# Automating Proofs of Integrity Constraints in Situation Calculus

Leopoldo Bertossi, Javier Pinto, Pablo Saez[1],
Deepak Kapur and Mahadevan Subramaniam[2]

**Abstract.** Automated support for proving integrity constraints (ICs) on deductive database update specifications is developed using an induction theorem prover, *Rewrite Rule Laboratory* (*RRL*) [6]. The approach proposed by Reiter [9, 11, 10] for solving the frame problem for such applications in a language of the situation calculus is used as a basic framework. Integrity constraints are propositions that are expected to be true in every accessible state of a database, and they should be provable from the specification of the evolution of the database. Accessible states are defined by induction [12] as those reachable from the initial state by update actions whose execution is possible. Induction theorem provers can only reason about quantifier-free formulas (i.e., universally quantified formulas) whereas in order to express integrity constraints, quantifiers may be used. It is shown that by making use of the fact that in relational data base applications, domain of objects under consideration is finite, such ICs expressed using quantifiers can be mechanically translated into quantifier-free formulas by introducing new predicates and by explicitly building domains of objects involved in updates. Bridge lemmas connecting the semantics of the new predicates to the fluents used to express integrity constraints can be mechanically generated and automatically proved in *RRL*. An interesting feature of the proposed approach is that mechanically generated proofs of integrity constraints have a structure similar to manually-generated proofs.

## 1 Introduction

In [9], Reiter proposed a solution to the frame problem in theories of action and change written in the situation calculus [7]. In [11], he applied this solution to specifications of deductive databases. In [12], he discussed an application of this approach for proving integrity constraints in deductive data bases. The importance of doing proofs by mathematical induction is clearly evident in these papers. In this paper, we show how such proofs can be mechanized using an induction theorem prover. We report our experience in using *Rewrite Rule Laboratory* (*RRL*), a theorem prover based on rewrite techniques [6].

Specification of theories of action include sentences expressing changes introduced when actions are performed. In most cases, it is desirable to omit the sentences that describe the properties of the world that remain unchanged if an action gets executed. From a logical viewpoint, non-change must be derivable from the logical specification. The problem of inferring non-change from a logical specification of theories of action is referred to as the *frame problem* and has received a great deal of attention in the knowledge representation community.

Reiter's partial solution to the frame problem (only deterministic actions and no explicit ramifications) [9] transforms preliminary specifications about evolving worlds into monotonic specifications. Roughly speaking, one starts from a preliminary specification that includes a description of the world (from the viewpoint

[1] Pontificia Universidad Católica de Chile, Departamento de Ciencia de la Computación, Santiago 22, Chile. E-mail: {bertossi, jpinto, pdsaez}@ing.puc.cl.
[2] Department of Computer Science, State University of New York at Albany, Albany, NY 12222. E-mail: {kapur, subu}@cs.albany.edu.

of the database) along with a description of the possible transactions that can be performed. Preliminary specifications embody implicit common-sense assumptions about the possible explanations for changes in truth values of *fluents*, properties that may change as the result of actions performed. The specification of the transactions/actions includes details regarding their preconditions and their effects on the world. The transformed, final, specification contains everything included in the preliminary specification, except for the effect axioms that are replaced by successor state axioms for each fluent. An effect axiom fully specifies when the fluent is true in a successor state obtained by executing an arbitrary possible action. The final specification includes unique name axioms for actions and states.

In addition to the specification, a set of *integrity constraints*, (ICs), are given. They describe certain conditions that the database must always satisfy. An important goal is to ensure that starting from an initial state, and after a sequence of transactions possible on a data base, the integrity constraints are satisfied by the resulting state of the data base. In [12], Reiter showed how such proofs could be manually done using his proposal to solve the frame problem.

Traditional database management systems do not provide facilities for proving that the integrity of the database cannot be violated as a result of performing transactions. Rather, these systems provide facilities for testing the integrity of the database whenever a transaction takes place. There is no guarantee that integrity constraints can never be violated. Testing of only very primitive ICs is supported.

This work forms part of a research being conducted at the Catholic University in Chile, where an automated system called *SCDBR* to reason from and about database specifications is being built [1].

## 2 A Library Example

In this section we present partial aspects of an example of a library database. The example is very simplified in order to keep the paper succinct. The logical specification of the initial database, the transactions and their effects is given in a situation calculus language, with an additional second-order axiom that inductively defines an *accessibility relation* between situations. This specification is given in the form proposed by Reiter [11].

