PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

PROVING INTEGRITY CONSTRAINTS IN THE SITUATION CALCULUS USING AUTOMATED THEOREM PROVERS

PABLO SÁEZ G.

Tesis presentada a la Escuela de Ingeniería
de la Pontificia Universidad Católica de Chile
para optar al Grado de
Doctor en Ciencias de la Ingeniería

Profesor Supervisor:
LEOPOLDO BERTOSSI D.

Santiago de Chile, 1999

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
Departamento de Ciencia de la Computación

PROVING INTEGRITY CONSTRAINTS IN THE SITUATION CALCULUS USING AUTOMATED THEOREM PROVERS

PABLO SÁEZ G.

Tesis presentada a la Escuela de Ingeniería
de la Pontificia Universidad Católica de Chile
para optar al Grado de
Doctor en Ciencias de la Ingeniería

Profesor Supervisor:
LEOPOLDO BERTOSSI D.

Santiago de Chile, 1999

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
Departamento de Ciencia de la Computación

PROVING INTEGRITY CONSTRAINTS IN THE SITUATION CALCULUS USING AUTOMATED THEOREM PROVERS

PABLO SÁEZ G.

Tesis presentada a la Comisión integrada por los profesores:

LEOPOLDO BERTOSSI D.

PABLO STRAUB

JORGE LOBO

JOSÉ MIGUEL AGUILERA

para completar las exigencias del grado de
Doctor en Ciencias de la Ingeniería

Santiago de Chile, 1999

Fecha de Impresi'on: 13 de octubre de 1999

Documento preparado en LaTeX usando

tesisuc.cls

El formato de Tesis de la Escuela de Ingeniería
de la Pontificia Universidad Católica de Chile.

Escrito y actualizado por

Marcelo Matus Acuña
mmatus@mat.puc.cl

Esta tesis está dedicada a sugerirle que evalúe en su casa la factibilidad del uso de fuentes de energía alternativas al gas (hornos microondas, cocinas eléctricas y/o una caldera común a varios departamentos para calentar el agua).

INDEX

Pág.

RESUMEN

Estudiamos en esta tesis el problema de la demostración de restricciones de integridad en bases de datos a partir de una especificación lógica de la dinámica de la base de datos, mediante demostradores mecánicos de teoremas. Construimos una metodología general para demostradores inductivos basados en reescritura condicional de términos, basada en una solución al problema de la representación de cuantificadores de primer order en estos sistemas. Analizamos la correctitud y el grado de aplicabiblidad de la solución en el ámbito de las bases de datos.

ABSTRACT

We study in this thesis the problem of proving database integrity constraints from a logical specification of the database dynamics by means of automatic theorem provers. We construct a general methodology for inductive conditional term rewriting theorem provers based on a solution to the problem of representing first order quantifiers on those reasoning systems. We analyse the soundness and scope of applicability of this solution in the database domain.

I. INTRODUCTION

Integrity constraints (ICs, for short) are (logical) sentences that are expected to be satisfied at every legal state of a database. In existing database management systems, integrity constraints are usually *checked* in real time at some states (see for example [28, 27] or in general [12, 40]). This checking process is usually time–consuming and moreover not always conclusive, since it does not explain why the IC is not satisfied or what could be done to satisfy it.

The scenario is different when a *logical specification* of the dynamics of the database is somehow available. In this case, one might attempt to *prove* in logical terms that the IC holds at every legal state of the database. By doing so, it becomes possible to ensure that the database will never fall into an inconsistent state[1] whenever only legal transactions are executed[2]. In this scenario, an integrity constraint becomes a proposition whose truth at every state of the database should be a logical consequence of the specification of the database updates.

Our interest in this thesis is the automation of proofs of integrity constraints and the integration of this functionality into SCDBR, an automated reasoner on database updates specifications [4, 14, 5].

We adopt Raymond Reiter's formalism [31, 34] for the specification of the dynamic behaviour of a database, which is based on the situation calculus [24], a family of many–sorted languages of predicate logic that allows specifying and reasoning about evolving worlds. The formalism is based on Reiter's general proposal for the specification of action and change in the situation calculus in such a way that the Frame Problem [24] has a simple solution [30].

