programming is used to express and evaluate constraints. At run-time, a given transaction can be checked to verify that it will maintain consistency with respect to a set of constraints. If consistency is not guaranteed, the system can explain which constraints are violated and can suggest compensating actions. A similar approach is described in [SMS87], however this work considers a compile-time rather than run-time environment. When transactions are determined to have potential for constraint violation, feedback is provided to the user in the form of suggested tests and updates to be added to the transaction. In [Wal89], constraint definitions include both conditions on the state (which are checked) and additional actions that are automatically executed after certain operations to help maintain consistency. This is similar to defining constraints directly as the rules that enforce them, as in [SK84]. In our approach, constraints are defined at a higher, non-procedural level, from which constraint-maintaining rules are derived.

1.2 Outline of Paper

Preliminary material is presented in Section 2: a case study is introduced, serving as a source of examples throughout the paper, and an overview of the rule language is given. Section 3 presents the syntax and semantics of the constraint language. Section 4 describes the derivation of a rule for enforcing a single constraint, including automatic generation of those operations that may cause constraint violation. Section 5 then considers the set of rules for maintaining multiple constraints; in particular, it shows how potential cyclic behavior in such rule sets can be detected. Rule optimization is covered in Section 6. Section 7 considers system execution, showing that termination in a consistent state is guaranteed under certain assumptions. Finally, in Section 8, we conclude, discuss general use of the facility, and describe future work.

2 Preliminaries

2.1 Case Study

Examples throughout the paper are drawn from a case study concerning a *Power Distribution Design System*, a database application supporting the design and maintenance of electricity networks.[2] Due to space limitations, only portions of the study are included in this paper; all details appear in the technical report [CW90]. Note that many of the constraints considered in the study cannot be supported in conventional database systems.

Briefly, a power network connects a collection of *plants* to a collection of *users*, possibly through intermediate *nodes*. The network designer determines the location of plants, nodes, and users. Also specified by the designer is the power produced by each plant, the power required by each user, and the power loss incurred at each intermediate node. The designer places directed *wires* between plants, users, and nodes, specifying for each the *wire type* along with the *voltage* and *power* to be carried by that wire.

Multiple wires may be placed between any two points, and each set of wires is enclosed within a *tube*. Placement of tubes is not difficult: there is at most one tube between any two points, a tube should be *protected* if it contains high voltage wires, and tubes should have large enough *cross sections* to enclose their wires. Tube placement can, in fact, be specified solely in terms of constraints—tubes are then automatically inserted and deleted by the rules that maintain the constraints. Thus, once appropriate constraints are defined, the designer need not consider tubes at all.

Further constraints specify that the power output required for each plant should not exceed the produced power, the power output required for each node should not exceed the deliverable power (based on input), and each user should receive at least its required power. For reliability, each user should be connected to at least two plants. Using capitalization to denote primary keys, the relational schema for the case study is:

```
plant(PLANT-ID, location, power)
user(USER-ID, location, power)
node(NODE-ID, location, loss)
wire(WIRE-ID, fr, to, type, voltage, power)
tube(TUBE-ID, fr, to, type)
wire-type(TYPE, max-voltage, max-power
        cross-section)
tube-type(TYPE, protected, cross-section)
```

ID's for plants, users, and nodes all are drawn from the same domain—a constraint will specify that these are not duplicated. Attributes `fr` and `to` of tables `wire` and `tube` take their values from this domain.

2.2 Production Rules

We provide a brief overview of the set-oriented, SQL-based production rules language used in the remainder of the paper. Further details and numerous examples appear in [WF90]. We consider a relational database system with an integrated production rules facility. The system has all the usual database functionality; in addition, a set of production rules may be defined. In general, production rules specify actions to be performed when certain events occur or conditions are met. In our language, each rule contains three components: a *transition predicate*, which controls rule triggering, a *condition*, which is checked before the rule may execute its *action*, which is a sequence of data manipulation operations or a rollback request. Production rules are not activated until the commit point of each transaction, at which time all triggered rules are considered. The data manipulation operations executed as part of a rule's action may trigger additional rules. Once there are no triggered rules left to consider, the transaction is committed. Further details follow.

Rules are based on the notion of *transitions*. A transition is a database state change resulting from execution of a sequence of data manipulation operations. We consider the *net effect* of transitions, meaning that: (1) if a tuple is updated several times, we consider only the composite update; (2) if a tuple is updated then deleted, we consider only the deletion; (3) if a tuple is inserted then updated, we consider this as inserting the updated tuple; (4) if a tuple is inserted then deleted, it is not considered at all.

The syntax for defining production rules is:

> **create rule** *name*
> **when** *transition predicate*
> [**if** *condition*]
> **then** *action*

A rule is *triggered* by a given transition when its *transition predicate* holds with respect to that transition. Transition predicates specify operations on particular tables and columns. The permitted forms are: `inserted into t`, `deleted from t`, and `updated t.c`, where `t` is a table name and `c` is a column of table `t`. A transition predicate lists one or more such specifications, and the predicate holds with respect to a transition if at least one of the specified operations occurred in the net effect of the transition. Once a rule is triggered, it may be chosen for evaluation (as described below). At this point, the rule's *condition* is checked—rule conditions are arbitrary predicates on the database state. If the condition is true, the *action* is executed. An action may specify a list of SQL data manipulation operations to be executed or it may request a rollback of the current transaction.

The condition and action parts of a rule may refer to the current state of the database through embedded SQL `select` operations. In addition, these components may refer to *transition tables*. A transition table is a logical table reflecting changes that have occurred during a transition. At the end of a given transition, transition table `inserted t` refers to those tuples of table `t` in the current state that were inserted by the transition, transition table `deleted t` refers to those tuples of table `t` in the pre-transition state that were deleted by the transition, transition table `old updated t` refers to those

[2] This problem was presented to author Ceri by L. Dalle in the context of a joint study between *ENEL*—the Italian Electric Energy Agency—and the Politecnico di Milano.

tuples of table `t` in the pre-transition state for which column `c` was updated by the transition, and transition table `new updated t.c` refers to the current values of the same tuples. Transition tables may be referenced in the `from` clauses of `select` operations in the usual way. As a restriction, a rule may only refer to those transition tables corresponding to its transition predicate. For example, a rule may refer to `old updated t.c` and `new updated t.c` only if `updated t.c` is included in its transition predicate; similarly for `inserted` and `deleted`.

