, , 1–31 ()

SCDBR: An Automated Reasoner for Specifications of Database Updates

LEOPOLDO BERTOSSI bertossi@ing.puc.cl
MARCELO ARENAS marenas@ing.puc.cl
CRISTIAN FERRETTI cfs@ing.puc.cl
Pontificia Universidad Católica de Chile, Escuela de Ingeniería, Departamento de Ciencia de la Computación, Casilla 306, Santiago 22, Chile.

Abstract. In this paper we describe SCDBR, a system that is able to reason automatically from specifications of database updates written in the situation calculus, a first–order language originally proposed by John McCarthy for reasoning about actions and change. The specifications handled by the system are written in the formalism proposed by Ray Reiter for solving the frame problem that appears when one expresses the effects on the database predicates of the execution of atomic transactions. SCDBR is written in PROLOG, and can solve several reasoning tasks, among others, it is able to derive the final specification from effect axioms, to answer queries to virtually updated databases, to check legality of transactions, to prove integrity constraints from the specification, to modify the specification in order to embed a desired integrity constraint, and to answer historical queries. For some of these tasks SCDBR can call other systems, like relational database systems, automated theorem provers, and constraint solvers.

Keywords: Specifications of Database Updates, Knowledge Representation, Automated Reasoning, Integrity Constraints, Situation Calculus.

1. Introduction

We seek to apply logic programming and automated reasoning techniques, in particular, theorem proving, for: (1) reasoning with the formal specification of the dynamics of a database, and (2) reasoning with knowledge stored in the database. The specification and the database content are given in first–order logic plus a small second–order extension that allows reasoning by induction, what is needed for proving database *integrity constraints* (ICs). To be more precise, in this paper we describe, SCDBR [1] (for *situation-calculus-based database reasoner*), an automated reasoning system we have developed in the last three years that has such reasoning capabilities. A preliminary report on SCDBR appeared in (Bertossi & Ferretti, 1994) (see also (Bertossi et al., 1996a)).

In general, theories of change, such as the specification of the evolution of a database, include sentences that state the changes that are introduced when transactions are executed. In most cases, it is desirable to omit the sentences that describe the properties of the world that remain unchanged as transactions are performed. However, from a logical point of view, non change must be derivable from the logical specification. The problem of inferring non-change from a logical specification of change is referred to as the *frame problem* and has received a great deal of attention in the knowledge representation community (in fact, this problem has motivated much of the research on non-monotonic formalisms).

In this work we start from Reiter's solution to the frame problem (Reiter, 1991), that transforms preliminary specifications about evolving worlds into monotonic specifications in the situation calculus (McCarthy & Hayes, 1969)[2]. In Reiter's framework, the preliminary specifications embody implicit common-sense assumptions about the possible explanations for changes in truth values of fluents[3]. Once we have the transformed specifications, we may do reasoning from them in first–order logic.

According to Reiter (Reiter, 1992, Reiter, 1995), the preliminary specification of the transactions acting on a database includes details regarding their preconditions and their effects on the world. Furthermore, the specification might consider a set of ICs, which describe certain conditions that the database must always satisfy. For instance, in a personnel system, we may require that no clerk be associated with two different departments. Thus, the specification of a database system is composed of logical sentences that represent: (1) the initial state of the database; (2) the possible transactions (including preconditions and effects); and possibly, (3) integrity constraints. This preliminary specification is transformed into a new specification as was mentioned before. This transformation process generates the so–called *successor state axioms*[4].

Our system, SCDBR automatically derives the final database specification of the database dynamics, given in terms of successor state axioms, from a preliminary specification given in terms of effect axioms. The specifications handled by the system are fully first–order. SCDBR is also able to perform other reasoning tasks like: (1) Deciding about the legality of sequences of transactions of the form $do(\alpha_1, do(\alpha_2, do(\alpha_3, \ldots) \ldots)))$; (2) "Regressing" queries posed to a virtually updated database to a query about the initial database state. This task could be also seen as a form of hypothetical reasoning; (3) Interfacing with theorem provers and database systems like: ORACLE, OTTER (McCune, 1994), PROLOG, RRL (Kapur & Zhang, 1995), CLP(FD) (Codognet & Diaz, 1996, Diaz, 1994). Among other things, those systems can be used for proving statements about the initial database; (4) Answering historical queries, when the specification plus a sequence of transactions (executed or to be executed) is given (following the procedure described in (Siu & Bertossi, 1996b, Siu & Bertossi, 1996a, Siu et al., 1996), (Siu & Bertossi, 1996c, Zakinthinos, 1993)); (5) Mechanically modifying the initial specification in order to subsume in the final specification a given integrity constraint (in some cases, this is possible (Lin & Reiter, 1994b, Pinto, 1994)); (6) Physical update of the database from a virtual update (Lin & Reiter, 1997) (see also (Lin & Reiter, 1994a, Lin & Reiter, 1995)) (i.e. from the initial database, the successor state axioms and a given sequence of transactions);(7) Automated proofs of integrity constraints. For this we interfaced the system with OTTER, and RRL, which turned out to be very useful for generating appropriate induction schemas (Bertossi et al., 1996b).

The system is written in SICSTUS PROLOG, which is used mainly as a programming language; and can be run on the top of a database management system, like ORACLE, adding new and important functionalities to traditional DBMS.

2. The Situation Calculus

Characteristic ingredients for a situation calculus language $\mathcal{L}$, besides the usual symbols of first–order languages for domain descriptions, are: (1) Sorts *action*, *state*, and *object* (this last one for the objects in the domain; this sort could be split into sub–sorts if necessary); (2) Predicate symbols of the sort $(object, \ldots, object, state)$ to denote fluents. These are predicates that depend on the state of the world; (3) Operation symbols of the sort $(object, \ldots, object) \rightarrow action$ for denoting actions with objects as parameters (or applied to objects); (4) A constant S_0 to denote the initial state; (5) An operation symbol *do* of sort $(action, state) \rightarrow state$, that executes an action at a given state producing a successor state.

In this first–order language there are variables for individuals of each sort, so it is possible to quantify over objects, actions, and situations. We usually denote variables for actions with a, $\alpha, \ldots$, and variables for states, with s, $\sigma, \ldots$.

3. Reiter's Solution to the Frame Problem and Databases

The specification of a dynamically changing world, by means of an appropriate language of the situation calculus, consists in stating the laws of evolution of the world. This is typically done by specifying: (1) Fixed, state independent, but domain dependent knowledge about the objects of the world; (2) Knowledge about the state of the world at the initial situation S_0, that is given in terms of formulas that do not mention any state besides S_0; (3) Preconditions for performing the different actions (or making their execution possible). We introduce a predicate *Poss* in $\mathcal{L}$ of sort $(action, state)$ to say that the execution of an action is possible in a state; (4) The immediate (positive or negative) effects of actions in terms of the fluents whose truth values we know are changed by their execution.

In Reiter's formalism, the knowledge contained in items (1) and (2) above is considered the initial database D_{S_0}. The information given in item (3) is formalized by means of a precondition axiom of the form $Poss(A(\vec{x}), s) \equiv \pi_A(\vec{x}, s)$, for each action name A. Finally, item (4) is expressed by effect axioms for pairs (transaction, fluent).

Positive Effects Axioms: For some pairs formed by a fluent F and an action name A, an axiom of the form:

$$Poss(A(\vec{x}), s) \wedge \varepsilon_F^+(\vec{x}, \vec{y}, s) \supset F(\vec{y}, do(A(\vec{x}), s)). \tag{1}$$

Intuitively, if action A is possible, and the preconditions for fluent F represented by the meta–formula $\varepsilon_F^+(\vec{x}, \vec{y}, s)$ is true at state s, then fluent F becomes true at the successor state $do(A(\vec{x}), s)$ obtained after execution of action A at state s. Here, $\vec{x}, \vec{y}$ are parameters for the fluent and action.

Negative Effects Axioms: For some pairs formed by a fluent F and an action name A, an axiom of the form:

$$Poss(A(\vec{x}), s) \wedge \varepsilon_F^-(\vec{x}, \vec{y}, s) \supset \neg F(\vec{y}, do(A(\vec{x}), s)). \tag{2}$$

This is the case where action A makes fluent F to become false in the successor state.

Example: Let us consider a library database. For simplicity, every book that can be ordered or appears in the initial database will be assumed to be included in a constant table, $BooksInPrint(isbn, title, author, editor, year, edition)$, which may be interpreted as a predicate. There are classified and unclassified books. So, we have fluents $Classified(isbn, id, s)$, $Unclassified(isbn, copies, s)$. Classified books have an *id* number assigned which includes the ISBN number of the book and the copy number (so that different copies of the same book have different *id* numbers). The unclassified books are those that were ordered, but have not yet been classified; there may be several exemplars of the same book waiting to be classified. There is a fluent which records the number of exemplars available of each book that is classified, $Stock(isbn, quantity, s)$. Finally, we also have fluents $SoldOut(isbn, s)$ and $LostBook(id, s)$, with obvious meaning.

There are update actions (atomic transactions): (1) $deleteBook(id)$, that decreases *Stock* by one and only when the quantity is zero it deletes the book from the *Classified*, *Stock* and *LostBook* tables; (2) $classifyBook(isbn, id)$, which assigns an *id* to a book in the *Unclassified* table, inserts it in the *Classified*, removes the book from *Unclassified*, and it also updates the *Stock* according to the number of *copies*; (3) $order(isbn, copies)$, which adds an item that is not *SoldOut* in *BooksInPrint* to the *Unclassified*.