[1] The usual notion of "inconsistent state" in the database field means that the state is inconsistent with the integrity constraints, or more precisely that the database state, seen as an interpretation, is not a model of the set of integrity constraints that have been defined for the database.

[2] The legality of a database transaction is usually checked by means of preconditions for the transaction that are verified *prior to* the execution of the transaction itself. It is therefore possible for a legal transaction to produce an inconsistent state if the database updates or its preconditions are wrongly defined.

Reiter's database specification formalism basically consists of:

- A complete description of the initial state of the database (called S_0).
- A set of transaction names, and for each transaction an axiom that specifies precisely the conditions that make the transaction possible (i.e. legal, or executable).
- A set of fluents (equivalent to tables in the relational model) and for each fluent an axiom that specifies its truth value (for every object in the appropriate domain) after the execution of an arbitrary transaction.
- A second–order axiom that defines the *accessible states* to be exactly those obtained from the initial state by the performance of a finite sequence of legal transactions [31, 34, 33].

In this scenario, it is possible to express static integrity constraints[3] as first order situation calculus sentences, and since SCDBR is already able at this moment to perform several reasoning tasks from those specifications, it is natural to consider including the automation of proofs for such integrity constraints as another SCDBR functionality.

Because of the inductive nature of the accessibility relation between states, the use of mathematical induction is convenient and often necessary. A typical proof of an arbitrary IC will consist in proving, first, that the IC holds at the initial state S_0, and next, in proving that if the IC holds at an arbitrary state s and a legal action a is performed at state s, the IC remains true in the succesor state $do(a, s)$.

The first question, then, to be considered in the automation of IC proofs in the situation calculus is whether to build an ad-hoc (inductive) theorem prover for that language or to use (and maybe adapt) an existing one. Since the construction of an ad-hoc prover for the situation calculus is a task that could exceed our

[3] A static integrity constraint, as opposed to a dynamic one, is a property of a single state, or a logical sentence that takes only one state as argument. Dynamic ICs are properties of two or more states, possibly consecutive ones. In the situation calculus, dynamic integrity constraints take two or more arguments of type state.

resources, and moreover would duplicate efforts, since most of the deductive needs in the situation calculus are already covered by many well-known theorem provers (for instance, Otter [25], or even Prolog), we considered using some already existing inductive theorem prover.

Since much of Reiter's formalism can be easily translated into a conditional equational language, we considered using a conditional term-rewriting based system with inductive capabilities. The theorem prover RRL [17, 18] was a natural alternative to be considered, since it has both features. We also considered the possibility of automatically proving integrity constraint using resolution–based theorem provers, although those systems do not provide in general induction mechanisms, which constitutes a drawback in the case of the situation calculus.

The main difficulty that appears when using conditional term rewrite systems (CTRS, for short) in our situation calculus problem (and in many other contexts as well) is the lack of a natural mechanism for representing arbitrary first order quantifications, that are used in the situation calculus (as in any first order language). Bertossi et al. provide in [6] a preliminary methodology for implementing proofs of integrity constraints using inductive CTRS and in particular for solving the quantification problem. This problem and its solution in the context of proofs of database integrity constraints are the main topic of chapter 3. We make more precise and expand the methodology reported in [6]. We also study the soundness and scope of applicability of the expanded methodology in the case of relational databases, where the existence of finite active domains constitutes the basis for the soundness of the methodology.

The solution consists basically in: (1) representing a quantified formula as a boolean term that encodes the quantification by going through a finite list of objects that represents a suitable domain; and (2) generating a set of inductive proof hints for the theorem prover, such as generalization suggestions and lemmata to be proved. This set of proof hints can be compared to a Hilbert-like deductive system [13] for bounded quantifications using term rewriting. We prove the completeness of this Hilbert–like deductive system (in an appropriate sense).