Finally, we describe rule execution. First, a user or application executes a transaction—a sequence of SQL operations; rules are considered at the commit point of the transaction. The state change resulting from this initial transaction creates the first relevant transition, and some set of rules are triggered by this transition. One rule is chosen from this set for evaluation. To influence rule selection, rules may be partially ordered, whereby a rule will be chosen such that no other triggered rule is higher in the ordering. The condition part of the selected rule is checked; if it does not hold, a new triggered rule is selected for evaluation. If the rule's condition does hold, its action is executed. (If no triggered rule with a true condition is found, rule execution terminates.) Let R (say) be the rule whose action is executed, and assume its action is not `rollback`. After execution of R's action, all rules not previously evaluated are now triggered only if their transition predicate holds with respect to the composite transition created by the initial transaction and subsequent execution of R's action. That is, these rules consider R's action as if it were executed as part of the user-generated transaction. Rules already evaluated have already "processed" the initial transaction; thus, they are triggered again if their transition predicate holds with respect to the transition created by R's action. Rule R will be triggered a second time only if future rule execution produces a new net effect satisfying its transition predicate.

Consider now an arbitrary point in rule processing. A given rule is triggered if its transition predicate holds with respect to the (composite) transition since the point following its most recent execution. If, at last evaluation, the rule's condition was found to be false, then the rule is considered with respect to the transition since that point of evaluation.[3] If a rule has not yet been evaluated, it is considered with respect to the transition since the start of the initial transaction. One rule is chosen from the set of triggered rules; if its condition holds, its action is executed, otherwise another triggered rule is selected. If a `rollback` action is encountered, the system rolls back to the start of the initial user-generated transaction and no rules are triggered. Otherwise, rule processing terminates when the set of triggered rules is empty or when no triggered rule has a true condition; the entire transaction is then committed.

[3]Here, we deviate somewhat from the semantics as presented in [WF90], where a rule is considered with respect to the transition since before its last execution. It should be possible in the rule system to permit both interpretations.

3 Constraint Language

We define a general language for expressing integrity constraints. A constraint has two parts: a *table list*, specifying tables relevant to the constraint, and an unrestricted *SQL predicate*, which must hold in exactly those states violating the constraint. As an introductory example, consider the constraint "each wire's voltage does not exceed the maximum for its wire type". In the proposed language, this is expressed as:

```
wire: voltage > any (select max-voltage
                     from wire-type
                     where type = wire.type)
```

(We must use "> any" here rather than the more natural ">" since the right side of the comparison is a singleton set, not a single value.) Remember that a constraint specifies the inconsistent states. Thus, a state violates this constraint if any tuple in table `wire` has a `voltage` exceeding the `max-voltage` for its wire `type`. A complete syntax and semantics follows.

3.1 Syntax

A grammar for the constraint language is given in Fig. 2; many examples are provided below. The core of the language is a variation on the usual SQL syntax for predicates, similar to that used in the `if` clause of our production rules [WF90]. Several advanced features are included, such as boolean types, tuple constructors, and user-defined functions. In [CW90], we also include *table expressions* (as described with respect to the *Starburst* database system in [HF*89]); space considerations have forced their omission here. In the grammar's productions, we use T_i to denote table names, V_i to denote table variable names, and C_i to denote column names. Repetition is represented explicitly by enumerating n terms ($n > 0$). Optional terms are enclosed in square brackets. (In our examples, we permit several straightforward abbreviations to the syntax.)

3.2 Semantics

The semantics of our constraint language is straightforward: a constraint is violated in a state iff one or more tuples in the Cartesian product of the listed tables satisfies the specified predicate. Constraints may be partially ordered: the user may specify for any pair of constraints that one constraint is to be enforced before the other (as long as there is no cycle in the prioritization). Constraint ordering allows the user to safely assume that some higher priority constraint will be satisfied whenever some lower priority constraint is considered. For example, in the constraint "each wire's voltage does not exceed the maximum for its wire type" (specified above), it is assumed that the constraint "each type in the wire table appears in the wire-type table" (specified below) already holds.

3.3 Examples

The referential integrity constraint "each type in the wire table appears in the wire-type table" is expressed:

```
wire: type not in (select type from wire-type)
```

1.	*Constraint*	::=	*Table-List* : *Predicate*
2.	*Table-List*	::=	T_1 [V_1], . . . , T_n [V_n]
3.	*Predicate*	::=	**exists** *Select-Exp*
4.		|	*Item-Exp Connector Select-Exp*
5.		|	*Item-Exp*$_1$ *Comp-Op Item-Exp*$_2$
6.		|	*Predicate*$_1$ **and** *Predicate*$_2$
7.		|	*Predicate*$_1$ **or** *Predicate*$_2$
8.		|	**not** *Predicate*$_1$
9.		|	(*Predicate*$_1$)
10.	*Select-Exp*	::=	**select** [**distinct**] *Val-Exps* **from** *From-List* [**where** *Predicate*]
11.		|	*Select-Exp*$_1$ *Set-Op Select-Exp*$_2$
12.		|	(*Select-Exp*$_1$)
13.	*Val-Exps*	::=	*
14.		|	*Val-Exp*$_1$, . . . , *Val-Exp*$_n$
15.	*Val-Exp*	::=	*Col-Name*
16.		|	constant
17.		|	*Fn*(*Val-Exp*$_1$, . . . , *Val-Exp*$_n$)
18.	*From-List*	::=	T_1 [V_1], . . . , T_n [V_n]
19.	*Item-Exp*	::=	*Val-Exp*
20.		|	< *Val-Exp*$_1$, . . . , *Val-Exp*$_n$>
21.		|	(**select** *Agg-Fn*([**distinct**] *Col-Name*) **from** *From-List* [**where** *Predicate*])
22.	*Col-Name*	::=	[T.] C
23.	*Connector*	::=	**in** | **not in**
		|	*Comp-Op* **any** | *Comp-Op* **all**
24.	*Set-Op*	::=	**union** | **intersect** | **minus**
25.	*Comp-Op*	::=	= | != | < | > | <= | >=
26.	*Agg-Fn*	::=	**sum** | **min** | **max** | **avg** | **count**
		|	user-defined aggregate function
27.	*Fn*	::=	+ | - | * | /
		|	user-defined function