Initial Database [5]**:**

- $BooksInPrint('3-540-18199-7', 'Foundations\ of\ Logic\ Programming', 'Lloyd,\ J.W.', 'Springer', '1987', '2')$
- $BooksInPrint('0-7167-8162-X', 'Database\ and\ Knowledge-Base\ Systems', 'Ullman,\ J.D.', 'Computer\ Science\ Press', '1989', '1')$
- $BooksInPrint('0-412-14930-3', 'The\ Theory\ of\ Computer\ Science', 'Brady,\ J.M.', 'A\ Halsted\ Press\ Book', '1977', '1')$
- $Classified(isbn, id, S_0) \equiv (isbn = '0-412-14930-3' \wedge id = '10') \vee (isbn = '0-412-14930-3' \wedge id = '11') \vee (isbn = '0-7167-8162-X' \wedge id = '12')$
- $Stock(isbn, int, S_0) \equiv (isbn = '0-412-14930-3' \wedge int = 2) \vee (isbn = '0-7167-8162-X' \wedge int = 1) \vee (isbn = '3-540-18199-7' \wedge int = 0)$
- $Unclassified(isbn, int, S_0) \equiv (isbn = '0-412-14930-3' \wedge int = 3) \vee (isbn = '0-7167-8162-X' \wedge int = 2) \vee$

$$(isbn =' 3{-}540{-}18199{-}7' \ \wedge \ int = 3)$$

Action Preconditions:

- $Poss(deleteBook(id), s) \equiv LostBook(id, s)$
- $Poss(order(isbn, copies), s) \equiv (\exists title, author, editor, year, edition)$
  $BooksInPrint(isbn, title, author, editor, year, edition) \wedge copies > 0$
- $Poss(classifyBook(isbn, id), s) \equiv (\exists copies)\, Unclassified(isbn, copies, s)$
  $\wedge \neg(\exists isbn')\, Classified(isbn', id, s)$

Effect Axioms:

- $Poss(deleteBook(id), s) \ \supset \ \neg LostBook(id, do(deleteBook(id), s))$
- $Poss(deleteBook(id), s) \ \supset \ \neg Classified(isbn, id, do(deleteBook(id), s))$
- $Poss(deleteBook(id), s) \ \wedge \ Classified(isbn, id, s) \ \wedge \ Stock(isbn, quantity, s) \ \supset$
  $\neg Stock(isbn, quantity, do(deleteBook(id), s))$
- $Poss(deleteBook(id), s) \ \wedge \ Classified(isbn, id, s) \ \wedge \ Stock(isbn, quantity, s) \ \wedge$
  $quantity > 1 \wedge quantity' = quantity - 1 \ \supset$
  $Stock(isbn, quantity', do(deleteBook(id), s))$
- $Poss(order(isbn, copies), s) \ \wedge \ \neg SoldOut(isbn, s) \ \wedge$
  $((Unclassified(isbn, copies', s) \ \wedge \ copies''' = copies' + copies) \vee$
  $(\neg\exists copies''\ Unclassified(isbn, copies'', s) \ \wedge \ copies''' = copies)) \ \supset$
  $Unclassified(isbn, copies''', do(order(isbn, copies), s))$
- $Poss(order(isbn, copies), s) \ \wedge \ \neg SoldOut(isbn, s) \ \wedge$
  $Unclassified(isbn, copies', s) \ \supset$
  $\neg Unclassified(isbn, copies', do(order(isbn, copies), s))$
- $Poss(classifyBook(isbn, id), s) \ \wedge \ Unclassified(isbn, copies, s) \ \wedge \ copies > 1 \ \wedge$
  $copies' = copies - 1 \ \supset$
  $Unclassified(isbn, copies', do(classifyBook(isbn, id), s)$
- $Poss(classifyBook(isbn, id), s) \ \wedge \ Unclassified(isbn, copies, s) \ \supset$
  $\neg Unclassified(isbn, copies, do(classifyBook(isbn, id), s))$
- $Poss(classifyBook(isbn, id), s) \ \wedge$
  $((Stock(isbn, quantity, s) \ \wedge \ quantity' = quantity + 1) \vee$
  $(\neg\exists quantity\ Stock(isbn, quantity, s) \ \wedge \ quantity' = 1)) \ \supset$
  $Stock(isbn, quantity', do(classifyBook(isbn, id), s))$
- $Poss(classifyBook(isbn, id), s) \ \wedge \ Stock(isbn, quantity, s) \ \supset$
  $\neg Stock(isbn, quantity, do(classifyBook(isbn, id), s))$
- $Poss(classifyBook(isbn, id), s) \ \supset \ Classified(isbn, id, do(classifyBook(isbn, id), s)$

$\square$

The specification given in the example does not mention the usually many things (fluents) that do not change when a specific action is executed. A solution to this problem, *the frame problem*, is necessary in order to count on the persistence of many properties after actions are performed. Many solutions have been given to the frame problem. Most of them have a procedural component (Fikes & Nilsson, 1971, Kowalski, 1979) or are non-monotonic (Reiter, 1987). See (Sandewall, 1994) for a systematic assessment of different approaches to the problem of formalizing knowledge about dynamically changing worlds with inertia. Nevertheless, in (Reiter, 1991), building on work by Haas (Hass, 1987), Pednault (Pednault, 1989) and Schubert (Schubert, 1990), Reiter presents a simple solution to the frame problem that that provides a first–order and monotonic specification. We sketch this solution in the rest of this section.

For example, if we have only two positive effects laws for fluent F: (1) and

$$Poss(A'(\vec{z}), s) \wedge \delta_F^+(\vec{z}, \vec{y}, s) \supset F(\vec{y}, do(A'(\vec{x}, s)), \tag{3}$$

we may combine them into one general positive effect axiom for fluent F:

$$\begin{aligned} &Poss(a, s) \wedge [\exists \vec{x}\ (a = A(\vec{x}) \wedge \varepsilon_F^+(\vec{x}, \vec{y}, s)) \vee \\ &\qquad \exists \vec{z}\ (a = A'(\vec{z}) \wedge \delta_F^+(\vec{z}, \vec{y}, s))] \supset F(\vec{y}, do(a, s)). \end{aligned} \tag{4}$$

In this form we obtain, for each fluent F, a general positive effect law of the form:

$$Poss(a, s) \wedge \gamma_F^+(a, s) \supset F(do(a, s)). \tag{5}$$

In similar form, we obtain, for each fluent F, a general negative effect axiom of the form:

$$Poss(a, s) \wedge \gamma_F^-(a, s) \supset \neg F(do(a, s)). \tag{6}$$

For example, if we consider the fluent *Unclassified*, in our library database, we obtain the following general effect axioms:

- $Poss(a, s) \wedge [\exists copies\ (a = order(isbn, copies) \wedge \neg SoldOut(isbn, s) \wedge (\exists quantity'\ (Unclassified(isbn, quantity', s) \wedge quantity = quantity' + copies) \vee \neg \exists quantity'\ Unclassified(isbn, quantity', s) \wedge quantity = copies))) \vee \exists id\ (a = classifyBook(isbn, id) \wedge \exists quantity'\ (Unclassified(isbn, quantity', s) \wedge quantity' > 1 \wedge quantity = quantity' - 1))] \supset Unclassified(isbn, quantity, do(a, s))$
- $Poss(a, s) \wedge [\exists copies\ (a = order(isbn, copies) \wedge \neg SoldOut(isbn, s) \wedge Unclassified(isbn, quantity, s)) \vee \exists id\ (a = classifyBook(isbn, id) \wedge Unclassified(isbn, quantity, s))] \supset \neg Unclassified(isbn, quantity, do(a, s))$

The basic assumption underlying Reiter's solution is that the general effect axioms, both positive and negative, for a given fluent F, contain all the possibilities for fluent F to change its truth value from a state to a successor state. Actually, for each fluent F we generate its **Successor State Axiom** (SSA):

$$Poss(a, s) \supset [F(do(a, s)) \equiv \gamma_F^+(a, s) \vee (F(s) \wedge \neg \gamma_F^-(a, s))]. \tag{7}$$

This axiom is tacitly universally quantified over actions. Intuitively, if action a is possible, then fluent F is true at the successor state if and only if a is one of the actions leading to F's truth (and the preconditions on the fluent are true), or F was already true when the action was executed and a is not one of the actions that make F false (or the preconditions on the fluent are not true). For example, we obtain the following SSA for the fluent *Unclassified* in our library database:

- $Poss(a,s) \supset Unclassified(isbn, quantity, do(a,s)) \equiv$
$$[\exists copies\ (a = order(isbn, copies) \wedge \neg SoldOut(isbn,s) \wedge$$
$$(\exists quantity'\ (Unclassified(isbn, quantity', s) \wedge\ quantity = quantity' + copies) \vee$$
$$\neg\exists quantity'\ (Unclassified(isbn, quantity', s)\ \wedge quantity = copies)))) \vee$$
$$\exists id\ (a = classifyBook(isbn, id)\ \wedge\ \exists quantity'\ (Unclassified(isbn, quantity', s) \wedge$$
$$quantity' > 1\ \wedge\ quantity = quantity' - 1))] \vee$$
$$Unclassified(isbn, quantity, s)\ \wedge\ \neg[\exists copies\ (a = order(isbn, copies) \wedge$$
$$\neg SoldOut(isbn,s)\ \wedge\ Unclassified(isbn, quantity, s)) \vee$$
$$\exists id\ (a = classifyBook(isbn, id)\ \wedge\ Unclassified(isbn, quantity, s))]$$

In order for things to work properly, it is necessary to add unique names axioms for actions and states: (1) For distinct action names A and A', $A(\vec{x}) \neq A'(\vec{y})$; (2) For each action name A, $A(x_1,\ldots,x_n) = A(y_1,\ldots,y_n) \supset x_1 = y_1 \wedge \cdots \wedge x_n = y_n$; (3) $S_0 \neq do(a,s)$; (4) $do(a,s) = do(a',s') \supset a = a' \wedge s = s'$.

