VLDB Journal, 2, 173–213 (1993), Georg Gottlob, Editor
©VLDB

Generating Consistent Test Data: Restricting the Search Space by a Generator Formula

Andrea Neufeld, Guido Moerkotte, Peter C. Lockemann

Received September 4, 1991; revised version received August 3, 1992; accepted October 21, 1992

Abstract. To address the problem of generating test data for a set of general consistency constraints, we propose a new two-step approach: First the interdependencies between consistency constraints are explored and a generator formula is derived on their basis. During its creation, the user may exert control. In essence, the generator formula contains information to restrict the search for consistent test databases. In the second step, the test database is generated. Here, two different approaches are proposed. The first adapts an already published approach to generating finite models by enhancing it with requirements imposed by test data generation. The second, a new approach, operationalizes the generator formula by translating it into a sequence of operators, and then executes it to construct the test database. For this purpose, we introduce two powerful operators: the generation operator and the test-and-repair operator. This approach also allows for enhancing the generation operators with heuristics for generating facts in a goal-directed fashion. It avoids the generation of test data that may contradict the consistency constraints, and limits the search space for the test data. This article concludes with a careful evaluation and comparison of the performance of the two approaches and their variants by describing a number of benchmarks and their results.

Key Words. Design, validation, logic, test data, consistency.

1. Introduction

Database design can be viewed as a process akin to specification in software development: It aims at capturing a given section of the application domain called the universe of discourse (UoD) and codifying it as a *semantic schema.* One of

Andrea Neufeld, Dipl. Inform., is Research Scientist; Guido Moerkotte, Dr.rer.nat., is Assistant Professor; Peter C. Lockemann, Dr.ing., is Professor, Universität Karlsruhe, Fakultät für Informatik, W-7500 Karlsruhe, Germany.

the principles of good software engineering is to validate the specification before implementation takes place. By starting the validation early, design errors can be identified before they become too costly. This is especially important in the case of database design where the schema may be used for decades.

In the database design process, the first formal representation is the semantic schema. The formalism used is the *semantic data model.* Unfortunately, these models are either general-purpose (unable to reflect all the idiosyncrasies of a particular UoD) or special-purpose (satisfying only the rules governing a very narrow UoD.) In either case, the semantic schema captures only part of the characteristics of an arbitrary UoD. The remainder must be formulated outside the schema, where they are referred to as *consistency constraints.* Consistency constraints should obey some formalism, such as first-order logic.

Schema and constraints succeed the informal representations used in the requirements analysis and the even more informal intentions of the user. Bridging the gap between the informal and formal worlds is often delegated to an expert designer by prospective users. Users, then, should have a method of confirming that their intentions have indeed been met by the specifications. Furthermore, users should be given the opportunity to experience the consequences of the specification, e.g., by entering and updating data or retrieving data from the database, and by comparing these to their intentions (a technique referred to as *rapid prototyping*). Experimenting with a prototype determines the adequacy of a specification in the intuitive sense of its *completeness* (does the specification exclude potentially interesting worlds?) and its *correctness* (does the specification accept undesirable worlds?).

Suppose that a database is consistent with the specification. While entering new data, a user may be confronted with a rejection due to the violation of some part of the specification. In such a case, the user may inspect the cause of the rejection and decide whether completeness has been violated and part of the specification should be adjusted. To test the correctness of the specification, the user should try to insert or update information that violates his or her view of possible worlds. If rejection does not occur, the specification must be adapted. Posing queries against the database, particularly if they aim at derived data, is also an essential mechanism to detect loopholes in the specification. Generated answers may miss expected facts or include unexpected facts, again hinting at an inappropriate specification.

Prototyping in database design corresponds to experimenting with a test database. For experiments to have meaning to the designer, tests and results should be in terms of the semantic schema. Consequently, what we have to consider is the generation and manipulation of a semantic-level database. Such a database could be defined within a uniform framework of consistency constraints, because semantic schemas (and database schemas in general) may themselves be interpreted in terms of first-order formulas.

To perform meaningful experiments raises a number of questions. How large an initial database must be generated so that constraints are not trivially fulfilled? How should the generation process guarantee that "significant" violations of completeness and correctness are detected after a fair number of experimental updates or queries? Because some constraints have to be readjusted and a new database generated after each detection of an inadequacy, how can "sufficient" efficiency of the generation process be achieved?

The issue of this article, then, is the efficient generation of test databases of controlled size, which conform to an arbitrary set of consistency constraints. We propose a two-step approach. First, the interdependencies between the consistency constraints are explored and a generator formula is derived on their basis. During its creation, the user can exert some control in order to streamline subsequent experiments, and can include information in order to restrict the search for possible consistent test databases, thus accelerating the generation.

In the second step, the test database is generated. Here, two different routes are taken. The first adapts a published approach for the generation of finite models by enhancing it with the requirements imposed by test data generation. The second route operationalizes the generator formula by translating it into a sequence of operators. For this purpose, we introduce two powerful operators: the generation operator and the test-and-repair operator. We enhance the generation operator with heuristics which enable it to generate facts in a goal-directed fashion. This avoids the generation of test data that could contradict the consistency constraints, and limits the search space for the test data. Another feature of the second route is the repair mechanism incorporated in the test-and-repair operator. It uses a form of intelligent backtracking realized by a trace with the effect of increased flexibility and better performance.

The outline of this article is as follows. The next section discusses related literature. In order to provide a common basis, Section 3 introduces the necessary preliminaries. Section 4 introduces the first step of our approach, i.e., the process of creating the generator formula in a user-controlled way. Section 5 presents two different routes that actually generate the test database. In Section 6 we compare the different approaches by a number of benchmarks. Section 7 concludes the article.

2. Related Literature

Related literature can be divided roughly into two groups. One discusses rapid prototyping (especially database prototyping), while the other is concerned with the problem of finite model generation.

There is a rich body of literature on rapid prototyping in general and some on database prototyping (Alavi, 1984; Budde et al., 1984; Oberweis et al., 1986; Schönthaler, 1989). However, test data generation as a means for schema evaluation is not discussed. By far the most effort has gone into automatic generation of test databases for performance evaluation, where data quality is not measured in terms of meeting complicated constraints, but in terms of the distribution of data values that have an impact on performance. Efforts toward generating such databases—

referred to as benchmarks—are described in Bitton et al. (1983) and DeWitt (1985). Stonebraker (1985) recommends that the amount of data be predictable. In Bitton et al. (1983), random number generators with different scopes are used to obtain uniformly distributed attribute values for benchmarking relational database systems. A bit closer to our approach are QIKSYS (Noble, 1983) and the TDGL-system (Neugebauer and Neumann, 1985), because they consider interdependencies between attributes in relational database systems. Both systems, however, limit themselves to a few specialized constraint types: QIKSYS emphasizes specifying and generating test data that obey various inclusion and functional dependencies; the TDGL-system only considers inclusion dependencies. Other approaches (Mannila and Räihä, 1989; Röhrle, 1989) study how to generate test data for a given relational query, e.g., to validate user-defined transactions or to improve the learning of a new relational query language. The limit of all these approaches is that the generation procedure considers only a small range of the inter-data-dependencies that may occur within a UoD.

The efforts of Silva and Melkanoff (1981) are even closer to our approach. These authors present a database design tool that generates an Armstrong relation for given functional and multivalued dependencies. This relation precisely obeys only the specified dependencies (and their logical consequences). The technique used in this context is known as the *tableau chase* (Ullman, 1988). Starting with the hypotheses of the dependencies that form the *tableau,* (i.e., a relation), the given dependencies are applied to this tableau as follows: For a tuple-generating dependency (e.g., a multivalued dependency), a tuple (the conclusion) is added to the tableau. For an equality-generating dependency (e.g., a functional dependency) the identifiers are made equal within the tableau as required. While removing violations of the dependencies, the chase produces the required relation. One may classify the underlying idea as one of "repairing" a database until it meets all constraints (see Section 5).

Generating a consistent test database corresponds to the problem of constructing a finite model for a given set of logical formulas. There are a few approaches on this topic within the context of logical databases. All attempt to satisfy a set of general consistency constraints. A set of formulas is finitely satisfiable if there is at least one *finite* model satisfying all the formulas in the set. Finite satisfiability as well as unsatisfiability are known to be semi-decidable.

A tableaux approach to finite model generation is presented by Kung (1985). He introduces a modified existential quantifier to avoid repeated rule applications on the same formula. This approach has been demonstrated to be neither correct nor complete for finite satisfiability (Geibel, 1991). Bry and Manthey (1986) address termination problems and propose several approaches to avoid infinity while checking satisfiability. Manthey and Bry (1987, 1988) describe a method for checking constraint satisfiability, based on a proof procedure called SATCHMO. SATCHMO is complete for unsatisfiability, but not for finite satisfiability: for the clauses $\{\{\neg p(x), p(f(x))\}, \{p(a)\}\}$ with f a Skolem function—resulting from an existential quantifier—SATCHMO generates an infinite series of facts $p(a), p(f(a)), p(f(f(a))), \ldots$. This drawback was overcome with case analysis for existentially quantified formulas (Bry et al., 1988). Finite satisfiability is shown in a constructive way, i.e., the procedure stops whenever a finite model has been constructed. The construction process is viewed as a sequence of successive updates, each adding new facts. Each update causing a constraint violation is attempted to be repaired by deriving further updates. If no repair is possible by updating, Prolog-derived chronological backtracking takes place.

From our point of view, SATCHMO comes closest to our approach, but has several drawbacks. We require more than finite model generation for satisfiability. First, for the generation of consistent test databases, database size should be included to generate specified quantities of test data (Stonebraker, 1985). Second, mnemonically pleasing constants must be generated to support the validation process. Third, the appearance of the model should be directable, i.e., the user should be able to specify which parts of the schema should be instantiated.

Finite satisfiability can be viewed as a prerequisite for generating test data. Because test data generation is an extended form of systematic model generation, the method proposed in this article also can be applied to check satisfiability. If a test database can be constructed, satisfiability is discovered; if the process terminates without having generated a test database, unsatisfiability is proven.

SATCHMO is a starting point for test data generation, but it introduces a bias into the generation, namely the emphasis on satisfiability. Therefore, it seems reasonable to compare it to an approach which emphasizes control over the test data (see Sections 5 and 6).

3. Preliminaries

First, to provide a common basis, we introduce some basic definitions. Subsequently, the statement of the problem addressed in this article is given in terms of these definitions. Finally, we sketch the scenario in which the test data generation will be embedded.

3.1 Basic Definitions

3.1.1 Formulas. We distinguish three sets of symbols: set V of variable symbols, set C of constant symbols, and set P of predicate symbols. Variables are denoted by $x, y, z, \ldots$ possibly with subscripts. Constants are denoted by $a, b, c, \ldots$. Predicates are denoted by $p, q, r, \ldots$. There is an arity associated with every predicate symbol. Predicate symbol p with arity n and constants $c_1, \ldots, c_n$, $p(c_1, \ldots, c_n)$ is a *fact*. A *term* is either a variable or a constant. (Note that we do not allow function symbols.) Predicate symbol p and terms $t_1, \ldots, t_n$, $p(t_1, \ldots, t_n)$ are positive *literals*, or *atoms*. If l is a positive literal, then $\neg l$ is a negative literal. We define formulas in the usual way. Every literal is a *formula*. If f_1 and f_2 are formulas, then

$f_1 \wedge f_2$, $f_1 \vee f_2$, $f_1 \Longrightarrow f_2$, and $\neg f_1$ are formulas, and for variable symbol x, $\forall x f_1$ and $\exists x f_1$ are also formulas. A *closed formula* has no free variable occurrences. Let l be a positive literal. Then l occurs positively in l, and negatively in $\neg l$. To indicate a positive l literal we write $|l|$. The set of all variables of formula f is denoted by $var(f)$. The set of free variables of formula f is denoted by $free(f)$.

Substitution σ_X is a mapping of variable set X into the set of constants and variables. The application of a substitution to a formula replaces every occurrence of a free variable $x \in X$ by its image under σ_X. For formula f, the application of σ_X to f is denoted by $f\sigma_X$. All variables not in X are left untouched. A substitution which maps variables to variables is also called a variable renaming. $\sigma_X(X)$ is the image of σ_X.

A formula is said to be in *Prenex conjunctive normal form* if it has the form $Q_1 x_1 \ldots Q_k x_k((l_{1,1} \vee \ldots \vee l_{1,m_1}) \wedge \ldots \wedge (l_{n,1} \vee \ldots \vee l_{n,m_n}))$ where the Q_i are quantifiers and each $l_{i,j}$ is a literal. $(l_{i,1} \vee \ldots \vee l_{i,m_i})$ represents a *disjunction* in the formula. A formula of the form $\forall x_1 \ldots \forall x_n\ f$ $(\exists x_1 \ldots \exists x_n\ f)$ is abbreviated to $\forall x_1, \ldots, x_n f (\exists x_1, \ldots, x_n f)$.