Figure 2: *Constraint Language Syntax*

The constraint "each set of wires fits into its tube", with an interpretation that the sum of the wires' cross-sections must not exceed 80% of the cross-section of the tube, is expressed as:

```
tube: (select sum(cross-section)
       from wire, wire-type
       where wire.<fr,to> = tube.<fr,to>
       and wire.type = wire-type.type) > any
      (select 0.8 * cross-section
       from tube-type where type = tube.type)
```

Finally, the constraint "each plant has only outgoing tubes" can be expressed quite simply:

```
plant, tube: plant-id = tube.to
```

4 Deriving Constraint-Maintaining Rules

Once one or more constraints are defined using the language of the preceding section, the system automatically produces *rule templates*—portions of the production rules necessary for constraint enforcement. We first describe how, for a given constraint, a set of *invalidating operations* is derived. This set includes every data manipulation operation whose execution (in a consistent state) may result in a state violating the constraint—these become the triggering operations for the constraint-maintaining rule. We provide examples, describe optimizations to the derivation, and finally present the full form of rule templates, from which the user produces a preliminary set of constraint-maintaining rules.

4.1 Deriving Invalidating Operations

Given a constraint, we want to derive automatically the set of operations that may cause constraint violation. We achieve this through purely static analysis of the constraint. However, because semantic information is not incorporated, the set may be conservative: some operations in the set may not have real potential for constraint violation. (This does not affect correctness of constraint enforcement, but it may affect efficiency.) We provide optimizations to minimize the possibility of such extraneous operations.

The set of invalidating operations contains elements of the form `inserted into t`, `deleted from t`, and `updated t.c`, where `t` is a table name and `c` is a column of table `t`. (These translate directly into the `when` components of constraint-maintaining rules.) For a given constraint, the set is generated by analyzing the constraint's syntactic structure, based on the grammar of Fig. 2. We first give a method for deriving the invalidating update operations, then a method for deriving the invalidating insert and delete operations. In practice, all invalidating operations can be derived in the same syntactic analysis (parse) of the constraint.

4.1.1 Update Operations

To enumerate the invalidating update operations, we must determine every column for which, if the value of that column is changed in one or more tuples, then the constraint predicate may become true for some tuple(s) in the Cartesian product of the constraint's table list. Two types of column references may appear in a constraint:

1. Columns whose individual values are used directly in evaluating the constraint. With respect to the grammar of Fig. 2, these are columns appearing as part of an *Item-Exp*: as single values (productions 19 and 15), in constructed tuples (production 20), or as objects of functions or aggregate functions (productions 17 and 21).
2. Columns forming part of a set (due to inclusion in the result of a `select` operation). These are columns appearing in the *Val-Exps* part of a *Select-Exp* (production 10).

Any update to a column of the first type can certainly affect the value of the constraint predicate.

- If `t.c`[4] is a column name appearing in the constraint as part of an *Item-Exp*, then `updated t.c`

[4]The grammar allows use of table variables; it also allows a column to appear without its table name. In both cases, there is an understanding that the relevant table can unambiguously be inferred.

is an invalidating operation.

Invalidating update operations also are generated for the second type of reference, except if the *Select-Exp* appears as the direct or indirect object of an `exists`, when the actual values of the tuples are unimportant.[5] (A *Select-Exp* is a direct object of an `exists` if `exists` is applied to the set generated by the *Select-Exp*. A *Select-Exp* is an indirect object of an `exists` if it and one or more other *Select-Exp*'s are connected by *Set-Op*'s, with `exists` applied to the result.) Thus:

- If `t.c` appears in the *Val-Exps* portion of a *Select-Exp*, and the *Select-Exp* is not a direct or indirect object of an `exists`, then `updated t.c` is an invalidating operation. When *Val-Exps* is "`*`", `updated t.c` is an invalidating operation for every column `c` of every table `t` appearing in the corresponding *From-List*.

4.1.2 Insert and Delete Operations

Generating the invalidating insert and delete operations requires more thorough, context-dependent syntactic analysis. Consider, for example, a constraint definition including a deeply nested `select` operation. We want to determine whether inserts or deletes performed on the tables in the `select` may cause the constraint's predicate to become true. To do this, we (effectively) construct a parse tree for the constraint. A top-down labeling of the tree is performed, using labels I, D, and ID. The labels ultimately considered are those appearing on the table-name leaves of the parse tree—these labels indicate the invalidating operations. If a node for table `t` is labeled I, then `inserted into t` is an invalidating operation for the constraint, if the node is labeled D, then `deleted from t` is an invalidating operation, and if the node is labeled ID, then both `inserted into t` and `deleted from t` are invalidating operations.

We provide a rigorous description of the labeling process using an *attribute grammar*—a formalism specifically designed for defining how labels (attributes) are assigned to the nodes of a parse tree [ASU86]. The labeling scheme we use defines an *inherited attribute*, meaning that an initial label is assigned to the start symbol of the grammar (i.e., to the root of the tree); for each grammar production, labels for the symbols on the right-hand side are computed based on the label for the symbol on the left-hand-side (i.e., each child's label is computed as a function of its parent's label). In computing labels, we use the function *Opp* (for opposite), defined as: $Opp(I) = D$, $Opp(D) = I$, $Opp(ID) = ID$.

The equations for computing labels are given in Fig. 3; explanations are provided below. The equations correspond to the productions of the grammar in Fig. 2, with certain omissions and changes. Since we ultimately consider only the labels on table-name nodes, we ignore

[5]Of course, an update to a column `t.c` may affect whether tuples satisfy the `where` clause of the `select` (thus generating fewer or more tuples in the set over which `exists` is applied), but this happens only when the column actually appears in the `where` clause. In this case, `updated t.c` is already generated, since `t.c` must appear as part of an *Item-Exp*.