In chapter 2 we briefly introduce Reiter's database updates specification formalism by means of an example. In section 3.1 we explain the quantification problem in more detail. In section 3.2 we recall in informal terms the solution to this problem as presented in [6] along with a survey of other relevant papers. The following sections are the continuation of that work. In section 3.3 we propose a pre-processing of Reiter's specification that allows eliminating many non-essential quantifiers from it, thus diminishing the proof overhead due to the introduction of bounded quantification. In section 3.4 we formalize the transformation of situation calculus specifications into conditional equational specifications. In section 3.5 we define a set of safeness criteria for situation calculus formulae that guarantee the soundness of the bounded quantification methodology, which is proved in section 3.6. In section 3.7 we discuss some implementation issues of the bounded quantification method using the RRL theorem prover. In section 3.8 we show an actual example of the application of the bounded quantification method in a situation calculus integrity constraints proof. In section 3.9 we compare this work to others. In chapter 4 we discuss the problem of stronger inductive lemmata generation for situation calculus integrity constraints: we present a simple result related to that topic and a set of heuristics that can lead to a solution of the addressed problem. In chapter 5 we discuss our implementation of situation calculus integrity constraint proofs by means of resolution–based theorem provers. This implementation is strongly based on Lloyd and Topor's algorithm [23]. We make some concluding remarks in chapter 6.

II. A LOGIC-BASED FORMALISM FOR SPECIFYING DATABASE UPDATES

In [30, 31, 34], Raymond Reiter proposed a general formalism for the specification of the evolution of a dynamic world (e.g. a database) that goes through different states as actions (e.g. database transactions) are performed. His formalism is based on the situation calculus [24], a family of first-order languages developed for modeling evolving worlds and reasoning about them. The world goes discretely from state to state, starting from the initial state, S_0 (it is therefore an epistemic assumption in the situation calculus that this initial state exists, is unique, and can be axiomatized). The formalism finds its origin in Reiter's solution to the frame problem [30].

We introduce the situation calculus and Reiter's database update specifications by means of a stock database example. We assume that orders can be received (from some client) only if the product in the order is in stock or is provided by some provider. However, section A of the store receives orders only for products for which no provider exists. An inductive consequence of this fact is that, if products are never deleted from stock (perhaps by defining 0 as a possible value for stocks), then the products ordered to section A are in stock at every accessible state of the database.

Let us assume the existence of two products in stock at the initial state, p_1 and p_2, of which product p_2 has been ordered in section A. The following axioms formalize the above statements:

$$
\begin{aligned}
(\forall prod) InStock(prod, S_0) &\equiv (prod = p_1 \vee prod = p_2) \\
(\forall prod) OrderedA(prod, S_0) &\equiv (prod = p_2) \qquad (2.1)
\end{aligned}
$$

Predicates $InStock(prod, s)$ and $OrderedA(prod, s)$ are called *fluents* in the situation calculus terminology. Fluents are formally predicates with one explicit argument of type state. A fluent can be thought of as a database table, whose contents may change from state to state. In this example, fluent $InStock(prod, s)$ contains all products $prod$ that are in stock at state s. Likewise, fluent $OrderedA$ contains all products ordered to section A in state s.

In order to change the state of the database, some transaction must be performed. Let us assume the existence of two transactions, $order(prod)$ and $newItem(prod)$, which are formally first order terms, with the intended meaning "receive an order for product $prod$ (from some client[1])" and "add product $prod$ as a new item in stock". In the situation calculus, prior to the execution of any such transaction, certain preconditions must be satisfied, which are specified by means of *precondition axioms*. In the example, we informally stated above that it is possible to order a product if and only the product is in stock or has a provider. We assume it is always possible to add a new product to fluent $InStock$. The corresponding axioms read:

$$\begin{aligned} &(\forall p, s) Poss(order(prod), s) \equiv InStock(prod, s) \vee \\ &\quad (\exists prov)(Provider(prov, s) \wedge Provides(prov, prod, s)) \\ &(\forall prod, s) Poss(newItem(prod), s) \equiv true \end{aligned} \tag{2.2}$$

Predicate $Poss(a, s)$ is an element of every situation calculus language, whose meaning is "action a[2] is possible (legal, executable) at state s". In the example, predicates $Provider$ and $Provides$ are also fluents.

Clearly, it is necessary to somehow axiomatize the change produced by the actions on the state of the database, more precisely on the truth values of the fluents. *Successor State Axioms* specify the exact conditions under which a predicate (fluent) becomes true or false at state $do(a, s)$, for any action a. $do(a, s)$ denotes the state resulting from performing action a at state s, and is called a *successor state* (of state s). The specification contains one successor state axiom for every fluent.