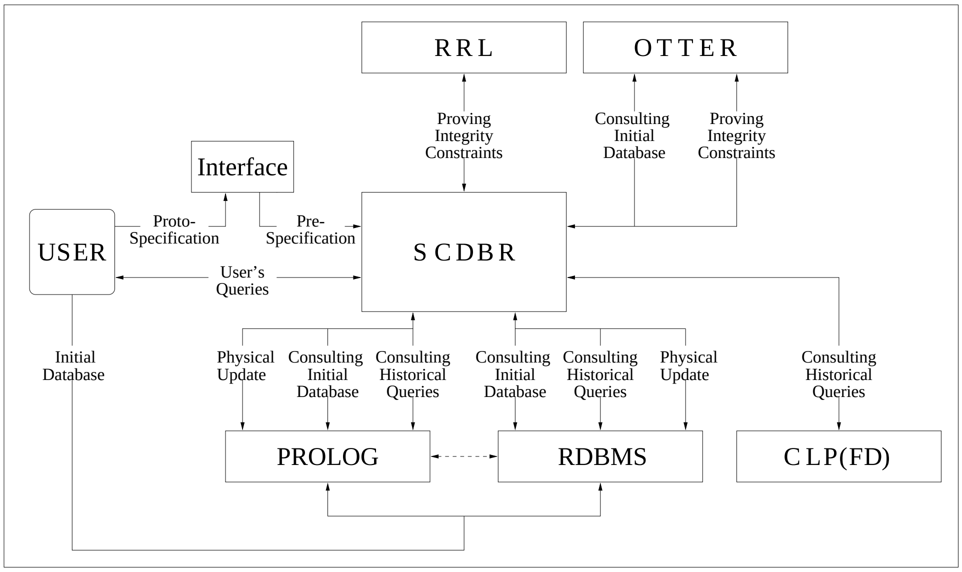

Figure 1. Interaction with SCDBR.

4. An Overview of SCDBR

Our system takes database specifications, given in terms of effect axioms, in a convenient format (proto-specification) by means of an external interface, which translates this information into SDCBR notation (pre-specification). Next, the system automatically derives the final database specification given in terms of successor state axioms according to Reiter's solution. Then the system can take advantage of the generated specifications for performing different reasoning tasks, for example:

1. In order to reason about the initial database, SCDBR can follow two different approaches: (1) If the initial database is relational, it uses SQL or PROLOG as query languages; (2) If the initial database is first–order, it uses OTTER as theorem prover.

2. Temporal projection: Given the initial database and a query referred to a (future) state, $do(A_n, do(A_{n-1} \ldots do(\ A_1, S_0) \ldots))$, SCDBR is able to generate a new equivalent query about the initial state, that is, the original query is *regressed* to the initial state. The *legality* of the list of transactions $A_1, \ldots A_n$ can be previously checked by SCDBR.

3. Given a transaction α, SCDBR is able to physically update the database, incorporating the changes due to the execution of α.

4. SCDBR can answer historical queries of the kind *"Has a given property been true in all states of the database"* or *"Has a given property been true in some states of the database"*, etc...

5. SCDBR can handle Integrity Constraints (IC) in two ways: (1) Proving, from the specification, that a given IC holds in all (legal) states of a database. (2) If an IC is not entailed by the specification, in some cases, SCDBR modifies the initial specification in order to subsume the IC.

For solving these tasks, the system is able to call relational database management systems and theorem provers. Figure 1 shows the interaction with SCDBR.

5. The Database Specification in SCDBR

5.1. Information Provided by the User

The user's proto specification, given to the system by means of an external interface, is the following:

1. The similarity type of the first–order language, in particular, names for fluents, predicates, actions and distinguished individuals.

2. For each action a definition of the form:

ACTION	*Action*
PRECONDITION	*ActionPrecondition*
EFFECTS	$Precondition_1$: $EffectPolarity_1$ $Fluent_1$
	...
	$Precondition_n$: $EffectPolarity_n$ $Fluent_n$

The first part of this declaration tells us that the *ActionPrecondition* needs to hold when the *Action* is to be executed, the second part establishes that *Action* affects either positively (+) or negatively (-) the $Fluent_i$ (at the successor state) if the $Precondition_i$ on the fluent holds at the current state, for each $1 \leq i \leq n$. So, the *EffectPolarity* is either + or -.

3. A set of formulas about the initial state (the initial database).

The external interface translates this information into SCDBR notation (logical formulas in our system are written in prefix notation). In particular, it converts the knowledge contained in item 2. into the following sets:

1. A set of binary tuples (*Action*, *ActionPrecondition*).

2. An incidence table, more precisely, a set of tuples of the form

$$
\begin{array}{l}
(\ [\ Action, \\
\quad [\ [\ Fluent_1,\ EffectPolarity_1\ ],\ Precondition_1\ ] \\
\quad \ldots, \\
\quad [\ [\ Fluent_n,\ EffectPolarity_n\ ],\ Precondition_n\ ])
\end{array}
$$

5.2. Representing and Processing the Specification Language

The formulas of language $\mathcal{L}$ are internally represented by PROLOG ground terms written in prefix notation, and are processed by PROLOG procedures. In consequence, the system uses two kinds of variables, on one side, the usual PROLOG variables (starting with upper-case letters), and variables of the language $\mathcal{L}$, starting with lower–case letters, that PROLOG considers as constants.

For a better readability of formulas, the system contains procedures for translating formulas in prefix notation into infix notation, `p_i(·,·)`, and in the other direction with `i_p(·,·)`.

5.3. Generating of the Final Specification Axioms

In order to generate the final specification starting from the information provided by the user, the system relies on unification. For example, for each fluent F, we know the general syntactic form, (7), of its SSA. This structure is represented by the PROLOG procedure (notice that the formula in this procedure is written in prefix notation):

```
ssa( F, all( a, implies( poss( a, STATE_VAR ),
            iff( FSUC, or( PT, and( FACT, not( NT ))))))) :- ...        (8)
```

Here, the PROLOG variables `F`, `FSUC`, `PT`, `FACT`, `NT` stand for the fluent, the fluent at the successor state, the positive gamma formula for F, the fluent at the current state, and the negative gamma formula for F, respectively. The procedure is used as follows: given a fluent name, we ask for its successor state axiom `A`, by posing the query `| ?- ssa(fluent-name, A)`. By unification, `A` will turn out to be a PROLOG term representing a formula, with concrete values for `FSUC`, `PT`, `FACT`, `NT`, obtained from the body of clause (8), using other procedures that collect and process the information provided by the user.

In the following subsections we present the PROLOG predicates that allow to generate the final specification axioms.

5.3.1. Computing Unique Names Axioms

The `una` predicate generates the unique names axioms (UNAs) for actions and states.

Example: Consider the library specification in section 3. The UNAs are obtained executing `| ?- una(U),p_i(U,UI)`. In this case, we are using the predicate `p_i` for translating the UNAs stored in `U` into infix notation. A part of the list is:

```
UI = [neg delete_book(id1) eq order(isbn1,int1),...,order(isbn1,int1)
      eq order(isbn2,int2) => isbn1 eq isbn2 & int1 eq int2,...,do(a,
      s) eq do(a1,s1) => a eq a1 & s eq s1]
```

This output corresponds to formulas:

- $\neg deleteBook(id_1) = order(isbn_1, int_1)$
- $order(isbn_1, int_1) = order(isbn_2, int_2) \supset isbn_1 = isbn_2 \wedge int_1 = int_2$
- $do(a, s) = do(a_1, s_1) \supset a = a_1 \wedge s = s_1$.

5.3.2. Computing Successor State Axioms

The predicate `ssa` constructs the SSAs as described in section 3 .

Example: Let us ask the system to compute the SSA for the fluent *LostBook* in the library database.

```
| ?- ssa(lost_book,F),p_i(F,R).

R = forall(a):(poss(a,s) => (lost_book(id1,do(a,s)) <=> lost_book(id1,
    s) & neg a eq delete_book(id1)))
```

The answer is stored in the variable `R`, and represents the following axiom:

$$Poss(a,s) \supset LostBook(id_1, do(a,s)) \equiv LostBook(id_1, s) \wedge \neg a = deleteBook(id_1). \tag{9}$$

5.3.3. Computing Action Precondition Axioms

With the predicate `ap`, we can compute the action precondition axioms (APs).

Example: Now we ask the system to compute the AP for action *classifyBook*.

```
| ?- ap(classify_book,F),p_i(F,R).

R = forall(s) : (poss(classify_book(isbn1,id1), s) <=> exists(int1) :
    unclassified(isbn1,int1,s)& neg exists(isbn2) : classified(isbn2,
    id1,s))
```

The answer, stored in the variable `R`, represents the following axiom:

$$Poss(classifyBook(isbn, id), s) \equiv \exists copies\ Unclassified(isbn, copies, s) \wedge \neg \exists isbn_1\ Classified(isbn_1, id, s). \tag{10}$$

6. Some Basic SCDBR Procedures

In this section we introduce some fundamental SCDBR procedures that will be used for solving several reasoning tasks in section 7.

6.1. The Regression Operator

The *regression operator* (Reiter, 1991) applied to a formula evaluated at a successor state returns an equivalent formula evaluated at the current state. This is done by using the SSAs. More precisely, if fluent F, appearing in a formula φ, has the following SSA:

$$F(x_1, x_2, \ldots, x_n, do(a,s)) \equiv \Phi_F(x_1, x_2, \ldots, x_n, a, s), \tag{11}$$

where the formula $\Phi_F(x_1, x_2, \ldots, x_n, a, s)$ does not mention the state $do(a,s)$, the operator, $\mathcal{R}$, applied to φ, replaces each occurrence of a term $F(t_1, t_2, \ldots, t_n, do(\alpha, \delta))$ by $\Phi_F|^{x_1,x_2,\ldots,x_n,a,s}_{t_1,t_2,\ldots,t_n,\alpha,\delta}$. In the system, the regression operator is represented by the predicate `reg`.

Example: We can invoke the regression operator on the formula

$$LostBook('0{-}7167{-}8162{-}X', do(deleteBook('0{-}412{-}14930{-}3'), S_0)), \tag{12}$$

in which we find the fluent *LostBook* in a successor state.

```
| ?- reg(lost_book('0-7167-8162-X', do(delete_book('0-412-14930-3'),
     s0)),F),p_i(F,R).

R = lost_book('0-7167-8162-X',s0) & neg delete_book('0-412-14930-3')
    eq delete_book('0-7167-8162-X')
```

The result is stored in variable `R` and represents the following formula:

$$\begin{aligned} &\mathit{LostBook}('0\text{--}7167\text{--}8162\text{--}X', S_0) \wedge \\ &\quad \neg(\mathit{deleteBook}('0\text{--}412\text{--}14930\text{--}3') = \mathit{deleteBook}('0\text{--}7167\text{--}8162\text{--}X')), \end{aligned} \tag{13}$$

which corresponds to the right–hand side of axiom (9), but with parameters id_1, a and s replaced by $'0\text{--}7167\text{--}8162\text{--}X'$, $\mathit{deleteBook}('0\text{--}412\text{--}14930\text{--}3')$ and S_0, respectively.