1.	*Predicate*.label = I
2.	T_i.label = I $\quad i = 1..n$
3.	*Select-Exp*.label = *Predicate*.label
4a.	*Select-Exp*.label = *Predicate*.label
4b.	*Select-Exp*.label = *Opp*(*Predicate*.label)
6,7.	*Predicate*$_1$.label = *Predicate*.label, *Predicate*$_2$.label = *Predicate*.label
8.	*Predicate*$_1$.label = *Opp*(*Predicate*.label)
9.	*Predicate*$_1$.label = *Predicate*.label
10.	*From-List*.label = *Select-Exp*.label, *Predicate*.label = *Select-Exp*.label
11a.	*Select-Exp*$_1$.label = *Select-Exp*.label, *Select-Exp*$_2$.label = *Select-Exp*.label
11b.	*Select-Exp*$_1$.label = *Select-Exp*.label, *Select-Exp*$_2$.label = *Opp*(*Select-Exp*.label)
12.	*Select-Exp*$_1$.label = *Select-Exp*.label
18.	T_i.label = *From-List*.label $\quad i = 1..n$
21.	*From-List*.label = ID, *Predicate*.label = ID

Figure 3: *Syntax-Directed Labeling Scheme*

any nonterminal that cannot eventually produce such a node. Thus, we eliminate from consideration nonterminals *Val-Exp(s)*, *Col-Name*, *Connector*, *Set-Op*, *Comp-Op*, *Agg-Fn*, and *Fn*. We also need not provide labels for any terminal symbols other than table names. The grammar is rewritten slightly to facilitate the labeling: we distinguish two classes of *Connector*'s, and we separate `minus` from the other two *Set-Op*'s. Productions 4, 11, and 23 are modified as follows:

4a.	*Predicate*	::=	*Item-Exp Connector*$_1$ *Select-Exp*
4b.		|	*Item-Exp Connector*$_2$ *Select-Exp*
11a.	*Select-Exp*	::=	*Select-Exp*$_1$ `union` *Select-Exp*$_2$
		|	*Select-Exp*$_1$ `intersect` *Select-Exp*$_2$
11b.		|	*Select-Exp*$_1$ `minus` *Select-Exp*$_2$
23a.	*Connector*$_1$	::=	`in` | `= any` | `!= all`
		|	`< any` | `> any` | `<= any` | `>= any`
23b.	*Connector*$_2$	::=	`not in` | `!= any` | `= all`
		|	`< all` | `> all` | `<= all` | `>= all`

Finally, because the right-hand-side labels for productions 1, 2, and 21 are defined irrespective of the left-hand-side labels, we need not consider nonterminals *Constraint*, *Table-List*, and *Item-Exp*.

We explain many of the labeling rules (by number). The remainder are left for the reader, and examples are provided below. As a general principle, an I label on a *Select-Exp* indicates that if an operation causes additional tuples in the `select`, then the constraint may become violated; a D label indicates that violation may occur if fewer tuples are in the `select`. On a *Predicate*, an I label indicates that operations making the predicate more likely to hold may cause the constraint to become violated, while a D label indicates that operations making the predicate less likely to hold may cause violation.

1. The I label on *Predicate* indicates that if the predicate is more likely to hold, then the constraint is more likely to be violated.

2. Insertions into top-level tables may cause new tu-

ples in the cross-product to satisfy the constraint predicate, violating the constraint. The I labels here imply that for every top-level table `t`, `inserted into t` is generated as an invalidating operation.

3. If additional tuples are in the *Select-Exp*, then the `exists` is more likely to hold. If fewer tuples are in the *Select-Exp*, then the `exists` is less likely to hold. Thus, the *Select-Exp* label is the same as the *Predicate* label.

4a. For connectors such as `in`, `= any`, `!= all`, ..., if additional tuples are in the *Select-Exp*, then the predicate is more likely to hold. If fewer tuples are in the *Select-Exp*, then the predicate is less likely to hold. 4b is the reverse.

10. If additional tuples are in the tables of the *From-List* or if the *Predicate* is more likely to hold, then additional tuples may be in the *Select-Exp*. Similarly for fewer tuples.

18. The label passed from the *From-List* to the table name indicates the invalidating operation(s) for that table.

21. For aggregate functions, both insertions and deletions (respectively, both predicates more likely and less likely to hold) may cause a change in the value of the result. Thus, both may cause the constraint to become violated.

We have now fully described how a set of invalidating operations is derived from a constraint definition. Note that a given constraint may be expressed in several different ways. We would like our algorithm to produce the same set of invalidating operations for any two equivalent constraints. In the technical report [CW90], we discuss several classes of equivalence, showing that our algorithm does produce equivalent invalidating operations.

4.2 Examples

Consider again the constraint "each wire's voltage does not exceed the maximum for its wire type":

```
wire: voltage > any (select max-voltage
                     from wire-type
                     where type = wire.type)
```

The columns appearing in the constraint as *Item-Exp*'s are `wire.voltage`, `wire-type.type`, and `wire.type`; the only column appearing in the *Val-Exps* part of a *Select-Exp* is `wire-type.max-voltage`. Thus, the invalidating update operations are:

```
updated wire.voltage, updated wire.type,
updated wire-type.type,
updated wire-type.max-voltage
```

For inserts and deletes, our algorithm first labels the constraint predicate and table `wire` with I (equations 1 and 2 in Fig. 3). Then, by equation 4a (since `> any` is a *Connector*$_1$), label I is passed to the `select` expression and subsequently to table `wire-type`. Hence, the invalidating insert and delete operations are:

```
inserted into wire, inserted into wire-type
```

As a second example, consider the constraint "each tube contains at least one wire", expressed as:

```
tube: not exists (select * from wire
                  where <fr,to> = tube.<fr,to>)
```

Since the `select` operation is the object of `exists`, we need not enumerate as invalidating update operations all columns in table `wire`. Considering all other mentioned columns, the invalidating update operations are:

```
updated wire.fr, updated wire.to,
updated tube.fr, updated tube.to
```

For inserts and deletes, table `tube` and the predicate are initially labeled with I. Table `wire` inherits label D, however, since the appearance of "`not`" causes propagation of the opposite label (equation 8). Thus, the invalidating insert and delete operations are:

```
inserted into tube, deleted from wire
```