For generating a consistent test database, it is important to decide whether a consistency constraint is valid solely on the basis of the test data. Consider the consistency constraint $\exists x \neg is(x, person)$ and $is(john, person)$ as the database. We need to know more about the permissible values of x (the domain of *person*). To circumvent the problem, Fagin (1982) introduced the notion of domain independence. Another (syntactic) criterion is provided by the notion of *range-restrictedness* (Nicolas, 1982). Intuitively, the semantics of a database depend only on its contents, not on the underlying domain. A formula in Prenex conjunctive normal form is called range-restricted if and only if

- each $\forall$-quantified variable appears in at least one negative (restriction) literal in each disjunction where the variable occurs;
- for each $\exists$-quantified variable x occurring in a negative literal there is a disjunction in which every literal is positive and contains x (restriction literals).

In Prenex conjunctive normal form the example formula $\forall x_1 \forall x_2 p(x_1, x_2) \Longrightarrow q(x_1)$ is $\forall x_1 \forall x_2 \neg p(x_1, x_2) \vee q(x_1)$. It is range-restricted, and the restriction literal for x_1 and x_2 is $\neg p(x_1, x_2)$. By contrast, the formula $\forall x_1 \forall x_2 q(x_1) \Longrightarrow p(x_1, x_2)$ is not range-restricted because there is no negative literal in $\neg q(x_1) \vee p(x_1, x_2)$ where x_2 occurs. Replacing the second $\forall$-quantifier by a $\exists$-quantifier, we obtain $\forall x_1 \exists x_2 p(x_1, x_2) \Longrightarrow q(x_1)$ which is not range-restricted either, whereas $\forall x_1 \exists x_2 q(x_1) \Longrightarrow p(x_1, x_2)$ is restricted. Note that transformation into a logically equivalent formula in Prenex conjunctive normal form is always possible.

We denote the truth values true and false by T and F.

3.1.2 Databases. A database, DB, consists of a finite set of facts, DB^a, and a finite set of consistency constraints, DB^c, where the consistency constraints are closed, range-restricted formulas. Note that traditional constraints (often used in relational databases, e.g., functional or inclusion dependencies) can be expressed in first-order logic, and also as a closed range-restricted formula. $C(DB)$ denotes the *completion* of a database, defined as $C(DB) := DB^a \cup \{\neg a|\ a$ *is a fact,* $a \notin DB^a\}$. This notion corresponds to the closed-world assumption (Reiter, 1978). A database DB is called *consistent* if and only if $C(DB) \models c$ for all $c \in DB^c$ where $\models$ denotes classical first-order logic derivability.

Starting with a set of consistency constraints, DB^c, a *test database* is a database (DB^a, DB^c) with DB^a a set of generated facts. We differentiate between two aspects of a test database.

- *Formal properties* include the logical notions of correctness and completeness. A test database is *correct* if and only if it is consistent. In relational theory, the well-known concept of an Armstrong database (Fagin, 1982) defines a possible notion of completeness. This database precisely obeys only a given set of dependencies (and their logical consequences). Fagin and Vardi (1983) show that there always is an Armstrong database for (standard) functional and inclusion dependencies. However, in the presence of general consistency constraints there can be no Armstrong database (Fagin, 1982). Hence, our only formal requirement for test databases is correctness. (The terms completeness and correctness as used in Section 1 have little to do with the formal notions referred to above. Rather they reflect intuitive objectives of the validating designer.)

- *Pragmatic properties* include an important issue: the desired *size*, i.e., the number of facts to be generated for each predicate symbol, and the desired size of DB^a. This information may serve as a criterion to stop the generation of the test database. Another issue is the *naming* of the generated individuals. For validation purposes it is important to have meaningful identifiers. Furthermore, the user should be able to define which parts of the schema should be instantiated. This definition is called the *starting formula*. To summarize, the pragmatic aspects serve as the input parameters for the generation procedure so that only test databases with these features will be generated. We assume that all these features will be controlled by the designer or user. Since these features constitute a rather new topic, more pragmatics may become apparent with growing experience.

Our objective, then, is to arrive at a variety of test databases which are *correct*, i.e., satisfy a given set of consistency constraints and, further, respect the specified features: database size, the naming of the individuals, and the starting formula that expresses which part of the schema is to be instantiated. In general, the pragmatic features cannot be met entirely, because we deal with general consistency constraints. Pragmatic features alllow an approximate measurement in order to avoid haphazard generation and to arrive at databases that come close to the user's intentions. The generation procedure has to be *correct*, i.e., it only produces correct test databases. We do not attempt to generate all possible correct test databases, which would be prohibitively expensive,[1] but we direct our attention to an *efficient* solution in order to avoid a bottleneck in the validation process.

3.2 Scenario

As we indicated, our main objective is the validation of a semantic schema. The validation takes place within a database design environment (Lockemann et al., in press) after requirements analysis and knowledge acquisition. It can be seen as an iterative process of generating test data (for parts of the schema), subsequent testing, and eventually redefining the schema. In a first phase, test data will be generated. During the subsequent test phase, the user can formulate transactions in a semantic-level language, similar to the approaches presented in Brodie and Ridjanovic (1984) and Ngu (1989). They subject the test database to updates, modifications, and retrieval. On the one hand, these transactions will reflect the transactions in the application domain so that unexpected constraint violations may indicate an "incomplete" semantic schema (in the sense of Section 1). On the other hand, the user may define test transactions in order to simulate non-allowed actions and thus discover specification loopholes if no violation occurs ("incorrectness"). Furthermore, we are investigating the possibility of automatically generating test transactions based on the semantic schema that supports the user.

4. Deriving the Generator Formula

4.1 An Example

To illustrate our approach, we introduce an example which will serve us throughout this article. Consider this problem: we want to generate test data for a database for a car registration office. Suppose that the semantics of the application domain, i.e., the laws that govern registration, are entirely expressed in terms of consistency constraints:

1. There might be an infinite number of correct test databases if there is no upper bound specified for each predicate. On the other hand, if upper bounds are specified, the number of possible test databases still explodes. For example, consider a schema with predicates $p_1(x_1)$, $p_2(x_2)$ and $p_3(x_1, x_2)$ and no consistency constraint. Assume that n_1 facts for p_1, n_2 for p_2 are requested, and n_3 is the upper bound for p_3. Hence, we obtain $\sum_{i=0}^{n_3} \binom{n_1 * n_2}{i}$ different correct test databases. Even small numbers, $n_1 = n_2 = 5$ and $n_3 = 12$, would result in an incredible 2^{24} = 16777216 test databases.

Let DB_{owns} be a database where DB^a is empty and DB^c contains the following consistency constraints:

1. $\forall x_1 \forall x_2 owns(x_1, x_2) \Longrightarrow is(x_1, person)^2$
   The first argument of the *owns* predicate must be a person.

2. $\forall x_1 \forall x_2 owns(x_1, x_2) \Longrightarrow is(x_2, car)$
   The second argument of the *owns* predicate must be a car.

3. $\forall x_1 \exists x_2 is(x_1, car) \Longrightarrow owns(x_2, x_1)$
   Every car must have an owner.

4. $\forall x_1 \exists x_2 is(x_1, person) \Longrightarrow owns(x_1, x_2)$
   Every (registered) person owns a car.

5. $\forall x_1 \forall x_2 \forall x_3 owns(x_1, x_2) \wedge owns(x_3, x_2) \Longrightarrow x_1 = x_3$
   Every car has at most one owner.

Now suppose that we wish to generate a number of instances of car type (i.e., facts of the form $is(\_, car)$). Clearly, the first step is to introduce facts of the form *is(c,car)* where $c \in C$. Obviously, the first fact violates constraint 3 and can only be compensated for by inserting fact *owns(b,c)* where $b \in C$. In turn, constraint 1 is violated and requires the addition of fact *is(b,person)*.

Alternatively, suppose that we start with persons. Again, introduction of fact *is(b,person)* must be succeeded by *owns(b,c)* and *is(c,car)* for some constant *c*. Assume further that the database has not been empty prior to these steps. Then there is a chance that *c* had been generated earlier, with some fact *owns(d,c)*. In this case constraint 5 would be violated.

We note two problems. First, the insertion of a fact often necessitates a chain of further insertions. Second, the inserted facts may collide with already existing facts so that corrective actions must be taken. In this section, we address the first problem. Our goal is to detect this chain of insertions and correcting actions prior to the actual generation of facts. The advantage is that the detection of the needed insertions and correcting actions has to be performed once for all facts, instead of once for each generated fact. The interdependencies between the constraints, which are responsible for the chain of insertions, are represented by means of a *generator formula*.

4.2 Generation Scheme

Below we describe how the generator formula is derived. The user specifies the test data giving the corresponding atoms, e.g., $is(x_1, car)$ and $is(x_2, person)$. The

2. Instead of $p(x_1)$ with p predicate, we write $is(x_1, p)$

conjunction of these atoms is the initial generator formula. The ultimate generator formula is then derived in a stepwise fashion by applying rewrite rules from two rule sets in an alternating manner. Because this involves fairly expensive inference steps, the savings by applying them just once are considerable. The first set consists of a single rule, the extension rule. This rule forward chains possible side effects of atoms within the current generator formula on further consistency constraints. It extends the chain mentioned above whenever necessary. The second set contains several reduction rules which are used to eliminate redundancies possibly introduced by the application of the extension rule. After applying the extension rule, the reduction rules are applied exhaustively.

First, we formally describe the rewrite rules and use simple examples (some straightforward extensions of the Section 4.1 example) for illustration. These definitions are followed by a generator formula construction from the Section 4.1 example.

Let F be a set of consistency constraints in Prenex conjunctive normal form. Let gf be a(n) (initial) generator formula. Then the generation scheme, with respect to F and gf, is defined by introducing the following rules.

4.2.1 Extension Rule. Consider $gf = is(x_1, person) \wedge is(x_2, car)$ and constraint 3 of DB_{owns}: $\forall x_1 \exists x_2 is(x_1, car) \Longrightarrow owns(x_2, x_1)$. Generating cars will cause the side effect of inserting corresponding $owns$-facts. If constraint 3 is rewritten in conjunctive normal form, $f = \forall x_1 \exists x_2 \neg is(x_1, car) \vee owns(x_2, x_1)$, it can be seen that literal $is(x_2, car)$ in gf and $\neg is(x_1, car)$ in f are unifiable. Because only the premises in the constraints are responsible for the chain of insertions, only the negative literals in the corresponding conjunctive form need be considered when applying the extension rule. The extension rule extends those literals in gf which occur unifiably and negatively in a formula of F. The result of the rule application will be a new generator formula consisting of gf and the matched formula. Thus, after renaming x_2 in f as x_3, gf has been changed to

$$is(x_1, person) \wedge is(x_2, car) \wedge (\exists x_3 \neg is(x_2, car) \vee owns(x_3, x_2)).$$

This context is stated more formally as follows:
preconditions:

- $f \in F$, i.e. f is of the form $f = \forall x_1 \ldots x_k Q \bigwedge_{1 \leq i \leq n} \bigvee_{1 \leq j \leq m_i} l_{i,j}, k \geq 0$ where Q is a series of quantifiers starting with $\exists$ or is empty.

- $gf = gf' \wedge l \wedge gf''$, l is a positive literal

- l is unifiable with a negative literal $l_{i,j}$ in f. This is denoted by means of three substitutions: $l\sigma = |l_{i,j}|\tau^1\tau^2$. The substitution σ maps variables of l to constants, τ^1 is restricted to those universally quantified variables of $l_{i,j}$ which are not governed by an existentially quantified variable (i.e. to

$\{x_1, \ldots, x_k\} \cap var(l_{i,j})$), and τ^2 is restricted to the remaining variables in $var(l_{i,j})$.

action: gf is replaced by

$$gf \wedge f'\tau^1$$

where

- f' results from f by removing the universal quantifiers for those variables that were substituted by τ^1 and renaming all remaining variables of f so that they are different from all variables appearing in gf.

For our example the substitutions are as follows: $\tau^1 = [x_1 \leftarrow x_2]$, and σ, τ^2 are empty substitutions. Note that only those variables of the considered literal ($l_{i,j}$) (which are universally quantified and not governed by an existentially quantified variable) will be substituted (by τ^1). Hence, the extension rule implements a form of non-clausal resolution, but it does not remove the literals resolved upon. This is done by the first reduction rule.

4.2.2 Reduction Rules. The main purpose of the reduction rules is to eliminate redundancies in the generator formula and thus accelerate the test data generation. Another purpose is to transform the generator formula so that as many side effects as possible are detected, i.e., the extension rule can be applied as often as possible.

1. Consider the formula resulting from the application of the extension rule of the example above

   $$is(x_1, person) \wedge is(x_2, car) \wedge (\exists x_3 \neg is(x_2, car) \vee owns(x_3, x_2)).$$

   If we remember that the quantified formula in this current generator formula can also be written as $\exists x_3 is(x_2, car) \Longrightarrow owns(x_3, x_2)$ we see that the premise of the formula, $is(x_2, car)$ can be removed, since it is already satisfied.

   Hence, this rule exploits resolution within the generator formula:

   preconditions:

   - $gf = gf' \wedge l \wedge gf''$, l a positive literal
   - there exists a negative literal l' in $gf' \wedge gf''$ with $l = |l'|$

   action: gf is reduced by substituting l' by F.

   Thus, applying Rule 1 (with $gf' \equiv T$) and the observation that $F \vee l''$ is equal to l'' (with l'' some literal), we obtain $is(x_1, person) \wedge is(x_2, car) \wedge (\exists x_3 owns(x_3, x_2))$.