6.2. The Pruning Operators

When regression is applied to a formula as in the previous section, it is usually the case that comparisons between actions and comparisons between individuals in the domain are generated. In order to handle these comparisons, SCDBR has three *pruning operators* that simplify formulas on the basis of the unique name axioms for actions, states and objects, obtaining logically equivalent formulas. They are $\mathcal{P}_{una}$, $\mathcal{P}_{uns}$ and $\mathcal{P}_{uno}$, respectively. For example, when the operator $\mathcal{P}_{una}$ is applied to a formula, it descends recursively until it finds the atomic formulas involving equalities. The important base case is a comparison between actions, in which case the operator is defined by:

$$\mathcal{P}_{una}[A(t_1, t_2, \ldots, t_n) = B(t'_1, t'_2, \ldots, t'_m)] := \begin{cases} \mathit{false} & \mathit{if}\ A \neq B \\ \bigwedge_{i=1}^{n} t_i = t'_i & \mathit{otherwise} \end{cases}$$

In other atomic cases, the operator does not do anything.

The operator $\mathcal{P}_{uns}$ has these relevant base cases: $\mathcal{P}_{uns}[S_0 = S_0] := \mathit{true}$, $\mathcal{P}_{uns}[S_0 = do(a, s)] := \mathit{false}$, $\mathcal{P}_{uns}[do(a_1, s_1) = do(a_2, s_2)] := a_1 = a_2 \wedge \mathcal{P}_{uns}[s_1 = s_2]$.

Usually, the order in which the operators are applied is the following: first, $\mathcal{P}_{uns}$ (if necessary), next, $\mathcal{P}_{una}$, because the first one may leave some comparisons between actions. Optionally, the $\mathcal{P}_{uno}$ operator can be applied if the assumption about unique names for individuals in the domain is made. As can be seen from the definitions, the resulting formula may contain the atoms *true* and *false*. In order to get a simpler formula, it is possible to call the predicate `csf`, that applies some logical rules of propositional and predicate calculus[6].

Example: We want to simplify the formula (13) that we obtained in the previous section by application of the regression operator.

```
| ?- i_p(lost_book('0-7167-8162-X',s0) & neg delete_book('0-412-14930
```

```
     -3') eq delete_book('0-7167-8162-X'),F),prune_una(F,A),prune_uno
     (A,B),p_i(A,A1),p_i(B,B1).

A1 = (lost_book('0-7167-8162-X',s0) & neg '0-412-14930-3' eq '0-7167-
     8162-X'),
B1 = lost_book('0-7167-8162-X',s0)
```

Notice that we first applied $\mathcal{P}_{una}$ with the SCDBR predicate `prune_una` to (13), obtaining as an intermediate result, contained in the PROLOG variable `A1`, the expression:

$$LostBook('0\text{–}7167\text{–}8162\text{–}X', S_0) \ \wedge \ \neg('0\text{–}412\text{–}14930\text{–}3' =' 0\text{–}7167\text{–}8162\text{–}X').$$

Finally, we applied $\mathcal{P}_{uno}$ with the SCDBR procedure `prune_uno` to this term. The final answer is contained in `B1`.

6.3. PROLOG as Query Language

The system is able to transform a set of first–order formulas into a PROLOG program. We will apply this functionality for solving several tasks. One of them consists in using PROLOG as a query language. This is useful for checking formulas against the initial database, what is needed in verification of legality of actions (section 7.1), proofs of integrity constraints (section 7.4.1), etc.

Typically we will have a relational database (at the initial state) plus a first–order query Q. The query is transformed into a PROLOG program, including a top goal, that is run against the relational database seen as a set of PROLOG facts. Our translation algorithm has two steps, a first one consisting essentially of Lloyd and Topor's algorithm for translating general programs into normal programs (Lloyd, 1987). In Lloyd and Topor's set up, a PROLOG program would be generated from a single clause $\leftarrow Q$, a goal clause. The second step reorders the generated subgoals in the resulting PROLOG rules in order to avoid floundering of queries in the presence of negation as failure when possible (Lloyd, 1987) and to handle arithmetical and comparison operators in subgoals.

The algorithm in the second step reorders the subgoals in the rule bodies with the following strategy: put first the positive literals that are not comparisons, second, the comparisons of the form $X = a$, third, comparisons of the form $X = Y$, fourth, equalities between arithmetical terms (using the `is` built-in predicate if necessary, as in $int_1 = int_2 + int_3$, that is transformed into `Int1 is Int2 + Int3`); next comparisons with order predicates, and finally, negative literals. For example, $\neg p(x) \wedge q(x)$ is transformed into `q(X),not p(X)`, avoiding floundering.

In order to obtain a correct answer with the whole translation and querying procedure, the first–order input query posed to the initial database must be *safe* (Ullman, 1988). If the formula is not safe, floundering queries in the presence of negation as failure may be generated. A syntactic subclass of safe formulas consists of the safe DRC formulas (Ullman, 1988). It can be proved that the program and the PROLOG query generated by the algorithm from the original first–order query will not flounder if this query is safe DRC (Saez, 1997). In this case, it is also possible to prove, using results appearing in (Lloyd, 1987, chap.4), that query Q has an answer θ wrt the relational database D iff the generated program and goal $P \cup \{G\}$ have θ as a computed answer (Saez, 1997).

6.3.1. Querying the Initial Database

The predicate `prolog_initial` is run by the system for answering first–order queries posed to an initial *relational* database (in section 6.4 we show how to call from SCDBR an automated theorem prover for checking queries against a first–order initial database). As described above, given the input formula, this procedure generates a PROLOG program plus an internal goal to be answered by the program.

Example: We wish the system to return a list of the identifiers *isbn* of those books that have at least one copy classified and one copy unclassified at the initial database. The corresponding first–order query is:

$$\begin{aligned}&\exists id\ \mathit{Classified}(isbn, id, S_0)\ \wedge \\ &\qquad\qquad \exists quantity\ (\mathit{Unclassified}(isbn, quantity, S_0)\ \wedge\ quantity > 0),\end{aligned} \tag{14}$$

where the variable *isbn* appears free, because it will be instantiated with the answer. The query is asked in this way:

```
| ?- i_p( exists(id1) : classified( isbn1, id1, s0 ) & exists(int1) :
     (unclassified(isbn1,int1,s0) & int1 > 0), F ),prolog_initial(F).
```

The predicate `prolog_initial/2` transforms first the formula (14) into the following PROLOG program:

```
pi_v1(Isbn1):-
        pi_v1_0_0(Isbn1),
        pi_v1_1_0(Isbn1).
pi_v1_0_0(Isbn1):-
        classified(Isbn1,Id1,s0).
pi_v1_1_0(Isbn1):-
        unclassified(Isbn1,Int1,s0),
        >(Int1,0).
```

After that, calling the top level predicate `pi_v1/1`, it gives as answer the lists of ISBNs:

```
[[isbn1,0-412-14930-3]]
[[isbn1,0-7167-8162-X]]
```

In this case, the values $'0-412-14930-3'$ and $'0-7167-8162-X'$ satisfy the formula (14). □

6.4. Using a Theorem Prover

In the previous section we showed how to use PROLOG as a query language for relational initial databases. In this section we will show how to call from SCDBR a more general automated theorem prover, for solving different tasks. This is particularly useful when we have a first–order initial database, and when we prove integrity constraints (see section 7.4.1). More specifically, we will call OTTER (McCune, 1994).

6.4.1. Verifying Formulas in the Initial Database

OTTER can be used to check the truth value of formulas of the form $\varphi(S_0)$, being S_0 the only state term in the formula. That is, we check the formula against the initial database, which in this case can be first–order. For example, we can ask SCDBR, to call OTTER to find out if there is a book for which there at least one copy classified at the initial state. The formula $\varphi(S_0)$ and the initial database, D_{S_0}, have to be translated into OTTER format, in particular, explicit unique names axioms for objects, predicate closure axioms can be automatically generated with SCDBR. The SCDBR predicate `otter_initial` calls the internal translation procedures and runs OTTER in order to answer the query.

Example: Let us assume that in our initial database in section 3, instead of having an explicit set of tuples in the *Unclassified* table, we now have the information that there is a positive quantity of unclassified books with *isbn* identifiers $'0{-}412{-}14930{-}3'$, but without knowing how many of them. So, we have the following sentence in the specification:

$$\exists quantity\ (Unclassified('0{-}412{-}14930{-}3', quantity, S_0)\ \wedge\ quantity > 0).$$

We want to know if there is a book with at least one classified copy and one unclassified copy at the initial state. This query is the first–order sentence

$$\begin{aligned}\exists isbn\ (\exists id\ Classified(isbn, id, S_0) \wedge \\ \exists quantity\ (Unclassified(isbn, quantity, S_0)\ \wedge\ quantity > 0)),\end{aligned} \tag{15}$$

which in SCDBR can be asked and positively answered in this way:

```
| ?-i_p(exists(isbn1) : (exists(id1) : classified( isbn1, id1, s0 ) &
    exists(int1) : ( unclassified( isbn1, int1, s0) & int1 > 0)), F),
    otter_initial(F).
Proof for the initial database:
------------------------------
invoking otter. I'll tell you when I'm done.
-------- PROOF --------
The proof can be found in temp/proof-s0
```

6.5. Connection of SCDBR to a RDBMS

We have seen in the previous section that SCDBR can be connected to a PROLOG database consisting of facts. Nevertheless, in applications it is more common to have data stored in a relational database management system (RDBMS). For this reason, we connected SCDBR to ORACLE. In this way, the initial database can be physically stored as a database in that system, and the first–order queries to the initial database can be translated into SQL queries to be posed by the RDBMS to the its stored database. Another uses of this facility are: (a) the possibility of having the facts for a PROLOG program stored as a relational database; (b) the possibility of updating tables and generating auxiliary tables in the RDBMS as an additional support to SCDBR for some of its reasoning tasks, e.g. answering historical queries, where small auxiliary tables containing data that is relevant to the query have to be created and updated as the query is processed (see section 7.5); (c) checking integrity constraints at the initial database; etc.