4.3 Labeling Optimization for Aggregate Functions

When an aggregate function appears in a constraint, our labeling procedure for generating insert and delete operations assigns ID to the relevant *From-List* and *Predicate* (equation 21 in Fig. 3). This is necessary because both insertions and deletions (respectively, predicates more likely and less likely to hold) may cause a change in the result of the aggregate. For certain aggregate functions in combination with certain comparison operators, however, we can determine that only one operation with respect to the *From-List* (insert or delete) can change the value of the entire expression; similarly for the *Predicate*. For example, consider the following expression:

(select max(*Col-Name***)**
from *From-List* **where** *Predicate***) >** *Item-Exp*

Our labeling algorithm assigns ID to the *From-List* and the *Predicate*, independent of the label inherited by the expression. By the monotonicity of function `max`, however, only insertions into the tables of the *From-List* or a *Predicate* more likely to hold can increase the value of `max` (making the entire expression more likely to hold); deletions or a *Predicate* less likely to hold can only decrease the `max`. Therefore, if the expression inherits label I, the *From-List* and *Predicate* can be labeled I. Symmetrically, if the expression inherits D, the *From-List* and *Predicate* can be labeled D. When the *From-List* or *Predicate* is complex, this optimization may significantly reduce the number of invalidating operations.

Similar optimizations apply for several other combinations of aggregate functions and comparison operators. Consider any expression of the form "*Item-Exp*$_1$ *Comp-Op* *Item-Exp*$_2$" or of the form "*Item-Exp*$_1$ *Connector* *Select-Exp*", where *Item-Exp*$_1$ is an aggregate. Let L denote the label inherited by the expression. The table in Fig. 4 summarizes all applicable optimizations based on the particular aggregate function (column 1) and the particular comparison operator or connector (column 2). Column 3 indicates how the components of the aggregate `select` operation are to be labeled,

AGGR.	OPERATOR/CONNECTOR	LABEL
min	<, <=, < any, <= any, < all, <= all	L
min	>, >=, > any, >= any, > all, >= all	$Opp(L)$
max	<, <=, < any, <= any, < all, <= all	$Opp(L)$
max	>, >=, > any, >= any, > all, >= all	L
count	<, <=, < any, <= any, < all, <= all	$Opp(L)$
count	>, >=, > any, >= any, > all, >= all	L
sum	<, <=, < any, <= any, < all, <= all	$Opp(L)$
sum	>, >=, > any, >= any, > all, >= all	L

Figure 4: Labeling Optimization for Aggregates

based on the inherited label L. We include optimizations for function sum, although these assume that only positive values are considered. No optimizations apply to function avg or to operators = and != . If, in an expression of the form "*Item-Exp*$_1$ *Comp-Op* *Item-Exp*$_2$", *Item-Exp*$_2$ is an aggregate, then the optimizations are the same as in Fig. 4, inverting the comparison operators appropriately (e.g., < becomes > and >= becomes <=). If both *Item-Exp*$_1$ and *Item-Exp*$_2$ are aggregates, optimizations can independently be applied to each.

These optimizations are relevant to several of the constraints in our case study. Consider, for example, the constraint "the total outgoing power for each plant does not exceed the produced power", expressed as:

```
plant: power < (select sum(power) from wire
                where fr = plant-id)
```

Without optimization, the labeling algorithm produces as invalidating insert and delete operations:

```
inserted into plant, inserted into wire,
deleted from wire
```

However, since power takes on only positive values, we apply the optimization for aggregate function sum, eliminating deleted from wire as an invalidating operation.

Other, more complex, labeling optimizations are also possible—there is certainly room for future work here. (For further discussion, see [CW90].)

4.4 Rule Templates and Constraint-Maintaining Rules

Once the set of invalidating operations is generated for a given constraint, the system can produce a rule template. The rule components that are completed automatically in the template are the transition predicate and the condition (recall Section 2.2). The user must provide a name and an action.

Consider a constraint of the form *Table-List*: *Predicate*. From this constraint, a set *Inv-Ops* (say) of invalidating operations is derived. The rule template produced for this constraint has the form:

```
create rule <NAME>
when Inv-Ops
if exists T:(select * from Table-List
             where Predicate)
then <ACTION>
```

The set of invalidating operations is translated directly into the rule's transition predicate: we want the rule to be triggered whenever any operation occurs that may cause the constraint to be violated. Once the rule is triggered, we want it to check if the constraint actually is violated. This behavior is achieved by translating the constraint definition into the if clause of the rule. The translation, illustrated above, is based on the semantics of our constraint language: the constraint is violated if there exists any tuple in the Cartesian product of the *Table-List* satisfying the *Predicate*. The select operation here is (automatically) labeled T as a matter of notational convenience. Often, in the action part of the rule, the user wants to refer to this particular set, since it contains exactly those tuples violating the constraint. We allow references to logical table T in the action, with an interpretation equivalent to textual expansion.

As an example, consider the constraint "each wire is contained within a tube", expressed as:

```
wire: not exists (select * from tube
                  where <fr,to> = wire.<fr,to>)
```

The rule template produced for this constraint is:

```
create rule <NAME>
when inserted into wire, updated wire.fr,
     updated wire.to, deleted from tube,
     updated tube.fr, updated tube.to
if exists T:(select * from wire
             where not exists
                (select * from tube
                 where <fr,to> = wire.<fr,to>))
then <ACTION>
```

Suppose the user decides that the appropriate compensating action here is to first delete all wires whose tubes were deleted, then to insert new tubes for remaining "tubeless" wires. To complete the template, the user gives the rule a <NAME>, and replaces <ACTION> by:

```
/* delete wires whose tubes were deleted */
delete from wire where
   wire-id in (select wire-id from T)
   and <fr,to> in (select fr, to
                   from deleted tube);
/* assign tubes to remaining wires */
insert into tube
  (select new-tube-id(), X.f, X.t,
          default-tube-type
   from X(f,t):(select distinct fr, to from T))
```

In the first part of the action, logical table T and transition table "deleted from tube" are used to find all wires whose tubes have been deleted; these wires are deleted. In the second part, logical table T is referenced again—recall the interpretation is textual expansion—within a table expression[6] that generates all fr,to pairs with "tubeless" wires. For each such pair, a tube is inserted with a new identifier and a default type. If the default type turns out to be inadequate, rules enforcing constraints on tube types will be triggered and will update the type appropriately.