2. Rule 1 considers two complementary literals in gf and replaces one with F. It is also possible that gf contains two identical literals. This rule corresponds to subsumption and removes redundancy. It is formally described as follows:

preconditions:

- $gf = gf' \wedge l \wedge gf''$, l a positive literal
- there exists a positive literal l' in $gf' \wedge gf''$ with $l = l'$

action: gf is reduced by substituting l' with T.

3. If some parts of the generator formula are logically equivalent, then there should be a rule to remove this redundancy. For this purpose we introduce the following rule.

   preconditions:

   - $gf = gf' \wedge f \wedge gf'' \wedge f' \wedge gf'''$ with f, f' formulas (f and f' may be interchanged)
   - $f \Longrightarrow f'$ is a tautology

   action: gf is reduced to

   $$gf' \wedge f \wedge gf'' \wedge gf'''.$$

   This rule should be implemented by testing for frequently occurring tautologies such as formulas which only differ in the names of their variables. Undetected tautologies will merely result in a loss in performance.

Additionally, there are the usual simplifications of formulas, e.g., replacing a conjunction of literals containing F with F or a disjunction of literals containing T with T.

These rules are equivalence-preserving, i.e., they transform a given generator formula into a logically equivalent formula. Unfortunately, these rules seems incapable of discovering many of the side effects. Remember the example resulting from the application of Rule 1: $is(x_1, person) \wedge is(x_2, car) \wedge (\exists x_3 owns(x_3, x_2))$. This example began with the generator formula resulting from extending the literal $is(x_2, car)$ into the starting formula $is(x_1, person) \wedge is(x_2, car)$ with constraint 3. The literals which are candidates for further extension are now $is(x_1, person)$ and $is(x_2, car)$. The constraints whose premises are candidates for such extensions are constraint 3 (this would be a repetition) and constraint 4. The other constraints (1, 2 and 5) would never be considered, because the literal $owns(x_3, x_2)$ is bound to the existential quantifier of variable x_3.

We introduce three more non-equivalence-preserving and heuristic reduction rules. The first infers side effects caused by atoms which are bound by an existential quantifier on the consistency constraints. In this case, the right choice for substituting the existentially quantified variable is important. The other two accelerate the generation by reducing the generator formula. Consequently, the number of different test databases could be reduced. For example, some constants which should be generated get lost (Rule 5) or not enough different facts are generated (Rule 6). This has no effect on the correctness of the generation procedure (the final test database will be consistent), because in the end the whole set of consistency constraints is tested, and violations are eventually repaired. However, in order to increase the efficiency and accelerate the generation (Section 3.1.2), the non-equivalence-preserving reduction rules restrict the search space for a correct test database, that is, some correct test databases may be precluded from being generated. Hence, these rules should be applied by the user; if they are applied by the system, they should be reconfirmed by the user.

1. As the example shows, we need a reduction rule that eliminates an existential quantifier in order to arrive at an unbound literal (which can later be extended by the extension rule). The following rule does this by substituting the existentially quantified variable with a free variable in the current generator formula.

   preconditions:

   - $gf = gf' \wedge (\exists x f) \wedge gf''$, and
   - for $y \in free(gf)$ $\sigma_{\{x\}} = [x \leftarrow y]$

   action: gf is reduced to

   $$gf' \wedge f\sigma_{\{x\}} \wedge gf''.$$

   In our example generator formula, the free variables are x_1 and x_2. Selecting x_1 and applying the substitution $\sigma = [x_3 \leftarrow x_1]$ we obtain $is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2)$. Obviously, the right choice of the variable is up to the user. The rule allows a user to exploit additional knowledge.

2. It is possible to find literals in the current generator formula that are "almost" equal, i.e., one can be transformed into the other by substituting some variables by constants. Consider a slightly different example: $gf = is(x_1, sports\_car) \wedge owns(x_2, x_1) \wedge owns(john, x_1)$. Applying the substitution $[x_2 \leftarrow john]$ to $owns(x_2, x_1)$, we obtain $owns(john, x_1)$. We may now drop this more specific information if we assume that *john* will later be generated as one of the constants for x_2. Because this assumption can only be upheld by the user, the following rule which corresponds to subsumption may only be applied in a user-controlled way.

   preconditions:

   - $gf = gf' \wedge l \wedge gf'' \wedge l' \wedge gf''', l, l'$ positive literals (l and l' may be interchanged)

- a substitution σ_X with $X \subseteq var(l)$ and $\sigma_X(X) \subseteq C$ so that $l\sigma_X = l'$.

action: gf is reduced to

$$gf' \wedge l \wedge gf'' \wedge gf'''.$$

In our example, gf is reduced to $is(x_1, sports\_car) \wedge owns(x_2, x_1)$. The reader may be surprised to find constants in gf—after all, the generator formula is supposed to introduce them later. Note, however, that constants can occur in some consistency constraints, e.g. $\forall x_1 is(x_1, sports\_car) \Longrightarrow owns(john, x_1)$, and by extension they will also occur in gf.

3. In Rule 5 two literals are "almost" equal (one contains more constants than the other). This rule considers literals that differ in their variables (one can be transformed into the other by variable renaming), and also correspond to subsumption.

   preconditions:

   - $gf = gf' \wedge l \wedge gf'' \wedge l' \wedge gf'''$, l, l' positive literals (l and l' may be interchanged)
   - a variable renaming $\sigma_{var(l')}$ exists so that $l'\sigma_{var(l')} = l$.

   action: gf is reduced to

$$(gf' \wedge l \wedge gf'' \wedge gf''')\sigma_{var(l')}.$$

   Consider $gf = is(x_2, sports\_car) \wedge owns(x_1, x_2) \wedge owns(x_1, x_3) \wedge has\_color(x_3, green)$. Rule 6 can be applied with the substitution $[x_3 \leftarrow x_2]$. Consequently, all sports-cars will now have to be green. This is an aggravation for the test data generation. For example, suppose there is another consistency constraint which states that if there is a sports_car then there must be one which is red. As a consequence, the generation of consistent test data is no longer possible. Therefore, this rule is only applied after interacting with the user.

4.2.3 Rule Application. Given an initial generator formula, rules are applied in turn. Construction of the generator formula stops if equivalence-preserving rule can not be applied any longer, or if the user cannot decide to apply a non-equivalence-preserving rule.

Note that the choice of rule, if more than one meets its precondition, is non-deterministic. However, we introduce one heuristic: The intermediate formula gf

is reduced as far as possible before an extension takes place. Under this condition the rewriting stops if no further extension is possible. If a reduction rule applies to gf, it is actually applied only if it is not user-controlled or the user accepts it.

If the extension rule applies to literal l in gf and literal $l_{i,j}$ in consistency constraint f, then it should be determined whether the rule has already been applied to l and $l_{i,j}$ in f in order to avoid redundant extensions. Therefore, each extension is registered by adding the triple $(l, f, l_{i,j})$ to set Z. The rewriting is always terminated, because F contains a finite number of constraints, the initial generator formula is a conjunction of a finite number of positive literals, and the extension rule can only be applied once for a triple $(l, f, l_{i,j})$. The restriction to the triples in Z does not affect the final generator formula, because if another literal l occurs in gf to which the extension rule could be applied (with $l_{i,j}$ in f), this literal would be eliminated by Reduction Rule 2.

The resulting generator formula must be considered a heuristic that describes declaratively how to generate facts so that as few consistency violations as possible are observed during subsequent updates. If non-equivalence-preserving rules are applied, the number of models possibly is restricted, too.

4.2.4 Example Revisited. To illustrate the complete rewrite process, we apply it to Example 4.1. As noted, the consistency constraints first must be transformed into Prenex conjunctive normal form:

$$\begin{aligned} F = \{ & (1)\ \forall x_1 \forall x_2 \neg owns(x_1, x_2) \vee is(x_1, person), \\ & (2)\ \forall x_1 \forall x_2 \neg owns(x_1, x_2) \vee is(x_2, car), \\ & (3)\ \forall x_1 \exists x_2 \neg is(x_1, car) \vee owns(x_2, x_1), \\ & (4)\ \forall x_1 \exists x_2 \neg is(x_1, person) \vee owns(x_1, x_2), \\ & (5)\ \forall x_1 \forall x_2 \forall x_3 \neg owns(x_1, x_2) \vee \neg owns(x_3, x_2) \vee x_1 == x_3 \}. \end{aligned}$$

Suppose that the starting formula is $gf = is(x_1, person) \wedge is(x_2, car)$, i.e., we wish to generate persons and cars.

- First, no reduction rule can be applied. Extending $is(x_1, person)$, due to the negative literal of formula (4) (notice the difference to the example in Section 4.2.1!), yields the following result: $is(x_1, person) \wedge is(x_2, car) \wedge (\exists x_3 \neg is(x_1, person) \vee owns(x_1, x_3))$

- Applying Reduction Rules 1 and 4 with $\sigma = [x_3 \leftarrow x_2]$ we obtain $is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2)$. The choice for σ would have to be made by the user. The system could support the user by revealing formula (2) (presumably in its original form) from which he or she could conclude that x_3 is a car.

- No further reduction is possible. Extending $owns(x_1, x_2)$ with formula (1) yields $is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2) \wedge (\neg owns(x_1, x_2) \vee is(x_1, person))$

- Applying Reduction Rules 1 and 2 yields $is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2)$, i.e., the extension by formula (1) had no effect.

- The subsequent extension of $owns(x_1, x_2)$ with formula (2) similarly has no effect on the intermediate formula due to reductions according to Rules 1 and 2.

- Extending $owns(x_1, x_2)$ because of the first negative literal of (5), and applying Reduction Rule 1 we obtain $is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2) \wedge (\forall x_3 \neg owns(x_3, x_2) \vee x_1 == x_3)$

- The literal $owns(x_1, x_2)$ could once more be extended with the second negative literal of (5). Because of Reduction Rule 3, this has no effect on the intermediate result.

- Now, we extend $is(x_2, car)$ with formula (3) and obtain $is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2) \wedge (\forall x_3 \neg owns(x_3, x_2) \vee x_1 == x_3) \wedge (\exists x_4 \neg is(x_2, car) \vee owns(x_4, x_2))$

- Applying Reduction Rules 1 and 4 with $\sigma = [x_4 \leftarrow x_1]$, and then Reduction Rule 2, we arrive at the final generator formula: $is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2) \wedge (\forall x_3 \neg owns(x_3, x_2) \vee x_1 == x_3)$

As shown, rewriting compresses the constraints into a very compact form, albeit in a somewhat subjective, user-controlled manner. The final generator formula differs logically from the constraints in the sense that it is "stronger": the constraints express an existential requirement for $owns$-facts as opposed to the generator formula with implicitly universally quantified variables for $owns$. If, for example, for each variable (x_1 and x_2) two constants (e.g. $person\_1$, $person\_2$, car_1 and car_2) are generated, there are four possibilities to combine with $owns$-facts. Generating all four possibilities violates the formula following the $owns$-formula in the generator formula. To avoid such conflicts and to augment the spectrum for possible test databases, we introduce pragmatic parameters for the generation operator to guide the generation of such facts (Section 5.2.6).

5. Test Data Generation

In this section, we recall the SATCHMO approach (Bry et al, 1988) and present the enhancements that fulfill the requirements for test data generation. Finally, we present our approach for generating test data according to a generator formula.

5.1 Adapting the SATCHMO Approach

5.1.1 A Short Description of SATCHMO. Bry, et al. (1988) suggest a method for checking finite constraint satisfiability. The constraints are restricted to formulas of restricted quantification. Although the restriction resembles our notion of range-restrictedness and guarantees domain independence, neither one is a subset of the other, due to the differences in existentially quantified variables. For example, the formula $\forall x \exists y \neg p(x, y) \wedge s(y)$ is range-restricted but not of restricted quantification, whereas the formula $\forall x(\neg p(x) \vee \exists y(s(y) \wedge \neg t(y))) \equiv \forall x \exists y(\neg p(x) \vee s(y)) \wedge (\neg p(x) \vee \neg t(y))$ is of restricted quantification but not range-restricted.

This method starts with an empty database. If no consistency constraint is violated, the current database is a model and finite satisfiability is proven. If each constraint is a universally quantified formula, this is already true for a database without facts. (Due to the restricted quantification, each formula can be represented as a disjunction of which at least one component is a negative literal; in an empty database all negative facts are true). Otherwise, a repair of each violated constraint is attempted by *adding* new facts. If no repair is possible, the procedure backtracks to the last choice point. The recursive algorithm stops as soon as all constraints are satisfied, with the current database as the resulting model.

5.1.2 Extending SATCHMO for Test Data Generation. In order to adapt SATCHMO to our purposes, we extended it with additional features. This extended version is called exSATCHMO. One feature is the ability to control the size of the model: According to the starting formula facts are inserted, e.g. when starting with $is(x_1, person)$, $is(person\_1, person)$ is added. If a model is found, the facts are counted. If the quantity is smaller than the quantity desired by the user, new facts are added automatically, and the algorithm is started again.

Another feature allows the user to specify a pattern which generates meaningful identifiers, e.g., $person\_i$ for persons.