Making SCDBR interact with a RDBMS is much more efficient than translating the whole relational database into the language of other reasoner like PROLOG or OTTER. In this section we show the query translation mechanism, the connection of SCDBR to ORACLE, and we discuss the coupling of PROLOG to a RDBMS.

6.5.1. *Generating SQL queries*

Suppose that we have a relational initial database. We want to answer to a query $\varphi(S_0)$, written in first–order logic. For translating the query into SQL, SCDBR contains a predicate `fol2sql` that, in its turn, calls three other predicates: (1) `normalize`, that receives a formula $\varphi(S_0)$ and returns a logically equivalent first–order formula $\psi(S_0)$ in *or-free parts normal form*; (2) `alwd` that checks the resulting formula for tractability, a sufficient syntactical condition for safeness; and finally (3) `trans_query` that translates the formula into an SQL query if the formula is tractable. All these concepts and translation algorithms are taken from (M. Böhlen, 1994).

For example, since the query $\forall(isbn_1, id_1)(\mathit{Classified}(isbn_1, id_1, S_0))$ is not safe, we obtain

```
| ?- i_p(forall(isbn1): forall(id1) : (classified(isbn1,id1,s0)),F),
     fol2sql(F,FSQL).

formula is not allowed
```

Nevertheless, with the functional dependency (22), as a safe query, we obtain the normal form:

$$\neg\exists(isbn_1, isbn_2, id_1)(\mathit{Classified}(isbn_1, id_1, S_0)\wedge \mathit{Classified}(isbn_2, id_1, S_0) \wedge isbn_1 \neq isbn_2), \tag{16}$$

that can be translated into a SQL (predicate `fol2sql` suppresses state S_0 as a first step):

```
| ?-fol2sql(all(isbn1,all(isbn2,all(id1,implies(and(classified(isbn1,
    id1,s0),classified(isbn2, id1, s0)),equal(isbn1,isbn2))))),SQLF).

SELECT 1
FROM DUAL
WHERE NOT EXISTS (SELECT *
                  FROM classified a1,classified a0
                  WHERE a0.c1<>a1.c1 AND a1.c2=a0.c2)
```

A SCDBR predicate can transform this SQL query into a string that is apt to be given directly to ORACLE. For this query, ORACLE returns 1 if it is true, and no tuples if it is false[7].

An alternative to this methodology for checking safeness and translating first–order formulas into SQL code is to implement the verification and transformation mechanisms presented in (van Gelder & Topor, 1987) as done in the implementation described in (Chomicki & Toman, 1995).

6.5.2. Coupling SCDBR to a RDBMS

We implemented a basic interface between ORACLE and SICSTUS PROLOG with the purpose of asking SQL queries to ORACLE from SCDBR (this is not a *coupling* (Ceri et al., 1990) of PROLOG to ORACLE in the sense we will discuss in section 6.5.3). For achieving this, a couple of primitives, that use the ORACLE Call Interfaces Library, were written in C. After that, these primitives were declared as PROLOG predicates by means of the C interface provided by SICSTUS PROLOG. They are: (1) `ora_open_db`: this predicate allows the connection to ORACLE. If, for some reason, this is not possible, it prints an error message on screen and fails. This is example of call: `| ?- ora_open_db("scott/tiger").` (where `scott` and `tiger` are the account name and password, respectively); (2) `ora_close_db`: this predicate causes disconnection from the server. It is used this way: `| ?- ora_close_db.`; (3) `ora_sql(+QUERY,-RESULT)`: this is the central predicate that asks the query to the ORACLE server. `QUERY` is a string containing the query to be posed, whose result will be given in the list `RESULT`, that contains all the tuples that satisfy the query. If the query is a sentence, then `RESULT` is `true` or `false`. Finally, `RESULT` is an error message if the connection to the ORACLE server is interrupted.

The high level predicate `sql_initial` is responsible for accepting an first-order formula at S_0 as a query, translating it into SQL, posing the query to the ORACLE database and returning the answer. All this is done by calling the predicates previously defined. For example, we can check the previous functional dependency at the initial database by calling:

```
| ?- i_p(forall(isbn1):forall(isbn2):forall(id1): (classified(isbn1,
```

```
id1, s0) & classified( isbn2, id1, s0 ) => isbn1 eq isbn2), F),
sql_initial(F).
```

If the ORACLE database corresponds to the initial database given in section 3, we obtain the answer `true`.

6.5.3. Coupling other reasoners with the RDBMS

We have already seen that from SCDBR it is possible to reason with PROLOG at the initial database level. This functionality required having the initial relational database as a list of PROLOG facts. Nevertheless, as pointed out at the beginning of this section, in applications it is most likely that the relational data will be stored as tables in a RDBMS. So, in the most common scenario we will have clauses to be used by an automated reasoner like PROLOG or OTTER plus a collection of relational data in a RDBMS containing the facts that would be needed by the automated reasoners. Since it is not practical to translate the whole contents of the relational system into a file of facts to be appended to the clauses and further used by the automated reasoner, it is necessary the *coupling* (Ceri et al., 1990) of the reasoners to the RDBMS. One possibility would be to have a *loose coupling*, where data from the database is accessed at compilation time. Another possibility would be to have a *tight coupling*, where data from the database is requested at execution time according to the needs of the deductive process. This requires submitting SQL queries from, e.g. PROLOG, to the database in an interactive way. The first alternative makes fewer accesses to the database, but may retrieve too many facts. The second alternative makes more, but more specific calls. For a discussion of the alternatives and optimizations see (Ceri et al., 1990).

We will not attempt a full implementation of the coupling of PROLOG to ORACLE, but rather use software that is already available for this specific task. The chosen system should be called by SCDBR.

7. Reasoning with SCDBR

In this section we will show some reasoning tasks that SCDBR can perform from the database specification.

7.1. Checking the Legality of Actions

A sequence of instantiated actions $[A_1(\vec{t}_1), \ldots, A_n(\vec{t}_n)]$, to be executed in this order, is *legal* if, for each action A_i, the precondition for its execution, say Π_{A_i}, is satisfied in the state that results from the execution of $A_1, \ldots, A_{i-1}$. In order to check the precondition for action A_i in the state resulting from the execution of the $i-1$ preceding actions, we apply $i-1$ times the regression operator to the formula $\Pi_{A_i}(do([A_1(\vec{t}_1), \ldots, A_{i-1}(\vec{t}_{i-1})], S_0))$. This has to be done for each i. In this way, we get rid of the *do* symbol and the resulting formulas are checked against the initial database. This algorithm was introduced by Reiter ((Reiter, 1992, Reiter, 1995)).

SCDBR has the predicate `al` (for *action legality*), that, given a list of transactions, returns the formulas to be checked against the initial database.

Example: We will ask the system to verify if the sequence of transactions, executed from S_0, $[order('3{-}540{-}18199{-}7', 1),\ classifyBook('0{-}412{-}14930{-}3', '4')]$ is legal.

```
| ?- al([order('3-540-18199-7',1),classify_book('0-412-14930-3','4')]
     ,[F1,F2]),p_l([F1,F2], prune_una,[F3,F4]),p_l([F3,F4],prune_uno,
     [F5,F6]),prolog_initial(and(F5,F6)).

true
```

The two formulas to be verified are stored in `F1` and `F2`. The pruners based on unique names for actions and objects are called, obtaining in `F5` y `F6` the following formulas:

- $(\exists title, author, editor, year, edition)$ $BooksInPrint('3{-}540{-}18199{-}7', title, author, editor, year, edition)\ \wedge\ 1 > 0$
- $(\exists quantity)\ Unclassified('0{-}412{-}14930{-}3', quantity, S_0)\ \wedge$ $\neg\exists isbn\ Classified(isbn, '4', S_0)$,

Finally, the predicate `prolog_initial` (see section 6.3.1) verifies if these formulas are logical consequences of the initial database, what turns out to be true.

7.2. Temporal Projection via Regression

We want to know if a formula φ is true in the database obtained after executing a sequence of *legal* actions $A_1, \ldots, A_n$, without having to physically update the database. More precisely, we want to verify if

$$\Sigma \models \varphi(do(A_n, \ldots, do(A_1, S_0)\ldots)) \tag{17}$$

holds, being Σ the specification of the database as given in section 3. In order to do this, we first apply n times the regression operator to φ, as proposed by Reiter (Reiter, 1991), so that the resulting expression only mentions the state S_0. Next, we apply the pruning operators, generating a new formula to be checked against the initial database. In SCDBR, this can be done with PROLOG as a query language or with OTTER.

Example: We want to verify if there is a stock of books with *isbn* $'0{-}412{-}14930{-}3'$ after executing the sequence of legal actions

$$[order('3{-}540{-}18199{-}7', 1), classifyBook('0{-}412{-}14930{-}3', '4')].$$

This can be expressed with the following sentence:

$$\exists n\ Stock('0{-}412{-}14930{-}3', n, do(classifyBook('0{-}412{-}14930{-}3', '4'), do(order('3{-}540{-}18199{-}7', 1), S_0))) \tag{18}$$

To answer this query, we call the following procedures:

```
| ?- i_p( exists(int1) : stock('0-412-14930-3',int1,do(classify_book
     ('0-412-14930-3','4'), do(order( '3-540-18199-7', 1), s0))),F),
     reg_n(F,2, F1),prune_una(F1, F2),prune_uno(F2, F3),csf(F3, F4),
     prolog_initial(F4).

true
```

In this case, the regression operator was applied twice (this is indicated to the system with the second parameter of predicate `reg_n`). Finally, the pruners based on unique names for actions and objects, and the simplifier `csf` were applied. As the result of all these operations we obtain the following formula (stored in variable `F4`):

$$\exists n\ (\exists n_1\ (Stock('0{-}412{-}14930{-}3', n_1, S_0)\ \wedge\ n = n_1 + 1) \vee \forall n_2\ \neg Stock('0{-}412{-}14930{-}3', n_2, S_0)\ \wedge\ n = 1) \tag{19}$$

Finally, with predicate `prolog_initial`, we invoked PROLOG in order to verify if the formula (19) is true in the initial database, what turned out to be the case.