When multiple constraints are defined, one con-

[6]This table expression should be self-explanatory; for details on the construct see [CW90,HF*89].

straint-maintaining rule is produced for each.[7] If constraints are partially ordered, ordering is automatically transferred to the rules. That is, if a constraint C_1 is to be enforced before a constraint C_2, then rules R_1 and R_2—for enforcing constraints C_1 and C_2 respectively—are ordered so that if both are triggered, R_1 will be considered first.

Finally, although we have been assuming a scenario in which a single user defines a number of constraints in "one sitting", our rule derivation facility easily can be used at multiple times and by multiple users, with appropriate integration. This is further discussed in Section 8.

5 Rule Analysis

When multiple rules are defined, execution of one rule's action may trigger a number of other rules. This behavior is crucial to enforcing multiple constraints, as discussed in detail in Section 7. Unfortunately, this behavior also produces the possibility of cyclic, infinite rule execution: rules may trigger each other indefinitely. (This will certainly happen, in fact, if the user tries to enforce conflicting constraints.) Although we do not attempt to determine whether given sets of rules are guaranteed to terminate—this problem almost certainly is undecidable in the general case—we do perform simple, conservative analysis of rule sets, indicating subsets of rules for which infinite triggering is possible. In many cases, the warnings produced may not be relevant. It may be known, for example, that once one rule's action is executed, all other rules' conditions will not hold. However, the user should validate for each potential cycle that termination in finite time is guaranteed.

We use a straightforward method for detecting potential cycles in rule triggering behavior. For a given set of rules, a *triggering graph* is constructed. The nodes of the graph correspond to the rules in the set. There is a directed edge from node R_i to node R_j iff execution of rule R_i's action can trigger rule R_j. Edges are determined by simple syntactic analysis: R_i can trigger R_j, $i \neq j$, if any data manipulation operation in R_i's action corresponds to an operation listed in R_j's transition predicate. Once the triggering graph is constructed, each cycle indicates potential for nontermination.

The user should inspect each cycle in the triggering graph, determining whether infinite triggering is actually possible. If so, relevant rules should be modified to eliminate this possibility. A triggering graph has been built for the preliminary rule set in our case study; the graph contains a number of cycles. All cycles except two are clearly not relevant. It is interesting to note, however, that for the two cases in which infinite triggering is possible, this fact was revealed only by formal analysis—we were unaware of the problem in our initial definition of the rules. (See [CW90] for details.)

[7]We may want to extend our facility to allow multiple rules enforcing a given constraint. This accommodates the possibility that different compensating actions may be appropriate for different invalidating operations. We plan to incorporate this extension.

More sophisticated rule analysis is certainly possible. It would involve additional syntactic analysis of the rules in each cycle, perhaps incorporating semantic information as well. The goal would be to determine automatically, in as many cases as possible, that infinite triggering will not occur. This is a topic for future research.

6 Rule Optimization

The rule analysis phase is complete when all necessary modifications have been made to prevent infinite rule triggering. At this point, we have a "final" set of constraint-maintaining rules. As illustrated in the system structure diagram of Fig. 1, these rules may further be processed by a *rule optimizer*. Here, as in other parts of the system, varying degrees of analysis can be performed with varying quality of results. We describe one fairly straightforward optimization, applicable to a wide class of rules derived from constraints.

The condition parts of constraint-maintaining rules are produced by a simple transformation on constraint definitions. Thus, rule conditions are static predicates on the database state. Sometimes, however, it is possible to evaluate a rule's condition only over the changes that have occurred since the constraint was last considered. (This may correspond to only a fraction of the database.) For example, suppose a constraint may be violated when tuples are inserted into a table. It may be sufficient for the condition part of the rule enforcing the constraint to inspect only those tuples that have been inserted, rather than inspecting the entire table. (References in the rule's action to pre-defined table `T`, derived from the condition, consequently also are improved.) We describe an automatic method for transforming rule conditions to incorporate this kind of optimization. The optimized conditions use the transition table feature of our rule language, which provides a mechanism for referring to database changes (recall Section 2.2).

Let R be the rule enforcing a constraint C. For the optimization, we consider the tables from C's table list, transforming references to these tables appearing in R's condition into references to appropriate transition tables. Here, we consider the case in which the table list contains one table; the generalization to multiple tables is straightforward and appears in [CW90].

Consider a rule R derived from a constraint with one table `t` in its table list. R's condition thus looks like:

```
if exists T:(select * from t where Predicate)
```

Our optimization is applicable to this condition if:

1. all operations listed in R's transition predicate are operations on table `t`;
2. table `t` does not appear in the `from` clause of any nested `select` operation in the *Predicate*.

(Note that by the second requirement and our algorithm for generating invalidating operations, `deleted from t` cannot appear in R's transition predicate.) If these requirements are met, then the `from` clause reference to t in the condition part of the rule can be replaced by the union of all transition tables corresponding to operations in R's transition predicate. That is, let R's transition predicate be:

```
when inserted into t,
     updated t.c1, ..., updated t.cn
```

After optimization, R's condition becomes:

```
if exists T:(select *
             from t:(inserted t
                     union new updated t.c1
                           ...
                     union new updated t.cn)
             where Predicate)
```

Notice that we use the new values of the updated tuples. Also, the union in the from clause is named t so that references to t within the *Predicate* are handled appropriately.[8] Assuming that the inserted and updated tuples form only a small portion of the original table t, this transformation can yield significant improvement in rule condition evaluation. Improvement may also be gained in the rule's action, since references to logical table T inherit the optimization. Correctness of the transformation is proven formally in [CW90].

The restrictions for applying our optimization may seem rather limiting, but in fact rules derived from constraints often meet the requirements. (Exactly half of the rules in our case study are eligible.) As an example, again consider the constraint "each wire's voltage does not exceed the maximum for its wire type". The enforcing rule's transition predicate and condition are:[9]

```
when inserted into wire, updated wire.type,
     updated wire.voltage
if exists T:(select * from wire
             where voltage > any
                (select max-voltage
                 from wire-type
                 where type = wire.type))
```

Here, in the condition, voltage is checked for all wires in the database. However, after optimization the condition is transformed to:

```
if exists T:(select *
             from wire:
               (inserted into wire
                union new updated wire.type
                union new updated wire.voltage)
             where ...
```

Here, the only wires checked are those that have been inserted or whose type or voltage has changed.