The most important extension includes a generator formula. The generator formula of our running example is $is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2)$ (for reasons apparent in Sections 5.2.4 and 5.2.5, we omit the last quantified formula). exSATCHMO proceeds as follows: If two persons and two cars are requested, $is(person\_1, person)$, $is(car\_1, car)$ and $owns(person\_1, car\_1)$ are added in a first step. Hence, variables x_1 and x_2 are related to the generated instances $person\_1$ and car_1. Keeping track of such relationships will be explained in detail in the next subsection. Because consistency remains untouched, a model has been found. However, the model is too small for our purposes (after all, we requested 2 persons and 2 cars). Facts are inserted once again, i.e. $is(person\_2, person)$, $is(car\_2, car)$ and $owns(person\_2, car\_2)$. These insertions also preserve consistency. Furthermore, now the required quantities have been generated. Thus, this model constitutes the result of the generation process.

Suppose two persons and only one car should be generated. After the insertion of $is(person\_1, person)$, $is(car\_1, car)$ and $owns(person\_1, car\_1)$, and after it has been determined that consistency is satisfied, $is(person\_2, person)$ is added to fulfill the required quantity of persons. Because $person\_2$ does not own a car constraint 4 is violated. SATCHMO tries to add new facts; an instantiation must be searched for $\exists x_2 is(person\_2, person) \Longrightarrow owns(person\_2, x_2)$. Instantiations that are deducible for the restriction literals are computed. For cases where no candidates are derivable, an additional heuristic has been incorporated in SATCHMO to find more restriction literals (Moerkotte and Lockemann, 1991). This way we arrive at $owns(person\_2, car\_1)$. But now, constraint 5 is violated and cannot be repaired by adding another fact. Hence, the previous computation is backtracked and, instead of selecting $owns(person\_2, car\_1)$, a new constant, $owns(person\_2, car\_2)$ with car_2, is inserted. Finally, to cure violated constraint 2, $is(car\_2, car)$ will be added.

5.2 An Approach Based on Operationalization of the Generator Formula

As we indicated, the SATCHMO approach comes closest to our own. It incrementally generates facts, tests for consistency, and repairs inconsistencies by adding new facts and by chronological backtracking. Although it has been extended with additional features to fulfill the requirements for test data generation, we propose an approach centered around the generator formula and, hence, more directly related to the requirements of test data generation.

5.2.1 General Approach. The generator formula is the result of the first stage. The generator formula, a descriptive abstraction of the involved consistency constraints and of possible conflicts within the generated test data, is translated into a sequence of operators. For this purpose, we introduce two powerful operators: the generation operator and the test-and-repair operator. The former generates new test identifiers according to a user-specified pattern, the latter checks whether the generator formulas (which are affected by the test data generated so far) remain valid for the current test database. In case of failure, an integrated repair mechanism adds and removes test data so that the invalidity is resolved.

First we demonstrate our approach to operationalizing the generator formula with the example from the previous section. $Persons$ and $cars$ should be generated by giving the starting formula $is(x_1, person) \wedge is(x_2, car)$. The corresponding generator formula is $is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2) \wedge (\forall x_3 \neg owns(x_3, x_2) \vee x_1 == x_3)$. To generate two persons and one car, there are several problems to attack: Generate the specified amount of data, represent the generated information, and meet the consistency constraints. We will now discuss each problem in more detail and illustrate specific solutions. Subsequent sections describe the general solution.

Because atoms are the smallest formulas in a generator formula, one would expect the generation operator to identify them and then create instances for them. For example, it could generate two facts, $is(person\_1, person)$ and $is(person\_2, person)$, for the atom $is(x_1, person)$. Unfortunately, this approach obscures the relationship between the variable x_1 and the generated instances $person\_1$ and $person\_2$. However, this relationship is needed because of the occurrence of variable x_1 in other atoms of the generator formula where it must be instantiated by the generated individuals. Hence, some bookkeeping with a suitable representation must accompany the test data generation. For this purpose, a triple called *test data structure* is introduced. The first component contains the set of atoms for which test data has been generated, in our case initially $\{is(x_1, person)\}$. The second component contains the generated individuals in form of substitutions for the free variables of the atom, so far $\{[x_1 \leftarrow person\_1], [x_1 \leftarrow person\_2]\}$. The third component is redundant but simplifies the formulation of the subsequent steps. It contains the set of variables that so far have been bound by the substitutions, $\{x_1\}$ in our example. This test data structure is then passed to the next operator within the generated operator sequence. Here, one *car* is to be generated next. Thus, $is(x_2, car)$, $[x_2 \leftarrow car\_1]$, and x_2 are added to the three components of the test data structure, respectively.

Consider the case of an atom whose variables are bound by previously generated test data, i.e., whose variables appear already in the third component of the test data structure. This is true, e.g., for the next atom to be treated, $owns(x_1, x_2)$. The generation operator proceeds as follows: it tries to combine the respective substitutions found within the test data structure. Here, the complete set of combinations is $\{[x_1 \leftarrow person\_1, x_2 \leftarrow car\_1], [x_1 \leftarrow person\_2, x_2 \leftarrow car\_1]\}$. The test data structure is augmented by $owns(x_1, x_2)$ in the first component, and the two combinations in the second component. The third component remains unchanged because no new variable is involved.

Treating the last conjunct $\forall x_3 \neg owns(x_3, x_2) \vee x_1 == x_3$ of our generator formula, we observe another problem. This formula obviously is false for the given input because both $person\_1$, and $person\_2$ own car car_1. Further, this formula cannot be made true by generating new persons or cars using the generation operator. We must invoke another mechanism to correct the test database created so far. The corresponding test-and-repair operator is applied whenever the generator formula contains a conjunct (here: an element of the conjunction that represents the generator formula) which is not an atom, or is an atom with $==$ as its predicate.[3]

Notice that we proceeded from left to right, i.e., we interpreted the generator formula as an instruction, not a logical formula, to generate, test, and repair facts. This suggests that we devise an algorithm that translates a given generator formula into an operator sequence (Section 5.2.5). First, we formally introduce the test data structure, the generation operator, and the test-and-repair operator.

5.2.2 The Test Data Structure. The test data structure consists of three components: a set of atoms for which test data has been generated; a set of substitutions for the variables occurring in the atoms of the first component; and a set of those variables bound by the substitutions in the second component of the test data structure. The

3. This limitation is due to the fact that the interpretation of equality is fixed and cannot be manipulated by generating facts of the form $a == b$.

third component could, of course, be derived from the substitutions but is kept redundantly for convenience. More precisely, we define:

Definition 5.1 A test data structure is a triple (A, S, V) where

- A is a set of atoms,
- S is a set of substitutions with $\forall a \in A \exists \sigma_X \in S : X \supseteq var(a)$ and $\sigma_X(X) \subseteq C$, i.e., S contains for each variable of atoms in A at least one substitution that maps the variable to a constant.
- $V = \bigcup_{a \in A} var(a)$ is a set of variables, by definition all those variables for which at least one substitution has so far been found.

Initially $A = S = V = \emptyset$.

Note that it is trivial to construct the test database from the test data structure. We obtain it by applying all the substitutions given in the second component to the atoms within the first component. We take this procedure for granted from now on.

5.2.3 Generation Operator. The generation operator has two input parameters, the (present) test data structure and an atom for which test data is to be generated, and one output parameter, the appropriately modified test data structure. The necessary changes to the structure vary, depending on the free variables of the argument atom and the variables already bound by earlier substitutions. We distinguish four cases:

1. In case the input atom is a fact, i.e., does not contain any variable, we add just this fact to the set of already treated atoms, i.e., the first component of the test data structure. The other two components remain unchanged.
2. In case all the variables of the input atom are already bound by substitutions of the input test data structure, the first two components are modified to include the atom and the combinations of substitutions, respectively. This procedure has been indicated above for $owns(x_1, x_2)$.
3. If no variable of the input atom is bound, individuals and substitutions using these individuals have to be generated (as for $is(x_1, person)$).
4. Finally, some but not all of the variables of the input atom may already be bound. This case is treated as a combination of cases 2 and 3.

Definition 5.2 ($gen - op$). The operator $gen - op$ generates new atoms and substitutions. It is defined by the following characteristics:
input parameters: atom a,
test data structure $T_{in} = (A_{in}, S_{in}, V_{in})$

output parameters: test data structure $T_{out} = (A_{out}, S_{out}, V_{out})$
semantics: $A_{out} := A_{in} \cup \{a\}$

Four cases are distinguished:

1. Atom a is a fact: $var(a) = \emptyset$:
   $S_{out} := S_{in}$,
   $V_{out} := V_{in}$.
2. For all variables of a at least one substitution already exists:
   $var(a) \neq \emptyset$ and $var(a) \subseteq V_{in}$:
   combine the substitutions of S_{in} to new ones:
   consider those substitutions $\sigma_{X_i} \in S_{in}$ with $var(a) \subseteq X_1 \cup \ldots \cup X_n$ and $X_1 \cup \ldots \cup X_n$ minimal
   $S_{out} := \{\sigma_{X_1} \circ \ldots \circ \sigma_{X_n}\} \cup \{\sigma_X | \sigma_X \in S_{in}, X \not\subseteq X_1 \cup \ldots \cup X_n\}$
   $V_{out} := V_{in}$.
3. No substitution exists for any variable of a:
   $var(a) = \{x_1, \ldots, x_k\}$ and $var(a) \cap V_{in} = \emptyset$:
   for each $x_i \in var(a)$ generate n_i constants c_{ij},
   generate new substitutions,
   $S_{out} := \{\sigma_{\{x_1\}} \circ \ldots \circ \sigma_{\{x_k\}} | \sigma_{\{x_i\}} \in \{[x_i \leftarrow c_{ij}] | 1 \leq j \leq n_i\}\} \cup S_{in}$,
   $V_{out} := V_{in} \cup var(a)$.
4. Only for a part of the variables of a substitutions already exist: $var(a) \neq \emptyset$ and $var(a) \cap V_{in} = X$ where $X \neq \emptyset$ and $X \neq var(a)$:
   for X proceed according to case 2 and produce S_1,
   for $var(a) \setminus X$ proceed according to case 3 and produce S_2,
   $S_{out} := \{\sigma_{X_1} \circ \sigma_{X_2} | \sigma_{X_1} \in S_1, \sigma_{X_2} \in S_2\}$,
   $V_{out} := V_{in} \cup var(a)$.

To allow for more flexibility and user influence on the generation process, additional parameterization of the generation operator is provided to direct the generation process and generate test databases with different semantics (Section 5.2.6).

5.2.4 Test-and-Repair Operator. As indicated in Section 5.2.1, this operator is invoked as part of processing the generator formula. The underlying repair mechanism (X4), which deduces repairs for violated constraints, originates from a system described in Moerkotte and Lockemann (1991). We adapted it to the purpose of test data generation. We sketch the general principles of the approach, the changes we made in order to fit it to our problem, and its relationship to SATCHMO. We also examine at what point the operator is to be invoked.

Similar to SATCHMO, X4 attempts to alleviate all violated consistency constraints simultaneously. Contrary to SATCHMO, X4 does not exclusively rely on the addition of new facts. Instead, it extracts *potential causes* for a violated constraint from a trace created during the consistency check. A potential cause is a minimal set of literals where a positive literal represents an atom which should be deleted from the current database and a negative literal $\neg a$ stands for an atom a which should be added to the database in order to alleviate the consistency violation. Consequently, it is possible for X4 to delete facts explicitly, whereas in SATCHMO facts can be deleted implicitly only by backtracking.

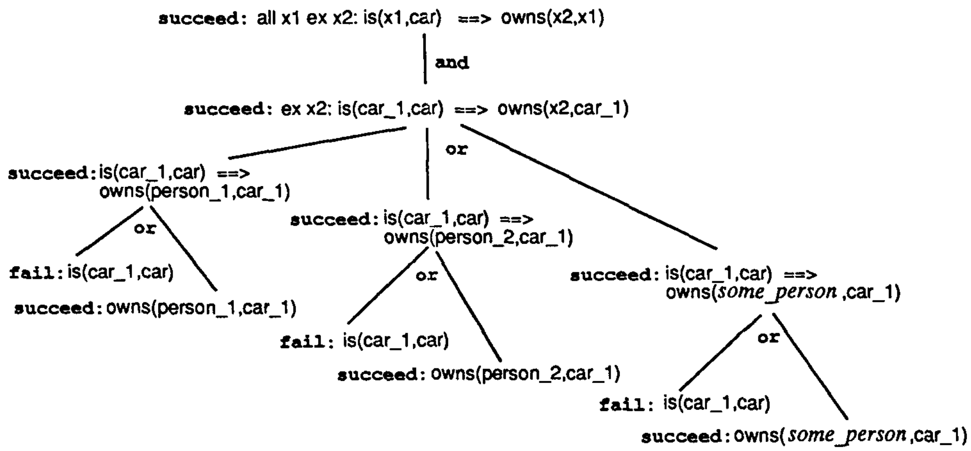
As in SATCHMO the algorithm is iterative in nature because a potential cause may not be an ultimate repair if several consistency constraints must be considered simultaneously. (The "repair" of one consistency constraint may violate another.) Therefore, we distinguish between a potential cause (concerned with a single consistency constraint), and a *definite cause* (concerned with the entire set of constraints). For each potential cause an attempt is made to derive iteratively a definite cause that also should be minimal (Moerkotte and Lockemann, 1991). The potential cause is "executed," i.e., for a positive literal a we compute $del(a)$ (a is deleted from the database), and for a negative literal $\neg a$ we have $add(a)$. Consistency is checked for the resulting database. If a violation is discovered, the set of its potential causes is derived. Each of those new potential causes is combined with the starting cause and then is executed again. If no constraint is violated, we arrive at a definite cause, otherwise new potential causes have to be computed and so on. A *repair* represents the executed version of a definite cause and consists of a minimal number of changes. Note that the generation of repairs is closely related to the problems of view and intentional update investigated (Cosmadakis and Papadmitriou, 1984; Manchanda and Warren, 1986; Tomasic, 1988; Rossi and Naqvi, 1989; Bry, 1990; Guessoum and Lloyd, 1990, 1991).