7.3. Physical Update of the Database

SCDBR is able to physically update the initial database after a legal action has been performed. The basic idea is to transform, first, the successor state axioms (SSAs) into PROLOG clauses with the fluents at the successor state in their heads (this can be done automatically the procedure described in section 6.3), and then to use the built–in predicate `findall` of PROLOG to find all the fluents that are true after executing an action. This idea was taken from the "rolling–forward" module implemented in GOLOG (Lesperance et al., 1994).

A more efficient procedure is implemented in SCDBR for specifications that have *context free* SSAs (Lin & Reiter, 1994a, Lin & Reiter, 1995, Lin & Reiter, 1997), that is, of the form:

$$Poss(a,s) \supset [F(\vec{x}, do(a,s)) \equiv (\exists \vec{y}_1)a = A_1(\vec{u}_1) \vee \cdots \vee (\exists \vec{y}_m)a = A_m(\vec{u}_m) \vee F(\vec{x}, s) \wedge \neg(\exists \vec{z}_1)a = B_1(\vec{v}_1) \wedge \cdots \wedge \neg(\exists \vec{z}_n)a = B_n(\vec{v}_n)]. \tag{20}$$

With this kind of SSAs it is easier to detect which are the tuples to be added or removed to/from the current database (Lin & Reiter, 1995, Lin & Reiter, 1997). SCDBR, before starting to progress the database, parses the SSAs in order to verify if they are context free. If this is the case, it applies the more efficient updating algorithm. Otherwise, it applies the general procedure.

Example: We wish to update the initial library database as the result of classifying a new copy of the book with *isbn* number $'0-412-14930-3'$:

```
| ?- progress(classify_book('0-412-14930-3','13')).
```

We obtain a new initial database, whose fluents (tables) have S_0 as the state component, as expected. So, we can use all the SCDBR functionalities with the new initial database. For example, we can see that in the updated database we now have one more copy of the book with *isbn* $'0-412-14930-3'$:

```
| ?- prolog_initial(stock('0-412-14930-3',int1,s0)).

[[int1,3]]
```

7.4. Integrity Constraints

In this paper we will concentrate on *static integrity constraints.* That is, on sentences of the form

$$\forall s \ (S_0 \leq s \supset \varphi(s)), \tag{21}$$

where s is the only state term mentioned in $\varphi(s)$. $S_1 \leq S_2$ means that S_2 is of the form $do([A_1, \ldots, A_n], S_1)$, where, for each i, $Poss(A_i, do([A_1, \ldots, A_{i-1}], S_1))$ holds. This relation is defined by induction on states in (Reiter, 1992, Reiter, 1995). In the rest of this section we will talk about "the IC $\varphi(s)$", tacitly assuming the condition $S_0 \leq s$.

Given an integrity constraint (IC), we would like to be sure that if it is true in all states that are accessible from the initial state by means of a finite sequence of legal actions. This could be established by proving the IC from the specification. Otherwise, we might feel tempted to modify our specification so that the desired integrity is entailed by the new specification (Lin & Reiter, 1994b).

SCDBR is able to solve these tasks, namely: (1) to automatically proof ICs that are logical consequences of the the specification; and, with some qualifications, (2) to automatically modify the specification in order to subsume the IC. In the next sections we describe these two functionalities.

7.4.1. Automated Proofs of Integrity Constraints

We want to prove the IC (21) from the specification Σ, This can be done by induction on states (Reiter, 1993). More precisely, according to a result by Lin and Reiter (Lin & Reiter, 1994b), the following steps have to be checked:

1. $\Sigma \models \varphi(S_0)$, that is, the IC hold at the initial state.

2. $\Sigma \models \forall(a,s).\varphi(s) \wedge Poss(a,s) \supset \varphi(do(a,s))$. This is the inductive step, proving that the IC holds at every legal successor state $do(a,s)$ of every state s at which the IC holds.

In SCDBR, these two tasks are done with the predicate `prove_ic`. The proof in the base case, $\varphi(S_0)$, is done with PROLOG as a query language (see section 6.3) or with ORACLE in case the initial database is a conventional relational database. The proof of the inductive step is done with OTTER[8].

Example: The specification of the library database entails the following functional dependency:

$$Classified(isbn_1, id, s) \;\wedge\; Classified(isbn_2, id, s) \;\supset\; isbn_1 = isbn_2, \tag{22}$$

which states that two different books cannot be classified with the same identifier id. We will use predicate `prove_ic` to prove this IC.

```
| ?- i_p( forall(isbn1) : forall(isbn2) : forall(id1) : (classified(
     isbn1,id1,s) & classified(isbn2,id1,s) => isbn1 eq isbn2 ), F),
     prove_ic(F).

Proof for the initial database:
------------------------------
The proof in the initial state was successful.

Inductive step:
--------------
invoking Otter. I'll tell you when I'm done.

-------- PROOF --------
```

□

We have also used the term rewrite system RRL (Kapur & Zhang, 1995) to prove integrity constraints. This system has powerful capabilities for automatically performing proofs by induction, so it is not necessary to give explicitly to RRL the inductive step. The methodology and results are reported in (Bertossi et al., 1996b). We are currently interfacing SCDBR to RRL and implementing the general methodology.

7.4.2. Modifying the Specification: Ramifications

A first attempt to embed desired ICs into the specification, in such a way that the IC is entailed by the new specification, consists in modifying the effect axioms (or directly the SSAs). This can be seen as a way of solving the ramification problem, that is, including the side effects of actions into the specification given in terms of SSAs. To provide a general solution is an open problem. Pinto (Pinto, 1994) gives a solution for a syntactical class of ICs, namely, binary ICs, that is, ICs where the formula $\varphi(s)$ in (21) is of the form:

$$[\neg]F_1(\vec{x_1}, s) \vee [\neg]F_2(\vec{x_2}, s) \vee \psi, \tag{23}$$

where F_1 and F_2 are fluents in the specification; ψ is a formula without state terms; and $[\neg]$ represent the fact that the fluents may be negated or not. Notice that functional dependencies are a special cases of binary ICs.

We will briefly describe Pinto's procedure. For simplicity, we will assume that the ICs have the form:

$$F_1(\vec{x_1}, s) \vee F_2(\vec{x_2}, s) \vee \psi. \tag{24}$$

For each negative effect axiom for F_1, of the form

$$Poss(A(\vec{x}), s) \wedge \varepsilon^-_{F_1}(\vec{x}, \vec{y}, s) \supset \neg F_1(\vec{y}, do(A(\vec{x}), s)), \tag{25}$$

the following new positive effect axiom for F_2 is added to the specification:

$$Poss(A(\vec{x}), s) \wedge \varepsilon^-_{F_1}(\vec{x}, \vec{y}, s) \wedge \neg\psi \supset F_2(\vec{y}, do(A(\vec{x}), s)). \tag{26}$$

The same is done for fluent F_2. If negations appear in (24), the procedure works in a similar way.

Example: Let us delete the following effect axiom from the library specification

$$\begin{aligned} &Poss(classifyBook(isbn, id), s) \ \wedge \ Stock(isbn, quantity, s) \ \supset \\ &\qquad\qquad \neg Stock(isbn, quantity, do(classifyBook(isbn, id), s)). \end{aligned}$$

We obtain a new, but semantically incorrect, specification, in the sense that now we can conclude that if, at a given state, we had *quantity* books in stock with identifier *isbn*, and an extra copy was classified, then we will have both $quantity + 1$ and *quantity* copies of the same book in stock, what is not intended. More precisely, the modified specification does not satisfy the functional dependency:

$$Stock(isbn, quantity_1, s) \ \wedge \ Stock(isbn, quantity_2, s) \ \supset \ quantity_1 = quantity_2.$$

This IC is logically equivalent to the binary IC:

$$\neg Stock(isbn, quantity_1, s) \ \vee \ \neg Stock(isbn, quantity_2, s) \ \vee \ quantity_1 = quantity_2.$$

The SCDBR procedure `ramification` will generate a new specification that entails the functional dependency (actually, it will recover the original library specification). More precisely, this is done in the system in this way:

```
| ?- i_p(neg stock(isbn1,int1,s) v neg stock(isbn1,int2,s) v int1 eq
     int2, R1),ramification(R1,R2),p_i(R2,R3).
```

The variable `R2` contains the new set of effect axioms. The original and the new sets of effect axioms differ by the following axiom provided by the procedure[9]:

$$
\begin{aligned}
& Poss(classifyBook(isbn, id), s) \ \wedge\ \neg quantity_1 = quantity_2 \ \wedge \\
& \qquad ((Stock(isbn, quantity_3, s) \ \wedge\ quantity_1 = quantity_3 + 1) \ \vee \\
& \qquad\ (\forall quantity\ \neg Stock(isbn, quantity, s) \ \wedge\ quantity_1 = 1)) \ \supset \\
& \qquad\qquad\qquad \neg Stock(isbn, quantity_2, do(classifyBook(isbn, id), s)).
\end{aligned}
$$

Notice that this axiom is slightly more general than the original one. It says that: (1) If there are *quantity* books with *isbn* in stock in the current state, then, after classifying an extra copy, there will be no quantity of copies in stock different from $quantity + 1$; (2) If the are no books with *isbn* in stock, then after classifying a book with this *isbn*, there is no quantity in stock different from 1.

7.4.3. *Modifying the Specification: Qualifications*

In (Lin & Reiter, 1994b), Lin and Reiter give an alternative and general methodology for embedding an IC into a new specification, that will entail it. It consists in modifying the action precondition axioms, putting further constraints on the actions that qualify to be executed. Given an action A, with precondition axiom $\Pi_A(s)$, and an IC, $\varphi(s)$, if we want to be sure that $\varphi(s)$ will hold after executing A, it suffices to replace $\Pi_A(s)$ by $\Pi_A(s) \wedge \varphi(do(A, s))$. In order to avoid the occurrences of both s and $do(A, s)$ in this formula, the regression operator can be applied, so that the new precondition will be $\Pi_A(s) \wedge \mathcal{R}[\varphi(do(A, s))]$. This is done for each named action A. In this way the IC will hold after executing any legal action (now, satisfying the new preconditions).