7 System Execution

So far, we primarily have considered the static aspects of our facility. Constraint definition, rule derivation, rule analysis, and optimization all are performed prior to system execution. Although execution time is not the focus of our study, we still must ensure that derived rules will behave as desired—that consistency with respect to all constraints is guaranteed.

Since rules are activated at the end of each transaction, we consider an arbitrary transaction, showing that rule execution will terminate and, when it does, all constraints will be valid. To do this, we must make certain assumptions about those aspects of the constraints-to-rules translation performed by the user. Correctness requires proving that consistency is restored—this obviously relies on appropriate compensating actions. Thus, we assume that the rule created by a user from the rule template for a given constraint, considered in isolation, does indeed restore consistency for that constraint:

Assumption 7.1 (Compensating Actions) Let C be a constraint and let R be the rule enforcing that constraint. Following any database transition triggering rule R, if R's condition holds and its action is executed, then the resulting state is consistent with respect to C.

When many constraints are enforced, rules executed to restore one constraint may cause others to be violated (and therefore additional rules to be triggered). This behavior is acceptable as long as it cannot proceed indefinitely. We prove that termination is guaranteed, under the assumption that the user properly validates all cycles produced by the rule analyzer:

Assumption 7.2 (Cycle Termination) Every potential rule triggering cycle will execute only a finite number of times.

7.1 Termination

Consider system execution when a set of constraint-maintaining rules $R_1, \ldots, R_n$ is defined. To prove that rule execution terminates at the end of every transaction, we first provide a formal description of system execution. Execution is modeled by a sequence: the first symbol in the sequence is τ, denoting the initial transaction. Each subsequent symbol is an R_i, $1 \leq i \leq n$, denoting execution of rule R_i's action. After each symbol in the sequence, we insert the set of rules triggered at that point. Recall from Section 2.2 that a rule is considered with respect to the net effect of the transition since the rule was last executed (or evaluated), or since the start of the transaction if it has not yet been evaluated. Thus, through the sequence, rules may appear and disappear from the set of triggered rules before their action is executed. When a rule finally is executed, we say it was "initially triggered by" the symbol preceding its most recent uninterrupted appearance in the set of triggered rules.

Now suppose, for the sake of a contradiction, that an execution sequence is infinite. We show that the sequence must then contain some triggering cycle executed infinitely often, contradicting Assumption 7.2. The following lemma is used:

Lemma 7.3 Let σ be an infinite string over a finite alphabet. For any $k > 0$, there is a substring s of σ

[8]Strictly speaking, we should assign a new name and an appropriate column list to this table expression, changing references accordingly. The details are not particularly interesting and therefore are omitted.

[9]Invalidating operations on table wire-type have been omitted since, in the case study, this table is considered read-only [CW90].

such that $|s| = k$ and an infinite number of disjoint occurrences of s appear in σ.

Proof: Partition σ into substrings of length k. Since there are only a finite number of strings of length k, some k-length substring must appear in σ an infinite number of times. □

Applying this lemma to our execution sequence with a sufficiently large k, we obtain the desired result. First, we modify the execution sequence model to explicitly contain "triggered by" information. Each element in the sequence (except the first) becomes a pair: the second half of the pair indicates the executed rule, while the first half—τ or a rule—indicates the symbol initially triggering the executed rule.

Theorem 7.4 (Termination) *Let σ be an infinite sequence representing rule execution. Some rule triggering cycle appears infinitely often in σ.*

Proof: Since each rule can be triggered by τ at most once, eventually in σ symbol τ no longer appears. Consider any infinite tail σ' of σ for which this is true. Each symbol in σ' is an $\langle R_i, R_j \rangle$ pair $(i \neq j)$, where R_j was triggered by R_i. A simple counting argument shows that if we consider any subsequence of σ' of length $\sum_{i=1}^{n} n^i$, where n is the number of rules, the subsequence must contain a triggering cycle. Now, by Lemma 7.3, there is some subsequence s of σ' such that $|s| = \sum_{i=1}^{n} n^i$ and s appears infinitely often in σ'. Since s must contain a cycle, some cycle appears infinitely often in σ. □

7.2 Correctness

Now that we know the constraint-maintaining rules are guaranteed to terminate, we must prove that they always terminate in a consistent state. Again, consider an arbitrary initial transition and subsequent rule execution. Since we assume that each transaction begins in a consistent state (the constraint-maintaining rules ensure this for all but some "first" transaction—see [CW90] for details), any `rollback` action trivially guarantees correctness. Thus, assume no `rollback` action is executed.

We again use a sequence model of system execution. In this case, we need not include triggering information, but we do need to include all rules evaluated, not just those whose actions are executed. Thus, execution is represented by a sequence in which the first symbol is τ and each subsequent symbol is an R_i, $1 \leq i \leq n$, denoting evaluation of rule R_i. By Theorem 7.4, every such execution sequence is finite.[10] Hence, we only must show that, at the end of the sequence, all constraints are valid.

Theorem 7.5 (Correctness) *Consider any finite sequence representing system execution. At the end of the sequence, all constraints are valid.*

Proof: At the end of the sequence, there are no triggered rules with true conditions. Thus, if a rule R_i enforcing a constraint C_i is triggered, then its condition does not hold; hence C_i is valid. Consider a rule R_j (for a constraint C_j) that is not triggered at the end of the sequence. We consider two cases:

1. R_j does not appear in the execution sequence: R_j was never evaluated; hence, at the end of the sequence, R_j is considered with respect to the transition since the start of the initial transaction. If the net effect of this transition includes any operations that might invalidate C_j, then R_j would be triggered. Thus, C_j cannot have been invalidated since the start of the transaction.
2. R_j does appear in the execution sequence. Then, at the end of the sequence, R_j is considered with respect to the transition since it was last executed or—if its condition was most recently false—since it was last evaluated. By Assumption 7.1, in either case validity of C_j was established. If C_j could have been invalidated since then, R_j would be triggered. Hence C_j must still be valid. □

8 Conclusions and Future Work

We have described a general framework for transforming constraints into constraint-maintaining production rules. The facility allows users to define general constraints at a natural, conceptual level—as predicates on database states—and to systematically derive "lower-level" rules guaranteed to enforce the constraints.