In light of the test data generation, minimality of repairs is superfluous. Furthermore, it is not necessary to compute all possible repairs because only one repair is needed to cure a constraint violation. What we need are criteria to select causes and repairs in order to govern the appearance of the test database and arrive at a certain variety. The number of potential causes increases as the quantity of test data grows and, thus, the number of all possible combinations increases enormously even for small quantities of test data. As a consequence, the computation of all potential causes may lead to memory problems and bad performance.

To illustrate the generation of potential repairs, consider our running example at the point where two persons and one car have been generated. The insertion of the appropriate facts leads to a violation of constraints 3 and 4. The corresponding traces are depicted in Figures 1 and 2.

The satisfaction of the failed constraint can be reached if all of its instantiations for x_1 are successfully derivable (Figure 1). (Note that the *all*-requirement leads to the *and*-connector.) Here, car_1 is the only instantiation deducible from the database. On the other hand, the test of the existentially quantified subformula will succeed if at least one instantiation evaluates to true (*or*-connector). In principle, one constant is as good as another. Because it does not make sense to consider all constants of the database, those instantiations which are deducible from all the restriction literals are chosen (the same principle found in exSATCHMO). If no instantiations are found, a heuristic to derive new restriction literals is applied. This is done by forward chaining within the constraints (Moerkotte and Lockemann,

Figure 1. Derivation tree for violated constraint 3


succeed: all x1 ex x2: is(x1,car) ==> owns(x2,x1)
and
succeed: ex x2: is(car_1,car) ==> owns(x2,car_1)
or
succeed: is(car_1,car) ==> owns(person_1,car_1)
or
fail: is(car_1,car)
succeed: owns(person_1,car_1)
succeed: is(car_1,car) ==> owns(person_2,car_1)
or
fail: is(car_1,car)
succeed: owns(person_2,car_1)
succeed: is(car_1,car) ==> owns(*some_person*,car_1)
or
fail: is(car_1,car)
succeed: owns(*some_person*,car_1)

1991). For our example, we arrive at $person\_1$, $person\_2$, and a new constant $some\_person$. Each of the instantiated subformulas is satisfied by the failure of its left-hand side or by the success of its right-hand side. Taking these subformulas we arrive at ground atoms, and the trace terminates at this point.

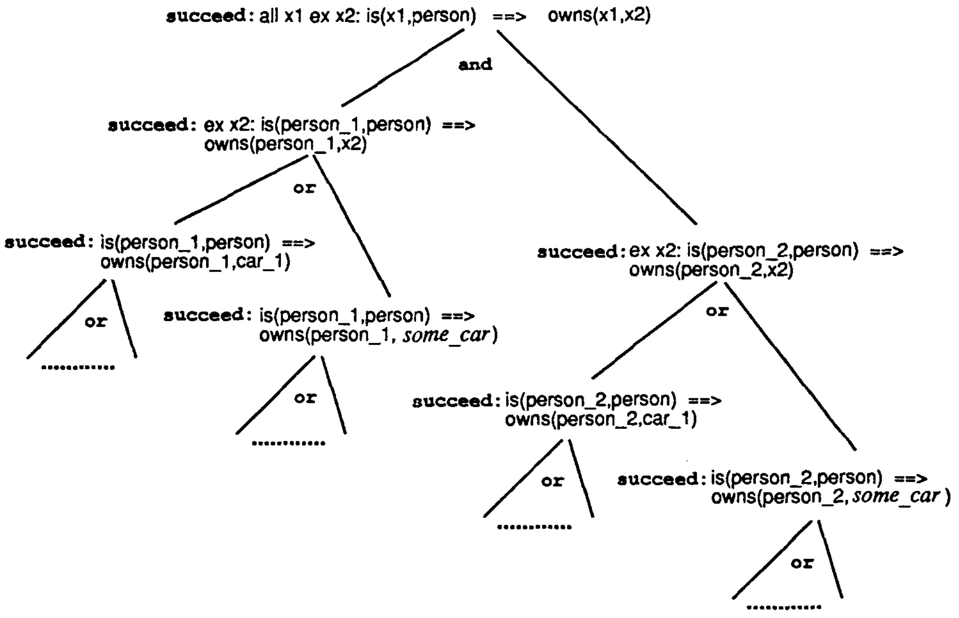
Starting at the root of the trace, we collect information as follows: we recursively collect the subnodes of each node with the connector given, and introduce negation if the goal is "succeed." Recursion terminates with a leaf. For the trace in Figure 1, this results in

$$\begin{array}{llll}
\text{(and} & \text{(or} & \text{(or} & is(car\_1, car) \\
 & & & \neg owns(person\_1, car\_1)) \\
 & & \text{(or} & is(car\_1, car) \\
 & & & \neg owns(person\_2, car\_1)) \\
 & & \text{(or} & is(car\_1, car) \\
 & & & \neg owns(some\_person, car\_1))))
\end{array}$$

This expression may be simplified to

$$\begin{array}{ll}
\text{(or} & is(car\_1, car) \\
 & \neg owns(person\_1, car\_1) \\
 & \neg owns(person\_2, car\_1) \\
 & \neg owns(some\_person, car\_1))
\end{array}$$

Figure 2. Derivation tree for violated constraint 4


succeed: all x1 ex x2: is(x1,person) ==> owns(x1,x2)
and
succeed: ex x2: is(person_1,person) ==> owns(person_1,x2)
or
succeed: is(person_1,person) ==> owns(person_1,car_1)
or
succeed: is(person_1,person) ==> owns(person_1, some_car)
or
succeed: ex x2: is(person_2,person) ==> owns(person_2,x2)
or
succeed: is(person_2,person) ==> owns(person_2,car_1)
or
succeed: is(person_2,person) ==> owns(person_2, some_car)
or

Ground atoms have been omitted, but may easily be reconstructed. Compare with Figure 1.

Each of its constituents corresponds to one potential cause (i.e., we arrive at four potential causes for constraint 3). For constraint 4, Figure 2 results in the expression

$$\begin{array}{lll}
(\text{and} & (\text{or} & is(person\_1, person) \\
 & & \neg owns(person\_1, car\_1) \\
 & & \neg owns(person\_1, some\_car)) \\
 & (\text{or} & is(person\_2, person) \\
 & & \neg owns(person\_2, car\_1) \\
 & & \neg owns(person\_2, some\_car)))
\end{array}$$

By transforming this expression into disjunctive normal form, we obtain $3*3 = 9$ potential causes. Because both constraints 3 and 4 must be satisfied, the overall consistency can be checked by joining them conjunctively. After simplification, we arrive at the following expression for both constraints

$$\begin{array}{lll}
(\text{and} & (\text{or} & is(car\_1, car) \\
 & & \neg owns(person\_1, car\_1)
\end{array}$$

$\neg owns(person\_2, car\_1)$
$\neg owns(some\_person, car\_1))$
(or $is(person\_1, person)$
$\neg owns(person\_1, car\_1)$
$\neg owns(person\_1, some\_car))$
(or $is(person\_2, person)$
$\neg owns(person\_2, car\_1)$
$\neg owns(person\_2, some\_car)))$

Hence we derive $4 * 3 * 3 = 36$ potential causes. This indicates that the number of causes can grow tremendously even for very few facts. One can derive roughly 1600 potential causes for the case of 3 persons and 2 cars. In the worst case, the number of repairs is exponential in DB^a size. To reduce this number, we perform some other simplifications of the expression resulting from the trace, e.g.,

(and (or l_1
l_2)
(or l_2
l_1))

with l_1, l_2 literals will be simplified to

(or l_1
l_2).

The original repair mechanism goes one step further and computes the *minimal* disjunctive normal form in order to gain minimality of repairs.

The example confirms our initial suspicion that selection criteria are urgently needed. These should be locally applicable, i.e., not requiring the knowledge—and hence, derivation—of all causes. We leave it up to the user to choose among one of three selection strategies:

- del-strategy:
  Positive literals are preferred.

- add-strategy:
  Negative literals are preferred. This strategy corresponds to the search order realized in SATCHMO.

- interactive-strategy:
  The user is interactively asked for each or-expression deducible from the derivation tree to select one of its constituents. That means he or she is able to directly select causes or even parts of a cause.

For our example (Figures 1 and 2) and its corresponding expression (above), the del-strategy yields the cause $\{is(car\_1, car), is(person\_1, person), is(person\_2, person)\}$, which makes little sense because it undoes all the previous updates. By contrast, the add-strategy yields $\{\neg owns(person\_1, car\_1), \neg owns(person\_2, car\_1)\}$, which better matches the intuitively desirable additions. However, this computed potential cause violates constraint 5 which can only be cured by deleting $owns(person\_1, car\_1)$ or $owns(person\_2, car\_1)$, respectively. Combining one of those literals with the starting cause results in a cause that contains a fact and its negation. Such inconsistent causes cannot be amplified to definite causes. Consequently, the computation of another potential cause is attempted according to the add-strategy. This yields $\{\neg owns(person\_1, car\_1), \neg owns(person\_2, some\_car)\}$, which finally leads to the definite cause $\{\neg owns(person\_1, car\_1), \neg owns(person\_2, some\_car), \neg is(some\_car, car)\}$ (the same strategy realized in SATCHMO, Section 5.1.2). However, since we simplify the expressions from the trace, the resulting test databases need not be identical.

In summary, if we are willing to sacrifice full automation of the generation process and permit the user to influence the sequence, a time-consuming search can be avoided, and the appearance of the test database can be effected. exSATCHMO permits a direct influence on the appearance as well, however this can only be done by rejecting the model found and initiating backtracking.

It has been shown that the original repair mechanism works correctly and computes all minimal repairs (Moerkotte and Lockemann, 1991). Dropping minimality and computing only *one* repair still preserves correctness. This allows us to defer the test-and-repair operation to the end of the sequence of $gen - op$ operations that corresponds to a generator formula, and then to apply it to the total set of consistency constraints.

Before providing a formal specification of the test-and-repair operator, we continue the example given at the beginning of this section. Remember that we generated two persons and one car, and both persons owned this one car. This situation was ruled out by the last partial formula $\forall x_3 \neg owns(x_3, x_2) \vee x_1 == x_3$ of the generator formula. A solution is to remove one of the substitutions, e.g., the one representing the fact that $person\_2$ owns the car car_1. This is exactly what the repair operator does in this case. Quantified formulas within the generator formula also have to be subjected to the test-and-repair operator and not the $gen - op$ operator.

The test-and-repair operator has two input parameters, a set of formulas, and the test data structure. If the formulas hold within the input test database, the test data structure remains unchanged. Otherwise, it is repaired so that the formulas hold within the output database. The computed repairs may result in additions or deletions of substitutions.

Definition 5.3. The operator $t\&r - op$ checks whether a set of formulas holds for the set of generated test data and repairs if necessary, the set of test data.

```
input parameters:   set of formulas $F$,
                    test data structure $T_{in} = (A_{in}, S_{in}, V_{in})$
output parameters:  test data structure $T_{out} = (A_{out}, S_{out}, V_{out})$
semantics:          check if $\{a\sigma_X | a \in A_{in}, \sigma_X \in S_{in}, var(a) \subseteq X\} \models$
                        $\{f\sigma_X | f \in F, \sigma_X \in S_{in}, free(f) \subseteq X\}$
                    then $T_{out} := T_{in}$,
                    else add or remove atoms and/or substitutions of $T_{in}$,
                        i.e., $T_{in}$ is changed to $T_{repaired}$ so that
                        $\{a\sigma_X | a \in A_{repaired}, \sigma_X \in S_{repaired}, var(a) \subseteq X\} \models$
                        $\{f\sigma_X | f \in F, \sigma_X \in S_{repaired}, free(f) \subseteq X\}$,
                        $V_{repaired} := \bigcup_{a \in A_{repaired}} var(a)$,
                        $T_{out} := T_{repaired}$.
```

5.2.5 Generation Procedure. Here we present the process for translating a generator formula into a sequence $op_1, \ldots, op_n$ of operator calls. As a heuristic, we order the operators in the generator formula so that conjuncts with the least number of still unbound variables are preferred. If there are several atoms with the same number of still unbound variables, the one with the least total number of variables is used. This leaves conjuncts which are not atoms, or are atoms with $==$ as predicates and, hence, are unsuitable as input parameters for the generation operator. Such conjuncts are passed to the test-and-repair operator and processed as soon as all their free variables are bound.