Fluents are used to model the dynamic properties of a domain. In this example, we consider a single fluent $classified$. Every book in the database is classified with its $ISBN$ number and a unique $id$. Two books with the same $ISBN$ denote copies of the same book and should have different $id$ numbers. The fluent is modeled by a ternary predicate $classified$ that takes an isbn, an id and a situation as arguments. $classified(Isbn, Id, S)$ is true if in situation $S$, the book with id $Id$ is classified with isbn $Isbn$. In a realistic example a number of other fluents such as *Stock*, *LostBook*, *SoldOut* etc., could be included to model properties of the domain. In addition to fluents, a situation calculus theory may also include static properties, i.e., properties that don't have a situation argument (e.g., the authorship of a book). Our example does not include such properties.

To specify transactions, we need to state: 1) when they are possible, and 2) what changes they provoke. The first is done with *action preconditions* while the second is specified with *successor state axioms.*

*Action Preconditions.* Action preconditions are formulas that specify the conditions on the state of a data base that have to be met for actions to be executable. For the example, we only consider the actions *classifybook* and *deleteBook*.

We consider that the transaction *deleteBook* is always possible and that the transaction $classifyBook$ is possible if no book is classified with the same id:

$$poss(deleteBook(id), s) \equiv true, \tag{1}$$

$$poss(classifyBook(isbn, id), s) \equiv \neg\ \exists isbn'\ classified(isbn', id, s). \tag{2}$$

*Successor State Axioms.* Successor state axioms provide necessary and sufficient conditions for a fluent to hold after performing an action. For these conditions to be valid, the action's preconditions must be satisfied. In general, these axioms take the form $poss(a,s) \supset f(do(a,s)) \equiv \phi_f^+(a,s) \vee (f(s) \wedge \neg\phi_f^-(a,s))$. Thus, a fluent $f$ holds after performing a possible action $a$ if it held before the action was performed and the action did not falsify $f$ ($\neg\phi_f^-$) or the fluent was made true by $a$ ($\phi_f^+$) [10, 11, 12]. There will be one such axiom per fluent predicate. For instance, for $classified$ we have:

$$poss(a,s) \supset classified(isbn, id, do(a,s)) \equiv (a = classifyBook(isbn, id) \vee (classified(isbn, id, s) \wedge a \neq deleteBook(id))).$$

*Integrity Constraints:* Aside from the specifications given above, in general a set of integrity constraints is included in a database specification. These constraints specify conditions that must be met in all valid states of the database. In the example, we use the constraint below, which states that for any situation $s$, any two books with same identification number have identical ISBN numbers.

$$classified(isbn1, id, s) \wedge classified(isbn2, id, s) \supset isbn1 = isbn2. \qquad (3)$$

ICs are specified to hold for the initial state of the database. Then, they must be shown to hold in all states of the database that are accessible from the initial state by means of a finite sequence of actions.

### 2.1 A Typical Hand Proof of an Integrity Constraint

The initial state of the database is identified with the situation constant $S_0$. The actions $classifyBook$ and $deleteBook$ – which classify and delete books, respectively – permit one to reach new states starting from other states. It is assumed that a $deleteBook$ action is always possible (1). A $classifyBook$ action with an identification number $id$ and an ISBN number $isbn$ is possible in a state $s$ only if there does not exist any classified book in the database with which $id$ is associated (2). We briefly review a proof of the integrity constraint stating that the fluent $classified$ is functional (3).

1. It seems quite natural to attempt a proof by induction on states. The basis case is obtained by directly replacing $s$ with $S_0$ in (3). The resulting sentence would typically follow from the definition of predicate $classified$ in $S_0$, ensuring that $IC$ holds in the initial state $S_0$.
2. The induction step case involves exhibiting that if $IC$ holds in a state $s$, then it also holds in $do(a,s)$ obtained by performing any action $a$ that is possible in $s$. The conclusion is (4) and the inductive hypothesis is (5):
$$classified(isbn1, id, do(a,s)) \wedge classified(isbn2, id, do(a,s))) \supset (isbn1 = isbn2). \quad (4)$$
$$classified(isbn1, id, s) \wedge classified(isbn2, id, s)) \supset (isbn1 = isbn2). \quad (5)$$
3. This can be shown by case analysis on the different types of actions (which is induction on a finite domain). Let us consider the case $a = classifyBook(isbn', id')$. By the successor state axiom of the fluent $classified$, $classified(isbn1, id, do(a,s))$ is equivalent to
$(a = classifyBook(isbn1, id)) \vee (classified(isbn1, id, s) \wedge a \neq deleteBook(id))$.
For the case when $a = classifyBook(isbn', id')$, this simplifies to:
$$(isbn' = isbn1 \wedge id' = id) \vee classified(isbn1, id, s),$$
since $classifyBook$ is a free constructor. Similarly, $classified(isbn2, id, do(a,s))$ is equivalent to
$$(isbn' = isbn2 \wedge id' = id) \vee classified(isbn2, id, s).$$
Each of the four possible combinations of the above literals leads to $isbn1 = isbn2$ : *a)* if $isbn' = isbn1 \wedge isbn' = isbn2$, then $isbn1 = isbn2$, *b)* if $classified(isbn1, id, s)$ and $classified(isbn2, id, s)$ are true, then $isbn1 = isbn2$ by induction hypothesis. *c)* the remaining two combinations lead to contradictory assumptions and $isbn1 = isbn2$ is vacuously true.