[1] The example is oversimplified in 'management' terms so as to facilitate its use as an illustration for bounded quantification.

[2] We will from now on speak undistinctly of 'action' or 'transaction'. The situation calculus term is 'action', which in a database context is understood as a database transaction.

Successor state axioms (SSAs, for short) embody a solution to the Frame Problem [30].

In our example, the SSA for fluent $OrderedA$ reads:

$$\begin{aligned}&(\forall prod, s, a)(Poss(a, s) \supset (OrderedA(prod, do(a, s)) \equiv \\ &OrderedA(prod, s) \vee (a = order(prod)) \wedge \\ &(\neg\exists prov)(Provider(prov, s) \wedge Provides(prov, prod, s)))),\end{aligned} \tag{2.3}$$

that is, an arbitrary product $prod$ is ordered in section A in the successor state $do(a, s)$ if and only if product $prod$ was already ordered at section A in state s or the action a was to order it and no provider exists for this product. In a similar way, the successor state axiom for fluent $InStock$ is[3]:

$$\begin{aligned}&Poss(a, s) \supset (InStock(prod, do(a, s)) \equiv \\ &\quad InStock(prod, s) \vee (a = newItem(prod)))\end{aligned} \tag{2.4}$$

Notice that states, actions and objects of the domain coexist at the same object–level in the situation calculus. This fact allows first order quantifications over all of them.

Finally, static integrity constraints (we do not consider in this thesis dynamic ones) are, in the situation calculus, sentences of the form $(\forall s \geq S_0)\varphi(s)$, where $\varphi(s)$ is a formula that contains s as its only state term, and $s \geq S_0$ means that state s is reachable from state S_0 by means of a finite sequence of legal transactions (the definition is given below). For example, the following formula formalizes the above conjectured IC by expressing that at every reachable state of the database

[3]In rigor, fluents $Provider$ and $Provides$ have successor state axioms also, but we omit them here so as to keep the example within a reasonable size, moreover considering that those SSA are almost irrelevant to the rest of the example.

all products ordered in section A are in stock:

$$(\forall s \geq S_0, prod) OrderedA(prod, s) \supset InStock(prod, s) \tag{2.5}$$

In summary, the final specification contains: (a) For each named action $a(\vec{x})$ (basic transaction), one precondition axiom of the form $(\forall \vec{x}, s) Poss(a(\vec{x}), s) \equiv \varphi(\vec{x}, s)$; (b) For each fluent $F(\vec{x}, s)$ one successor state axiom of the form $(\forall \vec{x}, a, s) Poss(a, s) \supset F(\vec{x}, do(a, s)) \equiv \varphi(\vec{x}, a, s)$, where φ is a formula that contains s as the only state term; (c) Unique names axioms for states and actions[4]; (d) A specification of the initial database (at S_0), that is, for each fluent $F(\vec{x}, s)$, an axiom of the form $F(\vec{x}, S_0) \equiv \varphi(\vec{x})$, where φ specifies a set of tuples in the initial database; (e) A second order induction principle for states [33][5]

$$(\forall p)(p(S_0) \wedge (\forall a, s)(p(s) \supset p(do(a, s))) \supset (\forall s) p(s)) \tag{2.6}$$

and (f) The definition of the accessibility relation between states $\geq$. Often only the $s \geq S_0$ relation (accessibility from the initial state S_0) is needed, which we denote by $reach(s)$. Given the induction principle (2.6), a convenient definition for the $reach(s)$ predicate is:

$$\begin{aligned} & reach(S_0) \equiv true \\ & reach(do(a, s)) \equiv Poss(a, s) \wedge reach(s) \end{aligned} \tag{2.7}$$

ICs are not considered as part of the specification, but as (possible) entailments of the specification [33, 34, 22].

[4]For distinct action names A and A', $A(\vec{x}) \neq A'(\vec{y})$. For each action name A, the axiom $A(x_1, \ldots, x_n) = A(y_1, \ldots, y_n) \supset x_1 = y_1 \wedge \cdots \wedge x_n = y_n$. $S_0 \neq do(a, s)$. $do(a, s) = do(a', s') \supset a = a' \wedge s = s'$.