Algorithm 5.4: Translation into operator sequence. Let gf be the generator formula in the form of a conjunction of formulas, and DB^c the set of constraints to be satisfied.

```
start operator sequence $OPS := ()$,
    test data structure $(A, S, V) := (\emptyset, \emptyset, \emptyset)$
    set of variables already bound $V_{bound} := \emptyset$

loop on $gf$

    • select the next atom $a$ in $gf$ so that the number of variables of $a$ which do
      not occur in $V_{bound}$ is minimal and, if there are choices, the total number
      of variables of $a$ is also minimal,
      $OPS :=$append$(OPS, gen - op(a))$,
      $V_{bound} := V_{bound} \cup var(a)$,

    • select all $f$ formulas which are not atoms, or are atoms with $==$ as predicates
      in $gf$ so that $free(f) \subseteq V_{bound}$, and there are no $a$ atoms with $gen - op(a)$
    $\notin OPS$ and $var(a) \subseteq V_{bound}$,
    $OPS$ :=append($OPS, t\&r - op(F)$) where $F$ is the formula set for
    which the condition holds.

endloop
append $t\&r - op(DB^c)$ to OPS
```

As soon as all free variables of f formulas in the generator formula are found in V_{bound} (i.e., constants have been generated for them), and there are no a atoms left with variables in V_{bound}, we append $t\&r - op(f)$ to OPS. That means that formulas are tested as early as possible to avoid generating too many eventually false test data.

At the end of the sequence the whole set of original consistency constraints is appended. This is necessary because the repair of a formula could violate the validity of one or more formulas processed earlier. Because the repair mechanism is complete and correct, the correctness of our approach is guaranteed. Further, termination of the entire test data generation procedure is guaranteed. Clearly, $gen - op$ terminates. Hence, the generation procedure terminates if $t\&r - op$ terminates (Moerkotte and Schmitt, submitted).

To illustrate the translation of a generator formula into an operator sequence, we apply it to Example 4.1 of DB_{owns} and its generator formula:

$$is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2) \\ \wedge(\forall x_3 \neg owns(x_3, x_2) \vee x_1 == x_3).$$

The first element of the sequence is $gen - op(is(x_1, person))(gen - op(is(x_2, car))$ is also a possible choice). Next, $is(x_2, car)$ is chosen, because no f formula exists with $free(f) \subseteq V_{bound} = \{x_1\}$, among the atoms (it contains fewer variables than $owns(x_1, x_2)$). Thus we have $OPS = (gen - op(is(x_1, person)), gen - op(is(x_2, car)))$. In the third step, atom $owns(x_1, x_2)$ is the natural candidate because its variables occur in $V_{bound} = \{x_1, x_2\}$. In the fourth step, the last formula is considered because it includes a still unbound variable (x_3). However, it is a quantified formula, not an atom, so no $gen - op$ operation is produced. Instead, it is appended together with all the consistency constraints of DB_{owns} to the operator sequence. Hence, we obtain the following operator sequence:

$$(gen - op(is(x_1, person)), gen - op(is(x_2, car)), gen - op(owns(x_1, x_2)), \\ t\&r - op(\{\forall x_3 \neg owns(x_3, x_2) \vee x_1 == x_3\}), t\&r - op(DB^c_{owns})).$$

5.2.6 Pragmatic Parameters of the Generation Operator. Here we briefly discuss additional parameters that could be supplied to the generation operator. One parameter (n_i, Definition 5.2, case 3) has to do with the number of constants to be generated by the operator. This allows the user to control the size range of the generated database. The size parameter is added to the generation operators in the final operator sequence as follows. For each variable in $var(a) \setminus V_{in}$ of a generation operator, the number of constants to be generated with atom a has to be specified by the user. The operator then generates the set(s) of constants of the specified size. This cannot be guaranteed to be the number of constants in the final generated test database, because the t&r-operator may introduce new constants or remove others. Thus, the size parameter is only an approximate measure for the number of constants in the final test database.

The second parameter concerns the combinations to be generated by the generation operator. Remember the operator sequence generated for our example DB_{owns}. Here, $gen-op(owns(x_1, x_2))$ is executed after constants for x_1 (persons) and x_2 (cars) have been generated. If, for example, for each variable two constants ($person\_1$, $person\_2$, car_1 and car_2) were generated, four possibilities exist to combine with $owns$-facts. If the generation operator has to generate facts for an atom $p(x_1, \ldots, x_n)$, and for each x_i there are n_i possible instantiations, the number of combinations can become unmanageably large. On the other hand, the predicate in the (here $owns$) atom reflects certain real-world situations. Consequently, we allow the user to roughly specify the semantics of the relation by providing an approximate mathematical specification. For example, the user may specify that each person should own a car, or vice versa. For a binary predicate, we allow the user to specify whether the relation should be injective, surjective, etc. or give functional dependencies. So far, we provide a set of 16 different relation types among which the user may select.

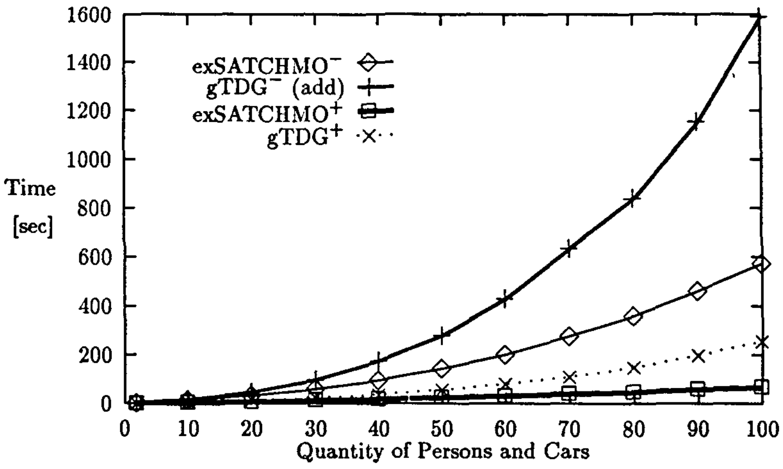
The advantage of this additional parameter is two-fold: First, the user can direct the generation process and arrive at test databases with different semantics. Second, it is possible to exploit additional knowledge the user might have (in order to avoid the generation of test data that otherwise would contradict some given consistency constraint) and thus reduce lengthy work of the $t\&r$-operations. We can even go one step further and automatically check whether the specification of the relation is a consequence of the database constraints (by means of a theorem prover), and hence, will not lead to a constraint violation.

6. Benchmark Results

In previous sections, we introduced exSATCHMO and our own approach (referred to in the following as the generic test data generator, gTDG) together with the possibility for including the generator formula. Although we speculated on the advantages and disadvantages of both, more solid measurements should be used to compare these approaches.

We differentiate between a +version and a –version for each method, resulting in exSATCHMO$^+$, exSATCHMO$^-$ and gTDG$^+$, gTDG$^-$ (+ indicates the exploitation of a generator formula and – represents the fact that only the user's starting formula is given). Furthermore, we indicate which strategy in gTDG was selected by noting it in brackets. (If the strategy does not play a role because repairs are unnecessary,

Figure 3. Cost of generating persons and cars



(Assuming equal quantities of persons and cars are specified)

we omit this information.)

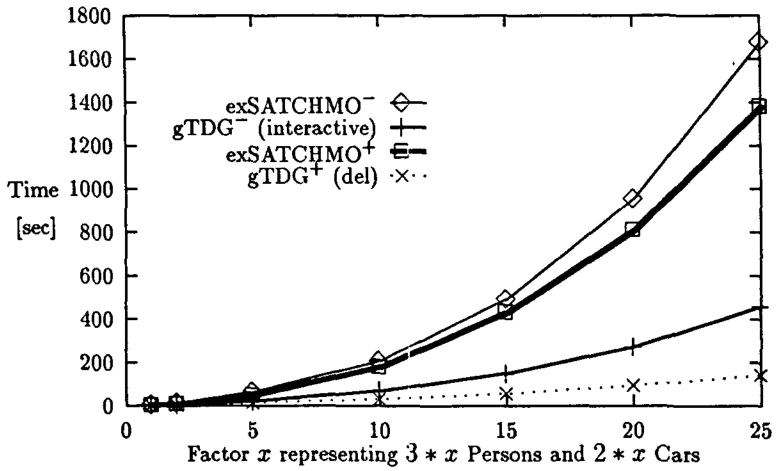
The benchmarks were run on a Sun4 Workstation under Quintus Prolog 3.0. Both approaches use the same underlying database system and consistency checker.

6.1 Benchmarking the Running Example.

Here the basis for the benchmark is our example as introduced in Section 4.1. The desired size of the test database, i.e., the number n of persons and cars to be generated, was varied. We measured the CPU times to generate n persons and n cars. The times for the user inputs were not considered. The time for deriving the generator formula could be neglected. The generator formula from Section 4 was applied: $is(x_1, person) \wedge is(x_2, car) \wedge owns(x_1, x_2) \wedge (\forall x_3 \neg owns(x_3, x_2) \vee x_1 = x_3)$ (Figure 3).

The benchmark illustrates the performance gain of the +versions due to the preceding analysis step: Because of the generator formula, the generation of *owns*-facts is explicitly enforced, whereas without this formula the generation of a car or a person results in a violation of constraints 3 or 4, which must be repaired. With the generator formula, no repairs were necessary because the generation operator combined the constants for the *owns*-facts so that the constraints 3, 4 and 5 were not violated (Section 5.2.6). The performance gain also increases with increasing quantities of test data: In the case of 100 persons and 100 cars exSATCHMO$^+$ is better than exSATCHMO$^-$ by a factor of 8; gTDG$^+$ is better than gTDG$^-$ by a factor of 6. The database generated by gTDG$^-$ depends on the strategy selected by

Figure 4. Cost of generating persons and cars



(Assuming different quantities of persons and cars are specified)

the user. For this benchmark, we selected the add-strategy. Hence, gTDG$^-$ behaves like exSATCHMO$^-$. The resulting databases are the same for both –versions and +versions.

For both versions, exSATCHMO is superior to gTDG. This can't be due to the different repair mechanisms, because no repairs were necessary in the +version. The poorer performance of gTDG is due to the consistency check. It gets worse the larger the quantities of test data that are generated at once. gTDG generates 100 persons and 100 cars and then tests for consistency, whereas exSATCHMO incrementally generates data, checks consistency, and repairs violations. One hundred small consistency checks testing two objects are less costly than a large check testing two hundred objects.

If the quantities of persons and cars are no longer identical and more persons than cars should be generated, the +versions necessitate repairs as well (due to constraint 5). Figure 4 visualizes the cost of generating persons and cars at 3/2 ratio. The values of the horizontal axis represent the factors, e.g. 25 means 75 persons and 50 cars should be generated.

When specifying different quantities for persons and cars the performance decreases for both +versions as compared to the symmetric situation. However, now gTDG shows a much better performance than exSATCHMO (nine times better when generating 75 persons and 50 cars ($x = 25$)).

Furthermore, the performance gain of the exSATCHMO generator formula shrank to 1.2 (for $x = 25$), due to its underlying repair mechanism. For example, when generating 3 persons and 2 cars, exSATCHMO$^+$ generates $is(person\_1,$

$person)$, $is(car\_1, car)$, $owns(person\_1, car\_1)$ then $is(person\_2, person)$, $is(car\_2, car)$, $owns(person\_2, car\_2)$ and finally $is(person\_3, person)$. Constraint 4 is violated because $person\_3$ does not own a car but should. When repair $owns(person\_3, car\_1)$ is added, constraint 5 is violated. The alternative repair is $owns(person\_3, car\_2)$ and leads to the same violation. Finally, $owns(person\_3, car\_3)$ with car_3 as new constant is selected. Due to constraint 2, $is(car\_3, car)$ is added and consistency is regained. This example illustrates that, with increasing numbers of persons and cars, the number of hopeless attempts (adding $owns(person\_n, car\_i)$ with $1 \leq i < n$) grows steadily.

Now consider 3 persons and 2 cars using gTDG. gTDG$^+$ first generates the same database as exSATCHMO$^+$ (3 persons, 2 cars and $owns(person\_1, car\_1)$ and $owns(person\_2, car\_2)$). When testing the consistency, constraint 4 is violated. The repair mechanism determines four potential causes: $add(owns, person\_3, car\_1)$,$add(owns, person\_3, car\_2)$, $add(owns, person\_3, car\_3)$, $del(is, person\_3, person)$ (this is the order in which exSATCHMO$^+$ searches for a successful repair). For this benchmark we applied the del-strategy. In contrast to SATCHMO the action $del(is, person\_3, person)$ constitutes an immediately successful repair (impossible in SATCHMO, because it does not allow for explicit deletes).

On the other hand, selecting the del-strategy for gTDG$^-$ leads to an empty database. Selecting the add-strategy yields a result similar to exSATCHMO$^-$. Hence, we selected the interactive-strategy. gTDG$^-$ performed better than exSATCHMO$^+$ (due to the fact that we selected the "right" cause). Despite the fact that gTDG$^+$ necessitates repairs like gTDG$^-$, the performance gain of the gTDG generator formula still is a factor of about three (for $x = 25$), due to the growing quantity of potential causes. gTDG$^-$ generates 3 persons, 2 cars and no owns-facts. Constraint 3 and 4 are violated and there are a fair number of potential causes to cure the inconsistencies: 64 for constraint 4 and 25 for constraint 3. The search space consists of all possible combinations, i.e., 1600 possible repairs as opposed to four in the +version.