4. The induction step corresponding to the case $a = deleteBook(id')$ can be established in a similar manner.

## 3 Formalizing Situation Calculus in *RRL*

A main concept in situation calculus is that of *situations* or *states* of the evolving world. Starting with an initial state, actions possible in a current state are executed to get new states. This can be modeled as a data type, which we call `state`. It has two constructor functions: (i) the constant constructor `s0` (corresponding to the initial state `s0` of the situation calculus), and (ii) a constructor `do` which given an action and a state, gives the resulting state after the action assuming that its execution is possible. The constructor `do` is a function from `act` and `state` to `state`, where `act` is a data type modeling actions.

`do` is a partial constructor; its value is undefined if a particular action is not possible in a given state. Such partial constructors cannot be modeled directly[3]. Instead, this is done in two parts: (i) define `do` as a total constructor, and then (ii) define a predicate `reach` to identify the subset of states *reachable* by executing actions possible in other reachable states. The predicate `reach` on states is recursively defined: the initial state `s0` is reachable; `do(a,s)` is reachable if `s` is reachable and action `a` is possible in `s`, i.e., `poss(a,s)` holds.

Reiter assumed that in the situation calculus, different actions executed in a particular state lead to different states, and that if two actions lead to the same state, then the actions must be the same; also, no action can lead to the initial state. This can be specified by declaring the constructors `s0` and `do` as *free*, much like the `successor` for natural numbers and `cons` for lists.

In Reiter's specifications, the definition of the `reach` predicate needs an induction axiom. Such use of induction is implicit in the specification of `state` when it is stated that `state` has `s0` and `do` as its free constructors.

The data type `act` is specified by introducing a constructor for each action type. For example, the action type `classifyBook` is specified as a constructor with `isbn` and `id` as arguments, and produces a value, which is the action corresponding to that `isbn` and `id` numbers. Other actions are similarly specified. The constructors of `act` are free since actions are assumed to be distinct.

In order to specify the preconditions on actions, the predicate `poss` is specified; it takes an action and a state as arguments and decides whether the action can be executed in the state. See below how the preconditions for `deleteBook` and `classifyBook` action types are expressed.

Fluents are specified as predicates. For example, the fluent `classified` is recursively defined below by specifying the effect of actions on it. In the initial state, the table for the fluent `classified` is given as a boolean expression. In this example, for simplicity, we have used a very short table for `classified`, which is specified to be true only if the isbn number is `i1` in which case ids can be `i2` or `i3`, or if the isbn number is `i4` with the id to be `i5`. How an action affects `classified` is expressed in the second conditional equation given below. Effects of actions on other fluents can be defined in a similar way. A part of the equational specification of the library example as input into *RRL* is given below.

```
reach(s0)                          := true,
reach(do(xa, xs))                  := reach(xs) and poss(xa, xs).
poss(deletebook(xid), xs)          := true,
poss(classifyBook(xisbn, xid), xs) := ball(xid, xs, dom(xs)).
classified(xisbn, xid, s0)         := cond(xisbn = i1 and xid = i2,
 true, cond(xisbn = i1 and xid = i3, true, xisbn = i4 and xid = i5)),
classified(xisbn, xid, do(xa, xs)) := cond(xa = classifyBook(xisbn, xid),
```

[3] See [5] for a method to extend algebraic axiomatizations to specify partial constructors.

```
true, cond(xa = deleteBook(id), false, classified(xisbn, xid, xs)))
```

It should be evident to the reader that except for the equation for the possibility axiom for `classifyBook`, it is quite straight-forward to generate the above quantifier-free equational axiomatization from Reiter-style specification. Such possibility axioms involving quantifiers are discussed in the next subsection. The translation of the successor state axiom for the fluent `classified` turned out to be straightforward because it is quantifier-free. For certain fluents, the action specifications in the successor state axiom may involve quantifiers. In such cases, translating successor state axioms directly as shown above can lead to equations with extra variables on their right-hand sides. These equations cannot be oriented into rewrite rules by *RRL*. This can be avoided by explicitly instantiating the action $a$ in the left side of successor state axiom of the fluent with every action type and then performing the translation.