[5]Notice its similarity with the induction principle for natural numbers $(\forall p)(p(0) \wedge (\forall n)(p(n) \supset p(n+1)) \supset (\forall n) p(n))$.

III. PROVING DATABASE INTEGRITY CONSTRAINTS USING REWRITE SYSTEMS AND AUTOMATED INDUCTION

3.1 The Quantification Problem

We make more precise in this section the problem that was pointed out in the introduction and which is the main topic of this chapter, namely the lack of a general mechanism in conditional equational languages for the representation of arbitrary first–order quantifiers, which may appear in situation calculus specifications (as in any first–order specification).

Some technical notions about conditional term rewriting inductive theorem provers (CTRS) are necessary before a precise explanation of the problem. We will briefly sketch those notions in the following subsection.

3.1.1 Elements of Conditional Term Rewriting Inductive Theorem Proving

Term rewriting systems are theorem provers that are able to reason from *equational specifications*. An equational specification is a set of equalities of the form $lhs = rhs$, where lhs and rhs are first order terms whose variables are all implicitely universally quantified. An example of an equational specification is $\{x * e = x, x * i(x) = e, x * (y * z) = (x * y) * z\}$ (axioms of group theory). The basic idea in term rewriting systems is to transform the set of equalities into *oriented equalities*, or *rewrite rules*, that is, equalities in which one is allowed to transform the left hand side of the equality (possibly instantiated, or unified with some other term) into the right hand side (with the same instantiation or unification)[1], but not the other way around. For a survey on term rewriting systems, see for example [41].

[1] In the example, using the extended Knuth–Bendix completion procedure [20], the group theory axioms are transformed by RRL into the following set of rewrite rules [18]:
i(e)→e. i(i(x)) →x. (x*i(x)) →e. (x*e) →x. (i(x)*x) →e. (e*x) →x. f1(x) →x. (i(y)*(y*x)) →x. (x*(i(x)*y)) →y. i((y*x)) →(i(x)*i(y)). ((z*x)*y) →(z*(x*y)).

Conditional term rewrite systems, on the other hand, are able to reason from *conditional equational specifications*, which are sets of conditional equations of the form

$$l_1 \wedge \cdots \wedge l_n \supset (lhs = rhs) \tag{3.1}$$

where lhs and rhs are first order terms and $l_1 \dots l_n$ are literals. All variables are again implicitly universally quantified. CTRS provide in general first–order complete deductive systems on conditional equational specifications. An example of such first–order complete conditional rewriting deductive systems is described in [43, 45].

Conditional equations are suitable for our database updates reasoning problem because a large part of Reiter's specification can be straightforwardly translated into conditional equations. For example, equivalence (2.4) is easily represented as a (conditional) equation as:

$$InStock(prod, do(a, s)) = (InStock(prod, s) \vee a = newItem(prod)). \tag{3.2}$$

Here, both sides of the equation are first order terms, in this case of boolean type; and also the set $\{l_i\}$ of literals is empty.

Besides conditional equations, an (implicit) induction principle on states, the counterpart of Reiter's induction principle (2.6), is an element of equational situation calculus specifications. Many mechanisms have been proposed to deal with induction (in general): structural induction [11, 2], inductionless induction [21], Boyer and Moore's approach [8, 9, 26], the Cover Set Method [16, 46] and others. The latter two can be considered among the most successfull approaches, considering their many 'difficult' automatic proof performances such as for instance the proof of the chinese remainder theorem [47], the proof of Ramsey's theorem [44] or a proof of the unsolvability of the Halting Problem [7]. Both are based on rewriting, and we briefly explain them now, focusing on the Cover Set Method and stressing the points which are relevant to our quantification problem:

First, it should be noticed that the induction principle is usually not explicitely stated (e.g. as in (2.6)) when using CTRS. Rather, the CTRS user declares the symbols S_0 and *do* to be *free constructors* of the domain of states[2]. It can be shown that Reiter's induction principle, plus the Unique Names Axioms for states, is equivalent to this free constructor declaration.