6.2 A Benchmark Including Totality and Recursiveness

We now consider more difficult constellations of constraints by introducing a totality constraint on a binary relation ($rel : dom \rightarrow dom$) and a recursive constraint (the transitive closure of rel realized with a relation $rel\_trans : dom \rightarrow dom$).

Example 6.1: Let DB_{rel} be a database where DB^a is empty and DB^c contains the following consistency constraints:

1. $\forall x_1 \forall x_2 rel(x_1, x_2) \Longrightarrow (is(x_1, dom) \wedge is(x_2, dom))$
   rel relates elements of dom to elements of dom.

2. $\forall x_1 \exists x_2 \exists x_3 is(x_1, dom) \Longrightarrow (rel(x_1, x_2) \wedge rel(x_3, x_1))$
   The relation rel is total on dom.

3. $\forall x_1 \forall x_2 rel(x_1, x_2) \Longrightarrow rel\_trans(x_1, x_2)$
Every rel-relation is also a rel_trans-relation.

4. $\forall x_1 \forall x_2 \forall x_3 rel(x_1, x_2) \wedge rel\_trans(x_2, x_3) \Longrightarrow rel\_trans(x_1, x_3)$
There is a transitive closure on rel.

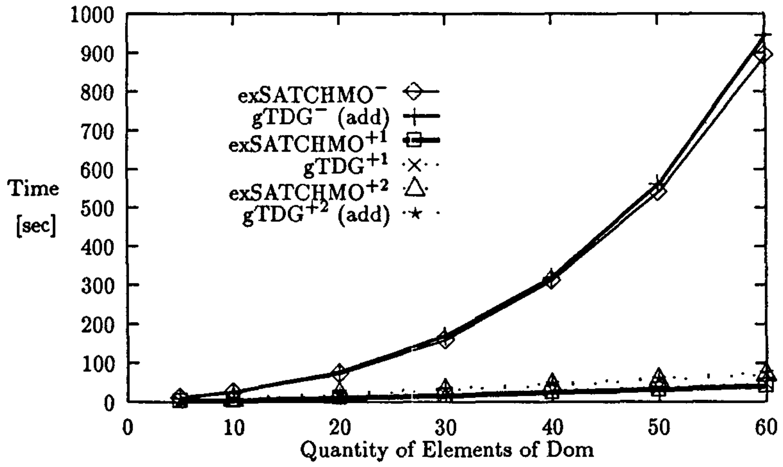
Suppose the user starts with $rel(x_1, x_2)$. After extending it and applying Reduction Rule 1, we obtain $rel(x_1, x_2) \wedge is(x_1, dom) \wedge is(x_2, dom) \wedge rel\_trans(x_1, x_2)$. Then the user is asked to substitute x_2 with x_1. If the user agrees, we arrive at the following generator formula: $rel(x_1, x_1) \wedge is(x_1, dom) \wedge rel\_trans(x_1, x_1) \wedge \ldots$ We leave the quantified formulas open, because they do not play a role in the generation process. The operator sequence looks like $gen - op(is(x_1, dom)), gen - op(rel(x_1, x_1)), gen - op(rel\_trans(x_1, x_1))$, $t\&r - op(\ldots), t\&r - op(DB^c)$. If the user rejects the proposed substitution but applies Reduction Rule 4 twice, the analysis yields a second, different sequence: $gen - op(is(x_1, dom)), gen - op(is(x_2, dom)), gen - op(rel(x_1, x_2)), gen - op(rel\_trans(x_1, x_2)), gen - op(rel(x_2, x_1)), gen - op(rel\_trans(x_2, x_1))$, $t\&r - op(\ldots), t\&r - op(DB^c)$.

We measured the CPU times to generate n elements of dom for gTDG and exSATCHMO, respectively. Again, we differentiated between +versions and –versions. Furthermore, we examined the behavior for both generator formulas and operator sequences. To mark which cost graph results from which generator formula, we added +1 and +2, respectively, to the name of the examined method. For this example, we chose the add-strategy for each benchmark. Figure 5 shows the resulting costs.

Both methods show an almost identical performance. Using generator formula 2 instead of 1 worsens the result by a factor of about 1.5. The +versions yield significantly better results than the –versions: In the case of 60 dom-elements, exSATCHMO^{+1} is better than exSATCHMO^{-} by a factor of 21. gTDG^{+1} is better than gTDG^{-} by a factor of 19. Taking the first generator formula both methods generate databases like $is(dom\_i, dom)$, $rel(dom\_i, dom\_i)$ and $rel\_trans(dom\_i, dom\_i)$ with $1 \leq i \leq n$. Consequently, no consistency constraint is violated and no repair is necessary. Because the specified quantities are still small, the inefficiency of the consistency check is not yet noticeable and the difference between exSATCHMO and gTDG remains slight. If the second generator formula is computed, both methods generate databases like $is(dom\_i, dom)$, $is(dom\_i + 1, dom)$, $rel(dom\_i, dom\_i + 1)$, $rel(dom\_i + 1, dom\_i)$, $rel\_trans(dom\_i, dom\_i + 1)$ and $rel\_trans(dom\_i + 1, dom\_i)$ with $i = 1, 3, 5, \ldots$. Repairs were necessary because of the transitivity of rel, which enforced additions of the form $rel\_trans(dom\_i, dom\_i)$ and $rel\_trans(dom\_i + 1, dom\_i + 1)$.

The poorer performance of the –versions is caused by the simultaneous violation of constraint 1 which requires adequate dom-facts, and of constraint 3 which requires the addition of the equivalent rel_trans-fact for each rel-fact. These facts are

Figure 5. Cost of generating elements of dom according to DB_{rel}



added in the course of repairs. Let us first consider exSATCHMO$^-$ which generates and tests incrementally. Suppose the ith step, which inserts $rel(dom\_i, dom\_i)$ $(1 \leq i \leq n)$, is executed. Constraint 1 and 3 are violated. The first is repaired by adding $is(dom\_i, dom)$, the latter by inserting $rel\_trans(dom\_i, dom\_i)$. Hence, we arrive at the same database as if taking the first generator formula. Although no backtracking is necessary, the performance loss is significant. gTDG$^-$ generates the same database as exSATCHMO$^-$.

Now we change the example by substituting a disjunctive constraint for the totality constraint so that the transitive closure of rel totally covers dom. Call the resulting database DB'_{rel}.

Example 6.2 Let DB'_{rel} be a database where DB^a is empty and DB^c contains the following consistency constraints:

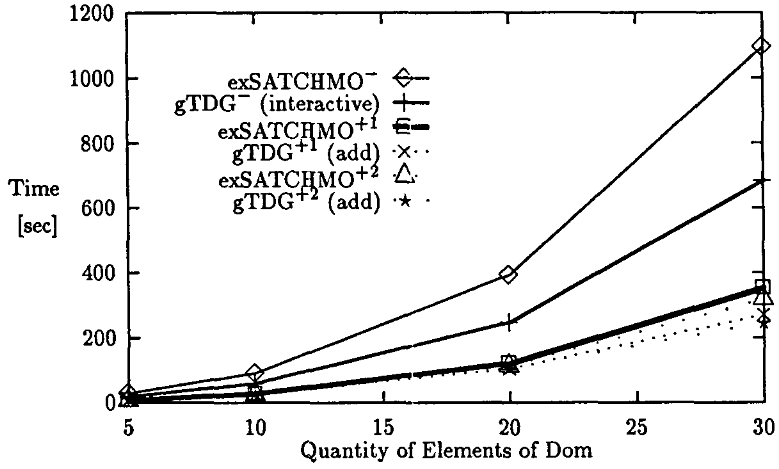
1. $\forall x_1 \forall x_2 rel(x_1, x_2) \Longrightarrow (is(x_1, dom) \wedge is(x_2, dom))$
   rel relates elements of dom to elements of dom.

2. $\forall x_1 \forall x_2 is(x_1, dom) \wedge is(x_2, dom) \wedge x_1 \neq x_2 \Longrightarrow (rel\_trans(x_1, x_2) \vee rel\_trans(x_2, x_1))$. All elements of dom are related by rel_trans with one another.

3. $\forall x_1 \forall x_2 rel(x_1, x_2) \Longrightarrow rel\_trans(x_1, x_2)$
   Every rel-relation is also a rel_trans-relation.

4. $\forall x_1 \forall x_2 \forall x_3 rel(x_1, x_2) \wedge rel\_trans(x_2, x_3) \Longrightarrow rel\_trans(x_1, x_3)$
   There is a transitive closure on rel.

Figure 6. Cost of generating elements of dom according to DB'_{rel}



Starting with $rel(x_1, x_2)$ yields $rel(x_1, x_1) \wedge is(x_1, dom) \wedge rel\_trans(x_1, x_1)$ $\wedge \ldots$ and $rel(x_1, x_2) \wedge is(x_1, dom) \wedge is(x_2, dom) \wedge rel\_trans(x_1, x_2) \wedge \ldots$, if the substitution $[x_2 \leftarrow x_1]$ proposed by Reduction Rule 6 is rejected. Again, we leave the quantified formulas open, because they do not play a role in the generation process. The operator sequence corresponding to the first generator formula is the same as the first one of example 6.1: $gen-op(is(x_1, dom)), gen-op(rel(x_1, x_1)), gen-op(rel\_trans(x_1, x_1)), t\&r-op(\ldots), t\&r-op(DB^c)$. The sequence resulting from the second generator formula is $gen-op(is(x_1, dom))$, $gen-op(is(x_2, dom)), gen-op(rel(x_1, x_2)), gen-op(rel\_trans(x_1, x_2))$, $t\&r-op(\ldots), t\&r-op(DB^c)$. The plot of Figure 6 shows the CPU times generate n elements of dom for both methods and both generator formulas. The indices +1 and +2, respectively, denote which generator formula was taken. For gTDG^{+1} and gTDG^{+2}, we chose the add-strategy; for gTDG^{-}, the interactive-strategy.

In comparison to Figure 5, the performance gain of the +versions over the -versions decreased from 8 to 2.8 for gTDG and from 9 to 3.4 for exSATCHMO when generating 30 elements of dom. This is due to the fact that, despite the use of the generator formula, the additional disjunctive constraint causes the need for repairs in the +versions.

Although exSATCHMO^{-} needs no backtracking, gTDG^{-} is superior to exSATCHMO^{-}. Both start with generating facts like $rel(dom\_i, dom\_i)$, so that constraints 1 and 3 are violated. To cure them, the insertion of $is(dom\_i, dom)$ and $rel\_trans(dom\_i, dom\_i)$ is required. These additional facts cause a violation of the disjunctive constraint. exSATCHMO generates $rel\_trans(dom\_i, dom\_j)$ and $rel\_trans(dom\_j, dom\_i)$ with $1 \leq j < i$ to cure it (all possible combinations are generated). For 30 elements this results in 900 rel_trans-facts. Due to the interactive-strategy selected for gTDG$^-$, we were able to restrict ourselves to the necessary combinations; for 30 elements we arrive at 435 plus the symmetric ones (30), which are generated because of the generator formula, i.e., 465 altogether.

The +versions show an almost identical performance. The versions with the second generator formula yield better results than those with the first one. Starting from the second generator formula, both methods generate facts like $rel\_trans(dom\_i, dom\_i + 1)$ with the result that fewer violations of the disjunctive constraint occurred than in the version with the first generator formula. Still, the difference between these versions is only about a factor of 1.2 for both exSATCHMO and gTDG.

We also observe that the +versions of gTDG show a slightly better performance than the corresponding ones of exSATCHMO and, although the add-strategy was selected, they generate smaller test databases than the +versions of exSATCHMO. Both +1-versions generate the same test database as their –versions. When starting with the second formula the +2-versions reduce the +1 test databases by the symmetric facts like $rel\_trans(dom\_i, dom\_i)$. Hence, although we chose the add-strategy for gTDG^{+1} and gTDG^{+2}, they produce smaller test databases than exSATCHMO^{+1} and exSATCHMO^{+2}, respectively. This is due to the simplifications of the logical expressions resulting from the derivation trees (Section 5.2.4). The better performance is a consequence of the smaller test database: fewer constraint violations are observed because fewer test data are generated.

6.3 Adding Negative and Uniqueness Constraints to the Benchmark

Consider the simple data type *chain* (or *sequence*). Expressing it in terms of consistency constraints leads to a combination of existentially quantified, recursive, disjunctive, and negative constraints. Suppose the objects of the chain are called nodes. We introduce relationship *next : node* ⟶ *node* to model the fact that two nodes are directly connected, and relationship *connected : node* ⟶ *node* to model the indirect connection, i.e. the transitivity of $next$.

Example 6.3 Let DB_{chain} be a database where DB^a is empty and DB^c contains the following consistency constraints:

1. $\forall x_1 \forall x_2 next(x_1, x_2) \Longrightarrow (is(x_1, node) \wedge is(x_2, node))$
   $Next$ relates nodes to nodes.