*Bounded quantification* Preconditions for the actions as well as successor state axioms for fluents could be arbitrary first-order formulas. For instance, in the library example, the precondition for $classifyBook$ is equivalent to:

$$poss(classifyBook(xisbn, xid), xs) \equiv (\forall\ xisbn1)\ \ \neg classified(xisbn1, xid, xs).$$

For automation, it becomes necessary to eliminate such quantifiers so that explicit instantiation of quantified variables can be done automatically.

In relational database applications, there are usually a finite number of objects that satisfy a fluent. Further, an action can only introduce finitely many objects. The domain of objects associated with a given state is finite. Using this observation, it is possible to replace arbitrary quantification by *bounded* quantification. A new function called `dom` is introduced on states; `dom(s)` stands for the finite set of objects possibly existing in the state `s`. Quantification in state `s` is then reduced to that over `dom(s)`, and fluents are evaluated relative to `dom(s)`. We thus replace quantifiers by bounded quantifiers and introduce range of the variables to be `dom(s)`. The domain `dom(s)` is constructed by collecting the objects in the initial state and those *mentioned* in any sequence of actions leading to the state `s` from the initial state `s0`. This view of databases was proposed by Reiter in [8].

For the library example, the domain construction can be expressed as:

$$\mathtt{dom(s0)} := \mathtt{c(a(i1),c(b(i2),c(b(i3),c(a(i4),c(b(i5),nothing)))))}$$

$$\mathtt{dom(do(classifybook(xisbn,xid),xs))} := \mathtt{c(a(xisbn),c(b(xid),dom(xs)))}$$

$$\mathtt{dom(do(deletebook(xid),xs))} := \mathtt{c(b(xid),dom(xs))}.$$

For every action, the domain constructor is used to introduce into the domain, the arguments passed to the action. The set `nothing` stands for the empty domain. A constructor `c` is used to add a new object to a domain. The function symbols `a` and `b` coerce `isbn` and `id` types to uniformly typed objects in the list.

Bounded quantifiers are simulated using new predicates defined in terms of fluents. A bounded quantifier predicate must be introduced for each formula on which the bounded quantifier acts. For a bounded universal (for all) quantifier on a formula `f`, we introduce a new predicate `ball_f` (`ball_f` stands for bounded for all over `f`); similarly, for a bounded existential (there exists) quantifier on `f`, a new predicate `bexists_f` is introduced. Since `bexists_f` can be expressed as $\neg$ `ball_f` $\neg$, we discuss how a bounded universal quantifier can be eliminated. The new predicate `ball_f` is of arity $k$ if `f` has $k$ free variables, say $x_1, \cdots x_k$ (we are assuming that the current state variable appears free in `f`; if not, then the arity of `ball_f` is $k+1$ to include a state variable also). The predicate `ball_f`$(x_1, \cdots, x_k,$ `dom(xs)`) will not have the quantified variable as its argument; instead, `dom(xs)`, the domain of the current state variable `xs`, is an argument to `ball_f`.

Given a formula with quantifiers, say $\forall\ x$ `f`, we define `ball_f`. If `f` is quantifier-free, then `ball_f` is expressed using `f`; otherwise if `f` is a formula with a quantifier, say $\exists\ y$ `g`, this procedure is recursively applied by introducing a predicate `bexists_g`, and so on. This translation can be mechanically done.

The definition of new predicates can be mechanically generated by induction on the domain. For the empty domain `nothing`, `ball_f` is true (`bexists_f` will be false on `nothing` domain). For each type of object possibly introduced by an action, simulated by the application of the domain constructor `c` on a given domain `xdom`, `ball_f` on the extended domain is the same as `ball_f` on the given domain if the quantification is not on that type of variable; otherwise, it is the conjunction of `ball_f` on the given domain and `f` applied on the object introduced by `c` (in the case of `bexists_f`, it will be a disjunction).