Based on this methodology, the SCDBR predicate, `qualification`, can compute a new specification that satisfies a given IC.

Example: In the original library specification we will replace the precondition axiom for the action *classifyBook* by the formula:

$$Poss(classifyBook(isbn, id), s) \equiv (\exists copies)\, Unclassified(isbn, copies, s). \tag{27}$$

Now we have an incorrect specification, because there may appear two different books with the same *id*. More precisely, this specification does not entail the functional dependency:

$$Classified(isbn_1, id, s) \ \wedge\ Classified(isbn_2, id, s) \ \supset\ isbn_1 = isbn_2 \tag{28}$$

The new specification, from which the IC does follow, can be computed in this way:

```
| ?- i_p( forall(isbn1) : forall(isbn2) : forall(id1) : ( classified
     (isbn1,id1,s) & classified(isbn2,id1,s) => isbn1 eq isbn2 ), F),
     qualification(F,R).
```

The new set of action precondition axioms is stored in the variable `R`. The predicate `qualification` automatically simplifies the formula resulting from the regression according to the unique names axioms for actions. The new precondition axiom for *classifyBook* is the formula:

$$\begin{aligned} Poss(classifyBook(isbn, id), s) \equiv\ & \exists copies\ Unclassified(isbn, copies, s) \wedge \\ & \forall(isbn_2, isbn_3, id_2)\ ((\ isbn = isbn_3\ \wedge\ id = id_2\ \vee\ Classified(isbn_3, id_2, s)) \wedge \\ & \qquad (\ isbn = isbn_2\ \wedge\ id = id_2\ \vee\ Classified(isbn_2, id_2, s)) \\ & \qquad\qquad \supset\ isbn_3 = isbn_2), \end{aligned}$$

which is equivalent to:

$$\begin{aligned} Poss(classifyBook(isbn, id), s) \equiv\ & \exists copies\ Unclassified(isbn, copies, s)\ \wedge\ \varphi(s) \wedge \\ & \forall isbn_2\ (Classified(isbn_2, id, s)\ \supset\ isbn = isbn_2). \end{aligned}$$

Here, $\varphi(s)$ is the IC (28). Notice that this formula can be eliminated from the precondition, because the IC can be proved by induction without that extra condition (it is exactly the induction hypothesis, see section 7.4.1).

The new axiom basically adds to (27) the fact that *classifyBook* can be executed if, among other conditions, every time we try to classify an exemplar of a book, with an *id* that was used before, then we must be classifying the same exemplar. This new precondition for *classifyBook* is weaker than the original precondition we had in the specification in section 3. Before, it was impossible to execute twice the same classification action, that is, on an already classified exemplar. Now, we are allowed to repeat the action (classifying an already classified exemplar with the same *id*), but there will be no new effects.

7.5. Historical Queries

SCDBR can answer historical queries (HQs). We have developed a methodology for posing HQs to a virtually updated database and for answering them. The methodology is presented in detail in (Siu et al., 1996, Siu & Bertossi, 1996a) (see also (Siu & Bertossi, 1996c)).

The most basic kind of query we can think of is "*Given a formula $\varphi_i(s)$ and two states s_{L_i}, s_{U_i} along a legal transaction list T, does φ_i hold in all intermediate states between s_{L_i}, s_{U_i} (including s_{L_i} and excluding s_{U_i})?*". A query of this kind can be expressed by the formula $(\forall s) s_{L_i} \leq s < s_{U_i} \supset \varphi_i(s)$, and is called a *universal query*. The states s_{L_i}, s_{U_i} are called the *bounds* of the HQ[10]. In SCDBR we can express queries that are existential quantifications on the bounds of conjunctions of universal queries (plus restrictions on the bounds). In the queries we find the accessibility relation, $s_1 \leq s_2$ (see section 7.4).

A general algorithm for answering queries in this format consists basically in the following steps: First, for every formula φ_i appearing in a universal query, obtain a list of states for which the formula is true. This requires evaluation in all states along T. The outcome of this module is what we call a *map* of the formulas φ_i.

Second, from the map and the restrictions on the bounds, that can be translated into numerical constraints by introducing a signed distance function of pairs of accessible states, a new query is generated that only mentions (in)equalities between pairs of integers. This new query can be seen as a Constraint Satisfaction Problem (CSP), for which we apply constraint logic programming, more precisely, CLP(FD) (Codognet & Diaz, 1996, Diaz, 1994). There original query has positive answer iff the CSP has a solution.

This algorithm needs to keep the truth value of every formula φ_i in every state along T. This process can be optimized: Only a few actions should be able to change the truth value of the formulas φ_i from one state to another, because every formula φ_i mentions only a few fluents. Moreover, only a few fluents should be "tracked" to answer the queries. We will explain how to characterize those "relevant" actions and fluents, which can be used to improve the algorithm for computing the map of the query by updating only a few tuples in the database (other optimizations can be found in (Siu & Bertossi, 1996b)).

If a fluent F has the successor state axiom $Poss(a,s) \supset F(do(a,s)) \equiv \Phi_F$, every fluent F' appearing in Φ_F will be "relevant" to F, since the truth value of F in the successor state "depends" on the truth value of F'. Therefore, if we are interested in the truth value of F along T, we must track any fluent mentioned in the SSA of F. We must repeat this process for each F', looking recursively into the SSAs until no new fluents are found. In (Siu & Bertossi, 1996b), a more refined and terminating version of this algorithm is described.

To determine the relevant actions, we follow a similar approach: any action A that affects the truth value of F from a state s to state $do(A,s)$ must be mentioned in Φ_F, so all the actions that are relevant to a fluent must be mentioned in the SSA of some fluent that is relevant to the given fluent[11].

Example: Let us consider the following legal transaction list:

$$\begin{aligned} T = [&classifyBook('0{-}412{-}14930{-}3','13'),\\ &\qquad classifyBook('0{-}412{-}14930{-}3','14'), deleteBook('10')] \end{aligned} \tag{29}$$

We want to know if there is a state, along the current transaction list, such that there is a book with more that three copies in stock. This is represented by the formula:

$$\exists s(S_0 \leq s \leq do(T,S_0) \ \wedge \ \exists(isbn,int).\ Stock(isbn,int,s) \ \wedge \ int > 3) \tag{30}$$

In SCDBR the query is posed as follows:

```
| ?- set_curr_trans([ classify_book( '0-412-14930-3', '13' ),
     classify_book('0-412-14930-3','14'),delete_book('10')]).

| ?- set_curr_query([[dist(su1,sl1)=1],[sl1,su1,some(isbn1,some(int1,
     and(stock(isbn1,int1),int1>3)))]]).

| ?- process_curr_query.

The map is
[[[2,2]]]
.
Query succeeds. Possible bounds values are:
Sl1 : 2
Su1 : 3
```

As expected, the answer is *true* (only in the state resulting from the execution of the first two actions).

8. Conclusions and Related Work

We have presented an automated system written in SICSTUS PROLOG that is able to perform several reasoning tasks when it is run with an initial input database and a preliminary specification of the effects of transactions and their execution preconditions expressed in a language of the situation calculus. First, the system generates a new specification that solves the frame problem in the form proposed by Ray Reiter.

Having the new specification, the system is able to perform several reasoning tasks, like: (1) checking legality of transactions by regression to the initial database, (2) answering queries about a virtually updated database by regression to the initial database, (3) modification of the specification in order to impose desirable integrity constraints, (4) automated proofs of integrity constraints, (5) materialization of virtual updates, (6) answering queries about the evolution of the database by goal-oriented progression of the database, (7) automated proofs at the initial database level.

For solving those tasks the system has been connected to other automated systems like: RRL, OTTER, CLP(FD), ORACLE, PROLOG (as a theorem prover). The system has some procedures to interact with these systems and the user like: conversion of first–order sentences into PROLOG programs, SQL queries, clauses to be used by OTTER, conditional equations to be used by RRL, automated generation of different domain closure axioms, latexification of formulas, ...

We have achieved to develop a computational system that is able to reason from and with relational databases (actually the system can also handle first–order databases, nevertheless we have not emphasized this capability) plus a logical specification of the transactions that may affect the tables. It is important to stress the fact that tables and specification of updates coexist at the same object level with a clean and clear first–order semantics. This is possible if one takes seriously Reiter's proposal about a transaction based semantics of database updates derived from specifications in the situation calculus. We have brought this approach into practice.

Our approach can be seen as a logical reconstruction and implementation of the transaction based approach presented in (Abiteboul & Vianu, 1989), but with basic transactions at a higher level than those of insertion, deletion and modification. Our action preconditions are also more general that the conditions allowed in that paper. It is interesting to realize that our system brings into practice reasoning about the relationship and interaction between transaction based and constraint based semantics for databases as discussed there.

Issues for further research and implementation around SCBDR are: (1) development of a bigger and interesting application for SCDBR, in which its facilities for modeling and reasoning can be better appreciated, (2) usage of SCDBR in combination with the RDBMS to which it is coupled in order to answer historical queries in the spirit of section 7.5, when one starts with a conventional relational database and makes partial progression of the database by means of a RDBMS, (3) extension of the methodology for proving integrity constraints with RRL and full integration of it into SCDBR.

We have found many points in common with the implementations done by the Cognitive Robotics Group at the University of Toronto (Lesperance et al., 1994). This is not strange since both groups have as a starting point specifications of dynamic worlds in terms of successor state axioms written in languages of the situation calculus. Nevertheless, their implementations and developments are focused on cognitive robotics and not on database systems. Their specifications are written as logic programs and directly in terms of successor state axioms (translated into rules); instead we try to stay first–order as much as possible having the freedom to commute to logic programming if desired. Furthermore, we start from preliminary specification of effects from which we derive the necessary successor state axioms. As far as we know, the Toronto group has not worked on implementations of proofs of integrity constraints and answers to historical queries. On their side, they have done a lot of interesting work on implementation of complex and epistemic (knowledge producing) actions in their high level robot programming language GOLOG. We have considered the case of basic transactions only.