The constraints-to-rules process has been described as if one user defines all constraints (and consequently derives all rules) at one time. Actually, it is equally possible for constraints to change over time: some initial set of constraint-maintaining rules is derived; later, additional rules may be added and existing rules may be removed. The only point to note is that whenever new rules are derived, rule analysis must be applied to the entire set of rules (new and old), since, at execution time, all rules coexist and interact. If multiple users are defining constraints for a particular application, an authorization facility might be useful, since it may not be desirable to have all constraints and rules accessible to all users.

We plan to implement the basic components of the facility and begin experiments. This should not be too difficult; most of our algorithms are based on syntactic analysis and transformation so parser-generator tools can be used. The next step will be to implement a convenient user-interface—the *Constraints Editor* and *Rules Editor* shown in Fig. 1. We also plan to continue work on extending the functionality of the system, as improvements can be made in many respects. In particular, we would like to:

- incorporate additional labeling optimizations for less conservative derivation of invalidating insert and delete operations;
- perform more complete rule analysis to eliminate trivially impossible cycles;
- define additional rule optimizations;

[10]Actually, for Theorem 7.4 we considered only the executed rules, not all rules evaluated. However, the result may be applied directly to show that these sequences also are guaranteed to be finite.

• allow multiple rules enforcing a single constraint.

Finally, as a broad area of future research, we plan to explore the possibility of automatically (or semi-automatically) deriving compensating actions.

Acknowledgements

We are grateful to Bruce Lindsay, for useful discussions and comments on a preliminary draft, to Guy Lohman, for carefully reading a "final" draft, to Alex Aiken, for assistance in proving Theorem 7.4, to Shel Finkelstein, for pointing out the correct interpretation of logical table T, and to Raymond Lorie, for suggesting the idea of constraints-to-rules translation some time ago.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1986.

[BBC80] P.A. Bernstein, B.T. Blaustein, and E.M. Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Proc. of Sixth VLDB*, pages 126–136, October 1980.

[CG88] S. Ceri and F. Garzotto. *Specification and Management of Database Integrity Constraints through Logic Programming.* Technical Report 88-025, Dip. di Elettronica, Politecnico di Milano, 1988.

[CTF88] M.A. Casanova, L. Tucherman, and A.L. Furtado. Enforcing inclusion dependencies and referential integrity. In *Proc. of Fourteenth VLDB*, pages 38–49, August 1988.

[CW90] S. Ceri and J. Widom. *Deriving Production Rules for Constraint Maintenance.* IBM Research Report RJ7348, IBM Almaden Research Center, March 1990.

[DE89] L.M.L. Delcambre and J.N. Etheredge. The Relational Production Language: a production language for relational databases. In *Expert Database Systems—Proc. from the Second Int. Conference*, pages 333–351, Benjamin/Cummings, 1989.

[EC75] K.P. Eswaran and D.D. Chamberlin. Functional specifications of a subsystem for data base integrity. In *Proc. of First VLDB*, pages 48–67, September 1975.

[Han89] E.N. Hanson. An initial report on the design of Ariel: a DBMS with an integrated production rule system. *SIGMOD Record*, 18(3):12–19, September 1989.

[HF*89] L.M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. of ACM-SIGMOD*, pages 377–388, May 1989.

[HI85] A. Hsu and T. Imielinski. Integrity checking for multiple updates. In *Proc. of ACM-SIGMOD*, pages 152–168, May 1985.

[HMN84] L.J. Henschen, W.W. McCune, and S.A. Naqvi. Compiling constraint-checking programs from first-order formulas. In *Advances in Database Theory, Volume 2*, pages 145–169, Plenum Press, 1984.

[IBM88] *IBM Systems Application Architecture, Common Programming Interface: Data-base Reference.* IBM Form Number SC26-4348-1, October 1988.

[KMS90] J. Kiernan, C. de Maindreville, and E. Simon. Making deductive databases a practical technology: a step forward. In *Proc. of ACM-SIGMOD*, pages 237–246, May 1990.

[KP81] S. Koenig and R. Paige. A transformational framework for the automatic control of derived data. In *Proc. of Seventh VLDB*, pages 306–318, September 1981.

[MD89] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proc. of ACM-SIGMOD*, pages 215–224, May 1989.

[Mor83] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proc. of Ninth VLDB*, pages 34–42, October 1983.

[Mor84] M. Morgenstern. Constraint equations: declarative expression of constraints with automatic enforcement. In *Proc. of Tenth VLDB*, pages 291–300, August 1984.

[Nic82] J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.

[QS87] X. Qian and D.R. Smith. Integrity constraint reformulation for efficient validation. In *Proc. of Thirteenth VLDB*, pages 417–425, September 1987.

[SJ*90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. of ACM-SIGMOD*, pages 281–290, May 1990.

[SK84] A. Shepherd and L. Kerschberg. Prism: a knowledge based system for semantic integrity specification and enforcement in database systems. In *Proc. of ACM-SIGMOD*, pages 307–314, May 1984.

[SMS87] D. Stemple, S. Mazumdar, and T. Sheard. On the modes and meaning of feedback to transaction designers. In *Proc. of ACM-SIGMOD*, pages 374–386, May 1987.

[Sto75] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proc. of ACM-SIGMOD*, pages 65–78, May 1975.

[UD90] S.D. Urban and M. Desiderio. Translating constraints to rules in CONTEXT: a CONstrainT EXplanation Tool. Manuscript, 1990.

[UD91] S.D. Urban and L.M.L. Delcambre. Constraint Analysis: a design process for specifying operations on objects. To appear in *ACM Transactions on Data and Knowledge Engineering*, 1991.

[Wal89] J.A. Wald. Implementing constraints in a knowledge base. In *Expert Database Systems—Proc. from the Second Int. Conference*, pages 163–183, Benjamin/Cummings, 1989.

[WF90] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. of ACM-SIGMOD*, pages 259–270, May 1990.