To translate the possibility axiom for `classifyBook`, the new predicate introduced is `ball_classified`, abbreviated as just `ball`. Its arguments are `xid, xs, dom(xs)` since `xid, xs` are the free variables of the possibility axiom of `classifyBook`; the quantified variable `xisbn` is replaced by `dom(xs)`. The equation (4) is thus replaced by $\mathtt{poss(classifyBook(xisbn, xid), xs)} := \mathtt{ball(xid, xs, dom(xs))}$. The definition of `ball` is:

$$\mathtt{ball(xid, xs, c(a(xisbn), xdom))} := \mathtt{ball(xid, xs, xdom)} \wedge \neg \mathtt{classified(xisbn, xid, xs)},$$

$$\mathtt{ball(xid, xs, c(b(xid1), xdom))} := \mathtt{ball(xid, xs, xdom)},$$

$$\mathtt{ball(xid, xs, nothing)} := \mathtt{true}.$$

It can be easily verified that the above equations ensure that for each *isbn* in the domain corresponding to a state $s$, the book with identification number $xid$ is not $classified$ in the state $s$.

In order to connect the semantics of the new predicates to the fluents, some intermediate lemmas (which we call *bridge lemmas*) must be proved. For a bounded universal predicate `ball_f`, a typical lemma needed is: $f(x_1, \cdots x_k)\ =\ true\ if\ ball\_f(x_1, \cdots, x_k, dom(xs))$.

Recall that the predicate `ball_f` is of arity $k$ and the quantified variable is not one of its arguments. The bridge lemmas for a bounded existential predicate `bexists_f` is similarly constructed. Bridge lemma for the predicate `ball` is,

$$\mathtt{classified(xisbn, xid, xs)}\ =\ \mathtt{false\ if\ ball(xid, xs, dom(xs))}.$$

In general, a lemma for each new predicate may be needed. It is possible to reduce the number of new predicates to be introduced by considering blocks of universal quantifiers (as well as blocks of existential quantifiers) together.

In [4], Boyer and Moore developed a logic of bounded quantifiers. The above method for handling bounded quantifiers is closely related to their work.

## 4 Proving Integrity Constraints in *RRL*

Reasoning about integrity constraints typically involves proving properties by induction over the set of reachable states and the set of possible actions as illustrated in section 2.1. Induction is mechanized in *RRL* using the the *cover set* method proposed in [15] (see [6] for further details).

For doing proofs by induction, *RRL* orients definitions and lemmas into terminating rewrite rules using the algorithms implemented in *RRL* for orienting equations into terminating rewrite rules. The main inference method is that of *contextual rewriting* an equation or a conditional equation by terminating rewrite rules under assumptions, called *context*. Decision procedures for equality on ground terms, propositional calculus and linear arithmetic (quantifier-free theory of natural numbers) are automatically invoked while performing contextual rewriting. In addition, case analysis is performed.

In the cover set method, given a complete definition of a function symbol `f` on a data structure as a finite set of terminating rewrite rules, in which the left-hand side of each rule is of the form $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_k)$, the definition is used to generate an induction scheme on the data structure. The left-hand sides of the defining rules are used to generate all the subgoals of a conjecture to be proved, and the recursive calls to `f` on the right-hand sides of the rules are used to generate induction hypotheses.

### 4.1 An Automatically Generated Proof in *RRL*

The integrity constraint *IC* stating that the fluent *classified* is functional can be formulated in *RRL* as:

```
(xisbn1 = xisbn2) if classified(xisbn1, xid, xs) and
   classified(xisbn2, xid, xs) and reach(xs).
```

The above formula was attempted in *RRL* by induction. There are three possible candidate induction schemes corresponding to the subterms `classified(xisbn1, xid, xs)`, `classified(xisbn2, xid, xs)` and `reach(xs)` in the conjecture, based on the cover sets generated from the definitions of the predicates *classified* and *reach*, respectively. The schemes suggested by the first two of these terms are identical since the only argument being recursed in both cases is `xs`. These schemes *subsume* the scheme obtained from the subterm `reach(xs)` since the induction cases of the latter scheme are refined by those of the former. The most appropriate induction scheme is therefore the scheme suggested by either of the *classified* predicates in the above formula.