An approach to database updates specifications is that is close in spirit to ours is *transaction logic* (Bonner & Kifer, 1994). They have developed powerful mechanisms for specifying and executing complex transactions. Nevertheless, that approach concentrates on complex actions, leaving basic actions not fully specified, in the sense that they do not provide an explicit declarative solution to the frame problem for such actions. An implementation of a Horn fragment of transaction logic is reported in (Hung, 1996).

Acknowledgments

We are grateful to Bernardo Siu (who worked on historical queries and physical updates), Pablo Saez (who worked on automated proofs of integrity constraints and PROLOG as a query language), Mauricio Strello (who worked on physical updates and connecting SCDBR to a DBMS), Alejandra Delaporte (who worked on the user interface). We are also very grateful to Ray Reiter, Deepak Kapur, Javier Pinto, Tania Bedrax, Erik Sandewall, Jack Minker, Richard Scherl, Mahadevan Subramaniam, Aris Zakinthinos, Daniel Marcu y Fangzhen Lin, Alvaro Campos, David Fuller and Pablo Straub for their help, advice, information, joint work, etc. This research has been supported partly by DIUC, and FONDECYT (#193–0554) grants.

Notes

1. SCDBR is available from `ftp://ing.puc.cl/pub/escuela/dcc/scdbr`.
2. Although, Reiter's transformation can also be seen as a particular form of non-monotonic reasoning. This solution was carefully discussed and assessed in the context provided by Sandewall's assessment methodology (Sandewall, 1994) in (Bedrax & Bertossi, 1995).
3. Fluents are properties that may change as actions are performed. In the database context, fluents correspond to the tables in the database.
4. In (Reiter, 1992, Reiter, 1995), and in contrast to the presentation in (Reiter, 1991), Reiter starts with specifications given in terms of successor state axioms.
5. Notice that the formulas in this database represent a relational database.
6. For example, it replaces a formula $\varphi \vee \mathit{false}$ by φ, and a formula $\exists x\ (\varphi \wedge \psi(x))$ by $\varphi \wedge \exists x\ \psi(x)$, when x does not appear in φ, etc.
7. SCDBR automatically generates columns or attributes `c1`, `c2`, ... in the ORACLE database, corresponding to the arguments in the first–order predicates.
8. For doing this, we put in the "usable list" the specification Σ and unique names axioms for actions and individuals. The negated inductive implication is plugged into the "set of support". Paramodulation, binary resolution and hyperresolution are also set.
9. The system is able to realize when a fluent in a binary constraint subsumes the other one, as in functional dependencies. In that case, it computes the new effect axiom for more general fluent.
10. In the literature it is possible to find this kind of queries about different states of the database under the name of "temporal queries" (Snodgrass & Ahn, 1986).
11. There is a counterexample to this claim in (Siu & Bertossi, 1996b). However, the claim does hold for specifications generated from effect axioms.

References

S. Abiteboul and V. Vianu. A Transaction-Based Approach to Relational Database Specification. Journal of the ACM, 36(4):758–789, 1989.

T. Bedrax-Weiss and L. Bertossi. Underlying Semantics for the Assessment of Reiter's Solution to the Frame Problem. In J. Wainer and A. Carvalho, editors, *Advances in Artificial Intelligence, Proc. XII International Conference of the Brazilian AI Society (SBIA'95), LNAI 991*. Springer, LNAI 991, 1995.

L. Bertossi, M. Arenas, and C. Ferretti. SCDBR: An Automated Reasoner for Database Updates (Extended Abstract). In *Proc. XVI International Conference of the Chilean Computer Science Society (SCCC'96), Valdivia, Nov. 13–15*, 1996.

L. Bertossi and J. Ferretti. SCDBR: A Reasoner for Specifications in the Situation Calculus of Database Updates. In *Temporal Logic. Proceedings First International Conference, ICTL '94, Bonn, Germany, July 1994, LNAI 827*, pp. 543–545. Springer, 1994.

L. Bertossi, J. Pinto, P. Saez, D. Kapur, and M. Subramaniam. Automating Proofs of Integrity Constraints in the Situation Calculus. In *Foundations of Intelligent Systems. Proc. Ninth International Symposium on Methodologies for Intelligent Systems (ISMIS'96), Zakopane, Poland*, pp. 212–222. Springer, LNAI 1079, 1996.

M. Böhlen. Managing Temporal Knowledge in Deductive Databases. PhD Thesis #10802, ETH, Zürich, 1994.

A. Bonner and M. Kifer. An Overview of Transaction Logic. In *Theoretical Computer Science*, 133:205–265, 1994.

S. Ceri and G. Gottlob and L. Tanca. Logic Programming and Databases. Springer, 1990.

J. Chomicki and D. Toman. Implementing Temporal Integrity Constraints using an Active DBMS. IEEE Transactions on Knowledge and Data Engineering, 7(4):566–582, 1995.

Ph. Codognet and D. Diaz, Compiling Constraints in CLP(FD). In Journal of Logic Programming, pp. 185-226, 1996.

D. Díaz. *CLP(FD) 2.21 User's Manual.* INRIA–Rocquencourt, France, 1994.

R. Fikes and N. Nilsson. STRIPS: A new approach to application of theorem proving procedures to problem solving. In *Readings in Planning*, 1971.

A. Haas. The Case for Domain-Specific Frame Axioms. In F. Brown, editor, *The frame problem in artificial intelligence. Proceedings of the 1987 Workshop*, pp. 343–348, Los Altos, CA, 1987. Morgan Kaufmann Publishers, Inc.

S.Y.K. Hung. Implementation and Performance of Transaction Logic in PROLOG. Master's thesis, Department of Computer Science, University of Toronto, 1996.

D. Kapur and H. Zhang. An Overview of Rewrite Rule Laboratory (RRL). *J. of Computer and Mathematics with Applications*, 1995.

R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

Y. Lésperance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. "A Logical Approach to High–Level Robot Programming – A Progress Report". In *Proc. AAAI Fall Symposium of Control of the Physical World by Intelligent Systems*, New Orleans, LA, November 1994.

F. Lin and R. Reiter. How to Progress a Database (and Why) I: Logical Foundations. In E. Sandewall J.Doyle and P. Torasso, editors, *Proc. Fourth Int. Conf. on Principles of Knowledge Representation and Reasoning*, pp. 425–436. Morgan Kaufmann, 1994.

F. Lin and R. Reiter. State Constraints Revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994. Special Issue on Action and Processes.

F. Lin and R. Reiter. How to Progress a Database II: The STRIPS Connection. In *Proc. IJCAI'95*, pp. 2001–2007, 1995.

F. Lin and R. Reiter. How to Progress a Database. *Artificial Intelligence*, 92(1–2), pp. 131–167, 1995.

J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.

J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pp. 463–502, Edinburgh, Scotland, 1969. Edinburgh University Press.

W.W. McCune. *OTTER 3.0 Reference Manual and Guide.* Argonne National Laboratory, Technical Report ANL-94/6, 1994.

E. Pednault. ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus. In H. Levesque R. Brachman and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pp. 324–332, San Mateo, CA, 1989. Morgan Kaufmann Publishers, Inc.

J. Pinto. *Temporal Reasoning in the Situational Calculus.* PhD thesis, Department of Computer Science, University of Toronto, 1994.

R. Reiter. Nonmonotonic Reasoning. In *Annual Reviews in Computer Science*, volume 2, pp. 147–186, 1987.

R. Reiter. The Frame Problem in the Situation Calculus: a Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pp. 359–380. Academic Press, 1991.

R. Reiter. On Specifying Database Updates. Technical Report KRR-TR-92-3, University of Toronto, Department of Computer Science, Toronto, Canada, 1992.

R. Reiter. Proving Properties of States in the Situation Calculus. *Artificial Intelligence*, 64(2):337–351, 1993.

R. Reiter. On Specifying Database Updates. *Journal of Logic Programming*, 25(1):53–91, 1995.

P. Saez. Automated Proofs of Database Integrity Constraints. PhD thesis, Catholic University of Chile, School of Engineering, Department of Computer Science, 1997.

E. Sandewall. *Features and Fluents I. A Systematic Approach to the Representation of Knowledge about Dynamical Systems.* Oxford University Press, 1994.

L. Schubert. Monotonic Solution of the Frame Problem in the Situation Calculus: an Efficient Method for Worlds with Fully Specified Actions. In H. Kyburg, R. Loui, and G. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pp. 23–67, Boston, MA, 1990. Kluwer Academic Press.

B. Siu and L. Bertossi. Answering Historical Queries in Databases using Relevance. *Submitted for publication*, 1996.

B. Siu, L. Bertossi and M. Arenas. A Semantical Notion of Relevance in Specifications of Database Updates and its Computational Counterpart. *Submitted for publication*, 1996.

B. Siu and L. Bertossi. Answering Historical Queries in Databases. Technical Report RT-PUC-DCC-2-96, Pontificia Universidad Catolica de Chile, Escuela de Ingenieria, Departamento de Ciencia de Computacion, Casilla 306, Santiago 22, Chile. ftp://ing.puc.cl/dcc/techReports, 1996.

B. Siu and L. Bertossi. Answering Historical Queries in Databases (Extended Abstract). In *Proc. XVI International Conference of the Chilean Computer Science Society (SCCC'96), Valdivia, Nov. 13–15*, 1996.

R. Snodgrass and I. Ahn. Temporal Databases. IEEE Computer, Sept.:35–42, 1986.

J. Ullman. *Database and Knowledge-Based Systems*, Vol. 1. Computer Science Press, 1988.

A. van Gelder and R. Topor. Safety and Correct Translation of Relational Calculus Formulas. Proc. ACM Symposium on Database Systems (PODS), pp. 313–327, 1987.

A. Zakinthinos. Improved Runtime Complexity for Historical Queries with Application to Secure Systems. Department of Computer Science, University of Toronto, 1993. Term Project for CSC2532.