Now, regarding the inductive proof process, the Cover Set Method, similarly to Boyer an Moore's approach, performs roughly the following steps:

– First, assuming that induction is to be performed in the proof process of a conjecture C, the Cover Set Method chooses, using appropriate heuristics, a term t from conjecture C. Suppose for instance that t is term $gcd(u,v)$[3].

– Then, the Cover Set Method considers the inductive definition of term t. In our example, the inductive definition for the *gcd* (boolean) function on integers, based on Euclid's algorithm, is:

$$\begin{aligned} &gcd(x,0) = x\\ &gcd(0,x) = x\\ &gcd(x,x+y) = gcd(x,y)\\ &gcd(x+y,x) = gcd(y,x) \end{aligned} \tag{3.3}$$

plus a specification of integer arithmetic $\{x+s(y) = s(x+y), x+0 = x\}$ in which 0 and the successor function s are declared as free constructors of the set of integers.

– From the above inductive definition, the Cover Set method derives a set of *inductive cases* for the proof of conjecture C, and in our example one inductive

[2] A set of *constructors* for a data type T is a finite set of function symbols $\{f_i\}$ such that every element in T is equal to some term made up with f_is only. The set $\{f_i\}$ is called a set of *free* constructors if all $f_i(x)$ are different from all $f_j(y)$ for $i \neq j$; and also $f_i(x)$ is different from $f_i(y)$ whenever $x \neq j$, for all x and y.

[3] $gcd(u,v)$ is the usual short-hand for the greatest common divisor of u and v.

case for each of the above equations[4]:

$$C|^{gcd(u,v)}_{gcd(u,0)} \tag{3.4}$$
$$C|^{gcd(u,v)}_{gcd(0,v)} \tag{3.5}$$
$$C \supset C|^{gcd(u,v)}_{gcd(u,u+v)} \tag{3.6}$$
$$C \supset C|^{gcd(u,v)}_{gcd(u+v,v)} \tag{3.7}$$

The Cover Set Method *proves* that every pair $< x, y >$ of integers is covered by some of the above cases, that is, that any pair $< x, y >$ of integers is equal to $< u, 0 >$, $< 0, v >$, $< u + v, u >$ or $< u, u + v >$ for some u and v. This fact (that all integer pairs are considered by (3.4), (3.5), (3.6) or (3.7)) is a proof of the soundness of the derivation of C (for all integers u, v) from (3.4) through (3.7).

3.1.2 The Quantification Problem

Now, returning to our main topic: the quantification problem. Recall that a large part of Reiter's database specification can be straightforwardly translated into (inductive) conditional equations (for example, equation (2.4) above). For some formulae, however, the translation process is more complicated: consider for example formula (2.3), defining $Poss(order(prod), s)$:

$$\begin{aligned}(\forall p, s) Poss(order(prod), s) \quad \equiv \quad & InStock(prod, s) \vee (\exists prov)(Provider(prov, s) \wedge \\ & Provides(prov, prod, s))\end{aligned}$$

The first idea that could come to the mind would be to express this formula with something like $Poss(order(prod, s)) =< formula >$; but, as mentioned earlier, there is no equivalent in CTRS for the existential quantifier symbols in (2.3)'s right hand subformula: all quantifiers are implicit and universal. There-

[4]We use throughout this thesis a general substitution notation, in which $\varphi|^{t}_{t'}$ denotes the formula that results from replacing all occurences of term t in φ by term t'. In the usual substitution notation, term t is allowed to be a variable only.

fore, $< formula >$ finds no straightforward expression in conditional equational languages.

A natural alternative idea would be then to skolemize (2.3), giving:

$$\begin{aligned}
&Poss(order(prod), s) \supset (InStock(prod, s) \vee Provider(f(prod), s) \wedge \\
&\qquad\qquad Provides(f(prod), prod, s)) \\
&(InStock(prod, s) \vee Provider(prov, s) \wedge Provides(prov, prod, s)) \supset \\
&\qquad\qquad Poss(order(prod), s)
\end{aligned} \tag{3.8}$$

where f is a Skolem function.