These analyses and heuristics are built into *RRL* and are performed automatically without any user guidance. Here is a part of the *RRL* transcript.

```
Let P(XS): XISBN1 == XISBN2 if REACH(XS) and CLASSIFIED(XISBN1,XID,XS)
  and CLASSIFIED(XISBN2,XID,XS)
Induction will be done on XS in CLASSIFIED(XISBN1, XID, XS), by scheme:
[1] P(S0)          [2] P(DO(CLASSIFYBOOK(XISBN1, XID1), XS)) if {P(XS)}
[3] P(DO(DELETEBOOK(XID1), XS)) if {P(XS)}
```

Subgoal `[1]` corresponds to the base case and follows trivially by case analysis using the definition of the fluent *classified*. Subgoal `[2]` corresponds to an induction step and establishes that the *IC* holds in a successor state obtained by a *classifybook* action if the *IC* holds in the previous state.

The induction conclusion and the hypothesis are both clauses of four literals each. The combination of the conclusion with each of the literals of the hypothesis leads to four subgoals. Two of these subgoals trivially reduce to true because of contradictory assumptions.

The remaining two subgoals are more complex and use the following intermediate lemma for proofs:

$$\mathtt{L1} : \mathtt{classified}(\mathtt{xisbn},\mathtt{xid},\mathtt{xs}) == \mathtt{false\ if\ ball}(\mathtt{xid},\mathtt{xs},\mathtt{dom}(\mathtt{xs})).$$

The reader may recall that the above is a bridge lemma, connecting the meaning of the new predicate `ball` introduced to eliminate the universal quantifier in the possibility axiom of `classifyBook`. This lemma can be used as a rewrite rule to eliminate the occurrence of `ball` and replace it by `classified`. In the first subgoal, the assumption `classified(xisbn11, xid, do(classifybook(xisbn1, xid1), xs))` along with the assumption `(not(classified(xisbn11, xid, xs)))` by the definition of `classified` implies that `xisbn1 = xisbn11` and `xid = xid1`.

Similarly, the assumption `classified(xisbn2, xid, do(classifybook(xisbn1, xid1), xs))` leads to two cases on simplification using the definition of `classified`. In the first case, `xisbn2 = xisbn1` and therefore, `xisbn11 = xisbn2` as desired. Otherwise, the literal `classified(xisbn2, xid, xs)` is true. This is contradictory to the assumption `reach(do(classifybook(xisbn1, xid1), xs))` that simplifies by the definition of the *reach* and *poss* to `ball(xid1, xs, doms(xs))` which by the lemma $L1$ implies that `not(classified(xisbn2, xid, xs))` since `xid1 = xid`.

The second subgoal is symmetric to the first subgoal, and is established in a similar fashion.

The subgoal [3], corresponds to an induction step that establishes the *IC* in a successor state obtained by a *deletebook* action from the *IC* in the previous state. The proof generated in *RRL* is similar to that of the subgoal [2] leading to four intermediate subgoals.

*Comparing hand proof with an automatically generated proof* : The reader would recall that the hand proof of the integrity constraint *IC* given in subsection 2.1 proceeds by induction on states. The basis case (step 1 of the hand proof) establishes the *IC* over the initial state $S_0$. For the induction step case (step 2 of the hand proof), case analysis on the actions is performed leading to two subgoals (steps 3 and 4 of the hand proof).

In the automated proof, the cover set method chooses the induction scheme based on the definition of `classified` as the most appropriate scheme. This scheme combines the two inductions (induction over reachable states and case analysis over possible actions performed one after another in the hand proof) into a single induction since the definition of the fluent `classified` is generated from its successor state axiom. The subgoal [1] in the automated proof corresponds to the step 1 of the hand proof, and the subgoals [2] and [3] correspond to the steps 3 and 4 of the hand proof respectively. The machine proof generated by *RRL* thus has the same top-level structure as the hand proof.

The subgoal [1] is established in *RRL* by case analysis on the definition of the fluent *classified* which ensures that *IC* holds at the initial state $S_0$ as assumed in the step 1 of the hand proof. The four intermediate subgoals generated from the subgoal [2] are essentially the same as the four cases described in the steps $3(a)$-$(c)$ of the hand proof.

In step $3(c)$ of the hand proof, the combinations imply that $\neg \exists$ $isbn'$ $classified(isbn', id, s)$, and $classified(isbn2, id, s)$, a contradiction. In the automated proof, this contradiction is obtained in the subgoals by using the bridge lemma $L1$ that relates the *ball* and the *classified* predicates.