2. $\forall x_1 \forall x_2 connected(x_1, x_2) \Longrightarrow (is(x_1, node) \wedge is(x_2, node))$
   $Connected$ relates nodes to nodes.

3. $\forall x_1 \forall x_2 next(x_1, x_2) \Longrightarrow connected(x_1, x_2)$
   Every $next$-relation is also a $connected$-relation.

4. $\forall x_1 \forall x_2 \forall x_3 : next(x_1, x_2) \wedge connected(x_2, x_3) \Longrightarrow connected(x_1, x_3)$
   Connected is the transitive closure of *next*.

5. $\forall x_1 is(x_1, node) \Longrightarrow \neg connected(x_1, x_1)$
   No cycles are possible.

6. $\forall x_1 \forall x_2 \forall x_3 next(x_1, x_2) \wedge next(x_1, x_3) \Longrightarrow x_2 = x_3$
   No branching to the right side is possible.

7. $\forall x_1 \forall x_2 \forall x_3 next(x_1, x_2) \wedge next(x_3, x_2) \Longrightarrow x_1 = x_3$
   No branching to the left side is possible.

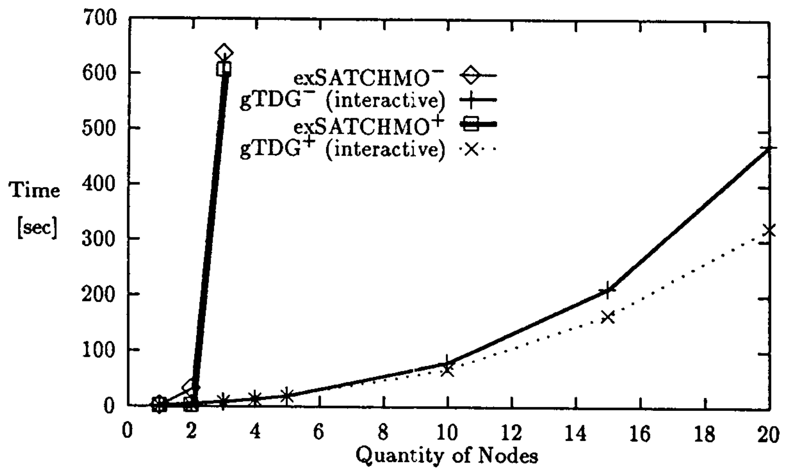
8. $\forall x_1 \forall x_2 \forall x_3 : is(x_1, node) \wedge is(x_2, node) \wedge x_1 \neq x_2 \Longrightarrow$
   $(connected(x_1, x_2) \vee connected(x_2, x_1))$
   All nodes are connected with each other.

9. $\forall x_1 \forall x_2 connected(x_1, x_2) \Longrightarrow (next(x_1, x_2) \vee (\exists x_3 next(x_1, x_3) \wedge$
   $connected(x_3, x_2))$
   Every *connected*-relation is decomposable into *next*-relations.

Suppose the user starts with $next(x_1, x_2)$. After extending it and applying reduction rule 1, we obtain $next(x_1, x_2) \wedge is(x_1, node) \wedge is(x_2, node) \wedge connected(x_1, x_2)$. Then the user is asked to substitute x_2 with x_1. If the user agrees, a contradiction is reported because of constraint 5, which forbids cycles. Hence, we arrive at the following operator sequence: $gen-op(is(x_1, node)), gen-op(is(x_2, node)), gen - op(next(x_1, x_2)), gen - op(connected(x_1, x_2))$, $t\&r - op(\ldots), t\&r - op(DB^c)$. (Again we leave open the quantified formulas because they do not play a role in the generation process). Figure 7 illustrates the CPU times to generate n nodes for both methods varied by the generator formula. For gTDG$^-$ and gTDG$^+$, we chose the interactive-strategy.

Figure 7 shows that the performance of exSATCHMO dramatically decreases. Note the situation for exSATCHMO$^+$ after $is(node\_1, node)$, $is(node\_2, node)$, $next(node\_1, node\_2)$, $connected(node\_1, node\_2)$ have been generated. If three nodes have been requested, $is(node\_3, node)$ $next(node\_3, node\_4)$, $connected(node\_3, node\_4)$ with $node\_4)$, a new constant is added. Constraints 1 and 8 are violated. To cure the first $is(node\_4, node)$ is inserted. The latter is repaired by adding $connected(node\_1, node\_3)$, $connected(node\_2, node\_3)$ and $connected(node\_3, node\_1)$, $connected(node\_3, node\_2)$. This database is the starting point for a time-consuming search which remains unsuccessful because the last two facts must be deleted to become consistent again. That means exSATCHMO has to backtrack to that point. Due to constraint 9, which enforces the generation of new constants, the memory space needed to store all alternative points for backtracking in the tree increases so that it is not possible to generate a test database for more than three nodes.

Figure 7. Cost of generating nodes according DB_{chain}



In gTDG the +version also necessitates repairs. Due to the interactive-strategy, we could directly select causes and arrive at a test database with the appropriate quantities.

6.4 Summary

We summarize the results obtained from our benchmarks as follows:

1. Starting from a generator formula results in a remarkable performance gain for both gTDG and exSATCHMO.

2. Due to the incremental consistency check, exSATCHMO performs better than gTDG when no repairs are necessary. However, the differences are often rather small when the same generator formula strategy is used. gTDG then offers the advantage of sometimes generating smaller test databases.

3. If the consistency constraints are more complex, the search for repairs can lead to extremely bad performance and to memory problems. This is where the advantages of gTDG of being able to select among various search and repair strategies become clearly visible.

The generation of realistically sized test databases is only possible in a semi-automatic fashion, with interactive support by the user needed to find a potential cause. We feel that our approach represents a significant advance because higher functionality and increased flexibility have been achieved without loss in performance. Moreover, the gains for more complex consistency constraints are considerable.

7. Conclusion

We propose a two-step approach to generate test data for an arbitrary set of general consistency constraints:

1. A generator formula is constructed that covers interdependencies between consistency constraints and gives rise to a sequence of operations that collectively preserve consistency. Some optimizations that are not necessarily equivalence-preserving can be performed in a user-controlled way.

2. The generator formula is translated into a sequence of two very powerful operators: the generation operator and the test-and-repair operator. The generation operator can be enhanced by an additional parameter which allows the user to generate facts in a goal-directed fashion.

This approach has been implemented and successfully tested within a database design environment that serves as a rapid prototyping tool for validating semantic schemas. We compared our approach with a pure model-generating approach for satisfiability checking (SATCHMO), and extended it with additional features for test data generation. The performance gain resulting from the analysis and computation of a generator formula has been shown to be impressive in both approaches. However, in contrast to SATCHMO, our generation procedure provides more flexibility so that there is no fixed order of searching for repairs and different strategies can be chosen. If the constraints become too complicated, exceeding memory space and causing bad performance, this can be overcome only by incorporating the user into the selection process. Our approach is interactive and, thus, exploits additional knowledge a user might have.

Future research will gain more experience with pragmatics, which also might be desirable for test data generation purposes. Furthermore, we plan to examine how to provide more user support in selecting a non-equivalence-preserving reduction rule, determining a strategy for finding repairs, and attaching a relation type as an additional parameter to the generation operators.

Acknowledgments

We gratefully acknowledge the many fruitful comments of the anonymous referees. The work reported in this article was partially supported by Deutsche Forschungsgemeinschaft (German Research Council) under grant no. Lo 296/5.

References

Alavi, M. An assessment of the prototyping approach to information systems development. *Communications of the ACM*, 27:556–563, 1984.

Bitton, D., DeWitt, D.J., and Turbyfill, C. Benchmarking database systems—A systematic approach. *Proceedings of the International Conference on VLDB*, Florence, 1983.

Brodie, M.L. and Ridjanovic, D. On the design and specification of database transactions. In: Brodie, M.L., Mylopoulos, J., and Schmidt, J.W., eds., *On Conceptual Modelling. Perspectives from Artifical Intelligence, Databases, and Programming Languages*, Springer-Verlag: New York, 1984, pp. 277–306.

Bry, F. and Manthey, R. Checking consistency of database constraints: A logical basis. *Proceedings of the International Conference on VLDB,* Kyoto, 1986.

Bry, F., Decker, H., and Manthey, R. A uniform approach to constraint satisfiability and constraint satisfiability in deductive databases. *Proceedings of the International Conference on Extending Database Technology,* Venice, 1988.

Budde, R., Kuhlenkamp, K., Mathiassen, L., and Züllighoven, H., eds. *Approaches to Prototyping.* Springer-Verlag: New York, 1984.

Cosmadakis, C.C. and Papadmitriou, C.H. Updates of relational views. *Journal of the ACM,* 31(4):742–760, 1984.

DeWitt, D.J. Benchmarking database systems: Past efforts and future directions. *IEEE Database Engineering,* 8(1):2–9, 1985.

Fagin, R. Horn clauses and database dependencies. *Journal of the ACM,* 28(4):952–985, 1982.

Fagin, R. and Vardi, M. Armstrong databases for functional and inclusion dependencies. *Information Processing Letters,* 16:13–19, 1983.

Geibel, P. Entwicklung und bewertung von strategien zur testdatengenerierung. In German. *Master's thesis,* Fakultät für Informatik, Universität Karlsruhe, April, 1991.

Guessoum, A. and Lloyd, J.W. Updating knowledge bases. *New Generation Computing,* 8(1):71–89, 1990.

Guessoum, A. and Lloyd, J.W. Updating knowledge bases II. *New Generation Computing,* 10(1):73–100, 1991.

Kung, C.H. A tableaux approach for consistency checking. In: Sernadas, A., Bubenko, J., Jr., and Olive, A., eds., *Proceedings of IFIP Conference on Theoretical and Formal Aspects of Information Systems,* Amsterdam: North-Holland Publishing Company, 1985, pp. 191–210.

Lockemann, P.C., Moerkotte, G., Neufeld, A., Radermacher, K., and Runge, N. Database design with user-definable modelling concepts, *Data and Knowledge Engineering* (submitted) .

Manchanda, S. and Warren, D.S. Towards a logical theory of database view updates. *Proceedings of the Workshop on Foundations of Deductive Databases and Logic Programming,* Washington, DC, 1986.

Mannila, H. and Räihä, K.-J. Automatic generation of test data for relational queries. *Journal of Computer and System Sciences,* 38(2):240–258, 1989.

Manthey, R. and Bry, F. A hyperresolution-based proof procedure and its implementation in PROLOG. In: Morik, K., ed., *German Workshop on Artificial Intelligence,* Berlin Heidelberg: Springer-Verlag, Informatik-Fachberichte 152, 1987, pp. 221–230.

Manthey, R. and Bry. SATCHMO: A theorem prover implemented in PROLOG.

In: Lusk, E. and Overbeck, R., eds., *Ninth International Conference on Automated Deduction*, Argonne, IL, Berlin Heidelberg: Springer-Verlag, 1988, pp. 415–434.

Moerkotte, G. and Lockemann, P.C. Reactive consistency control in deductive databases. *ACM Transactions on Database Systems*, 16(4):670–702, 1991.

Moerkotte, G. and Schmitt, P.H. Analysis and repair of inconsistencies in deductive databases. *Journal of Logic Programming* (submitted).

Neugebauer, L. and Neumann, K. Schema-driven test data generation for relational database systems. Informatik-Bericht 8502, TU Braunschweig, Institut für Informatik, Abtl. Datenbanken und Informationssysteme, 1985.

Ngu, A.H. Conceptual transaction modeling. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):508–518, 1989.

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

Noble, H. The automatic generation of test data for a relational database. *Information Systems*, 8(2):79–86, 1983.

Oberweis, A. Schönthaler, F., Lausen, G., and Stucky, W. Net-based conceptual modelling and rapid prototyping with INCOME. *Proceedings of the Third Conference on Software Engineering*, Paris, 1986.

Reiter, R. On closed world data bases. Gallaire, H. and Minker, J., eds., *Logic and Data Bases*, Plenum Publishing: New York, 1978, pp. 55–76.

Röhrle, J. Dynamic test data generation for the evaluation of database prototypes. *Informatik Forschung und Entwicklung*, 4:139–148, 1989.

Rossi, F. and Naqvi, S.A. Contributions to the view update problem. *Proceedings of the Sixth International Conference on Logic Programming*, Lisbon, 1989.

Schönthaler, F. Rapid prototyping to support the conceptual design of information systems. *Ph.D. thesis,* Universität Fridericiana Karsruhe, 1989.

Silva, A.M. and Melkanoff, M.A. A method for helping discover the dependencies of a relation. In: Gallaire, H., Minker, J., and Nicolas, J.-M., eds., *Advances in Data Base Theory*, Vol. 1. Plenum Publishing: New York, 1981.

Stonebraker, M. Tips on benchmarking data base systems. *IEEE Database Engineering*, 8(1):10–18, 1985.

Tomasic, A. View update translation via deduction and annotation. *Proceedings of the International Conference on Database Theory,* Bruges, 1988.

Ullman, J.D. *Principles of Database and Knowledge-Based Systems,* Vol. 1. Rockville, MD: Computer Science Press, 1988.