*Proving the intermediate lemma*: Bridge lemma $L1$ is crucial in the proof of the *IC* above. Recall that $ball(xid, xs, dom(xs))$ represents $\forall$ $xisbn$ $\neg$ $classified(xisbn, xid, xs)$. This lemma is similar to the logical axiom for universal quantifiers in Hilbert style deductive systems for first order logic: $(\forall x)\phi(x) \supset \phi(y)$. As stated earlier, such bridge lemmas can be easily generated from the definitions of new predicates introduced to eliminate bounded quantifiers. So a user does not have to explicitly specify them. The proof of $L1$ is done by induction in *RRL*. The subterm `classified(xisbn, xid, xs)` suggests an induction scheme based on the cover set from `classified`. Here is a part of the *RRL* transcript.

```
Let P(XS): CLASSIFIED(XISBN, XID, XS) == FALSE if BALL(XID, XS, DOM(XS))
Induction will be done on XS in DOM(XS), by scheme:
[1] P(S0)                 [2] P(DO(CLASSIFYBOOK(XISBN,XID),XS)) if {P(XS)}
[3] P(DO(DELETEBOOK(XID), XS)) if {P(XS)}
```

The basis case [1] trivially follows from case-analysis on definition of `classified`. The subgoal [2] is split by *RRL* into two subgoals based on the combination of the clauses as described earlier. One of these subgoals follows by simplification from the definitions of `classified, ball` and `dom`. The other subgoal,

```
[2.1] not(CLASSIFIED(XISBN1, XID1, DO(CLASSIFYBOOK(XISBN, XID), XS))) if
BALL(XID1, DO(CLASSIFYBOOK(XISBN, XID), XS),
DOM(DO(CLASSIFYBOOK(XISBN, XID), XS))) and (not(BALL(XID1, XS, DOM(XS))))
```

is established with the help of the following intermediate lemma,

```
L2: ball(xid, do(classifybook(xisbn, xid1), xs), dom(xs)) ==
    ball(xid, xs, dom(xs)) if not(xid = xid1).
```

One of the assumptions `ball(xid1, do(classifybook(xisbn, xid), xs), dom(do(classifybook(xisbn, xid), xs)))` is simplified by the definitions of `dom` and `ball` to `ball(xid1, do(classifybook(xisbn, xid), xs), dom(xs))`. And, this contradicts the assumption `(not(ball(xid1, xs, dom(xs))))` by lemma $L2$.

The subgoal `[3]` is similarly split into two intermediate subgoals which follow from the following intermediate lemma $L3$ and the definitions.

```
L3: ball(xid, do(deletebook(xid1), xs), dom(xs))  ==
    ball(xid,  xs, dom(xs)) if not(xid = xid1).
```

*Generalization heuristic* : The automated proofs of the intermediate lemmas $L2$ and $L3$ illustrate yet another interesting feature of our theorem prover *RRL* viz. *generalization*. It is well known that in many cases, it is much easier to prove a stronger version of a given property by induction than the property itself. Stronger versions of lemmas are automatically generated by *RRL* using the generalization heuristic. A non-variable subterm appearing in both sides of an equation (or condition in case of a conditional equation) is generalized to a new variable; conditions on variables appearing in the subterm are dropped.

Given lemma $L3$, *RRL* does not attempt an inductive proof of $L3$, but the generalization heuristic instead generates:

$$\mathtt{ball(xid, do(deletebook(xid1, xs), xdom) == ball(xid, xs, xdom) \ if \ not(xid = xid1)},$$

by abstracting the subterm $dom(xs)$ in $L3$ to be an arbitrary domain variable $xdom$. If the more general lemma can be proved, then the proof of $L3$ follows by rewriting. The lemma $L2$ is similarly established in *RRL* by inductively proving its generalized version (obtained by abstracting $dom(xs)$ by a domain variable $xdom$) using the cover set based on the definition of *ball*.

## 5 Concluding Remarks

It is shown how Reiter's method for proving integrity constraints of deductive data bases can be mechanized using *RRL*, an automated theorem prover for induction. Using this methodology, we have been able to prove many simple integrity constraints. Mechanical proofs generated by *RRL* closely resemble manual proofs resulting from Reiter's approach. In this sense, the proposed methodology implements Reiter's approach using an automated induction theorem prover.

We have discussed how Reiter's approach can be formalized equationally using abstract data types in a quantifier-free calculus, and Reiter-style specifications using quantifiers can be mechanically translated into an equational quantifier-free calculus by introducing new bounded quantifier predicates. For these new predicates, bridge lemmas connecting their semantics to the fluents and related predicates in the original specification are needed. These bridge lemmas can be automatically synthesized and proved in *RRL*.

We believe that other reasoning capabilities of *RRL* such as a decision procedure for Presburger arithmetic to reason about numbers would be useful in proving general database integrity constraints involving numbers and arithmetic operations such as $+, -$ and relations such as $>, \geq, <, \leq$.

Our methodology has been applied in the proofs of some ICs, taking only a couple of minutes for each of them. It must be noticed that the proof of the integrity constraints is done only once in the lifetime of the database. Therefore, it is not a big problem for the system to take a few minutes in the proof of an integrity constraint, because once the proof is done, the user knows that his specification is correct and nothing has to be done afterwards in real time situations.

Sheard and Stemple [13] developed an approach for showing the safety of transactions with respect to a given set of integrity constraints, i.e., none of the transactions violate any of the integrity constraints. They proposed a database specification language and described a system based on Boyer and Moore's theorem prover for specifying and reasoning about safety of transactions. Bounded quantification over finite sets and tables in terms of primitives `for-some` and `for-all` was used to check if some (or all) objects in a table or a set satisfy a given property. However, their approach supports only a restricted set of transactions. It does not support Reiter's successor state axioms.

In contrast, we prove a much stronger claim for ensuring data base integrity, particularly that the database can never evolve into a state in which the integrity constraints are violated if the initial state satisfies the integrity constraints and the preconditions for the transactions are met. The safety of the individual transactions is a consequence of such a proof. Further, using our approach, we can reason about any action whose successor state axiom could be specified in terms of the general template given in section 2.

## References


1. L. Bertossi and J. Ferretti. SCDBR: A Reasoner for Specifications in the Situation Calculus of Database Updates. In *Temporal Logic. Proc. First Intl. Conf. ICTL '94, Bonn, Germany, July 1994, LNAI 827*, 543–545, Springer.
2. L. Bertossi, J. Pinto, P. Saez, D. Kapur, and M. Subramaniam. Automated Proofs of Integrity Constraints in Situation Calculus. Technical Report, Computer Science Dept., SUNY, Albany, Nov. 1995.
3. R.S. Boyer and J S. Moore. *A Computational Logic Handbook*. AP, 1988.
4. R.S. Boyer and J S. Moore. The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover. *Journal of Automated Reasoning*, 4:117–172, 1988.
5. D. Kapur. *Constructors can be Partial too.* Dept. of Computer Science, State University of New York at Albany, 1994.
6. D. Kapur and H. Zhang. An Overview of Rewrite Rule Laboratory (RRL). *J. of Computer and Mathematics with Applications*, 1995.
7. J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, eds, *Machine Intelligence*, vol. 4, 463–502, Edinburgh, Scotland, 1969. Edinburgh University Press.
8. R. Reiter. Towards a Logical Reconstruction of Relational Databases Theory. In J. Mylopoulos and J. Schmidt, eds, *On Conceptual Modeling: Perspectives from AI, Databases and Programming Languages*, 191–233. Springer-Verlag, 1984.
9. R. Reiter. The Frame Problem in the Situation Calculus: a Simple Solution (sometimes) and a Completeness Result for Goal Regression. In V. Lifschitz, ed, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, 359–380. Academic Press, 1991.
10. R. Reiter. Formalizing Database Evolution in the Situation Calculus. In *Proceedings of the Fifth Generation Computer Systems*, Tokyo, Japan, June 1992.
11. R. Reiter. On Specifying Database Updates. Technical Report KRR-TR-92-3, University of Toronto, Department of Computer Science, Toronto, Canada, 1992.
12. R. Reiter. Proving Properties of States in the Situation Calculus. *Artificial Intelligence*, 64(2):337–351, 1993.
13. T. Sheard and D. Stemple. Automatic Verification of Database Transaction Safety. TR 88-29, Dept. Computer and Information Science, U. Mass., Amherst, MA, 1988.
14. H. Zhang and D. Kapur. First–Order Theorem Proving using Conditional Rewrite Rules. In Lusk and Overbeek, eds., *Proc. 9th Intl. Conf. on Automated Deduction (CADE-9), LNCS 310*, 1–20. Springer, 1988.
15. H. Zhang, D. Kapur, and M.S. Krishnamoorthy. A Mechanizable Induction Principle for Equational Specifications. In Lusk and Overbeek, eds, *Proc. 9th Intl. Conf. on Automated Deduction (CADE-9), LNCS 310*, 162–181. Springer, 1988.


This article was processed using the LaTeX macro package with LLNCS style