Three problems arise with skolemization in the context of rewrite systems:

a) Formulae (3.8) cannot (in general) be recognized by CTRS as a *complete definition* of $Poss(order(prod), s)$, in the sense that $Poss(order(pl), s)$ has a uniquely defined (truth) value for every pl and s. Comparing for example formulae (3.8) and equation (3.2), we see that (3.8) is of the form

$$p(\vec{u}) \supset \psi(\vec{u}) \tag{3.9}$$

$$\varphi(\vec{u}) \supset p(\vec{u}) \tag{3.10}$$

whereas (3.2) is of the form

$$p(\vec{u}) = \varphi(\vec{u})$$

In the latter case, the (truth) value of (boolean) function p is well defined for every $\vec{u}$, and moreover this value is equal to $\varphi(\vec{u})$. In the former case, a deductive process would be necessary in order to determine that (3.9) and (3.10) constitute a complete definition for boolean function p[5].

b) More important, the absence of recognizable complete definitions causes the Cover Set Method and similars to be unable to find *induction schemata* for

[5] For instance, it would be necessary to prove for that purpose that formulae φ and ψ are contradictory. The proof of the fact that a set of formulae is contradictory is in general undecidable.

the proof of integrity constraints. For example, the fact that (3.3) can be recognized as a complete definition of function *gcd* is fundamental, as seen above, for the derivation of inductive schema (3.4) through (3.7), and the proof of its soundness.

c) Finally, skolemized formulae such as (3.9) and (3.10), represented as conditional equations, tend to be 'purely conditional', in the sense that the strength of conditional rewriting systems, namely rewriting, is somehow lost. For example, a greater use of a rewriting theorem prover's capabilities is obviously made with specifications containing (3.2) than with specifications containing (3.9) and (3.10).

We propose in this chapter another alternative (besides skolemization) to the problem of representing quantifiers on logic specifications, which overcomes the above mentioned problems, and which is based on bounded quantification over finite domains. The solution is designed for situation calculus specifications and is suitable for use with conditional term rewriting systems, but can be extended to other (inductive) languages and kinds of theorem provers as well.

3.2 The BPSKS Methodology for Proving ICs

This section summarizes previous work presented in [6] on automated proofs of situation calculus integrity constraints using inductive conditional term rewrite systems; we also survey other papers related to the topic.

A general quantification mechanism, similar to that presented in [6] was developped by Boyer and Moore in [9, 10] for the NQTHM theorem prover that consists —roughly speaking— of a second order function whose parameters are: a list, an initial value and a first order function. The quantifying second order function goes through the list and computes the parameter function, starting with the initial value parameter. For example, if the list contains integers, the parameter function is the sum and the initial value is zero, then the second order 'quantifier' function computes the sum of the elements in a list. It is called a (general) quantifier since an interesting particular case arises when the list contains boolean values and the parameter (boolean) function is '$\wedge$' or '$\vee$'. In this particular case, the general quantifier behaves as usual universal and existential quantifiers for finite lists.

Bertossi et al's methodology [6] can be seen as a situation calculus adaptation and application of Boyer and Moore's bounded quantification ideas. Its first –rather simple– element is the declaration of the functions S_0 and *do* as free constructors of the domain of states. As mentioned earlier, the second order induction principle in Reiter's specification is a logical consequence of this constructor declaration. Also the action symbols are declared to be constructors of the data type *action*. In our example, action symbols *order* and *newItem* are declared as free constructors of data type *action* [6].

Second, the *reach* predicate on states from equation (2.7) is inductively defined as follows:

$$reach(S_0) = true \tag{3.11}$$

$$reach(do(a, s)) = (Poss(a, s) \wedge reach(s)) \tag{3.12}$$

Because of the implicit inductive definition of states (from the free constructor declaration), the satisfaction of the condition that *reach* is the minimum predicate satisfying (3.11) and (3.12) is obtained for free.

The third element presented in [6] is the quantifier representation methodology for CTRS, which we illustrate as follows.

Consider an arbitrary formula

$$(\exists x)\varphi(x, \vec{y}, s) \tag{3.13}$$

to be represented within a conditional equation. In the context of full first order

[6]This action type free constructor declaration implies by the way that every action in the data type is either an *order* action or a *newItem* action.

logic, the quantification on x may be performed over arbitrary domains, depending of the models of the considered theory, and in particular over infinite domains. In the context of (relational) databases, however, interpretations or models, and therefore domains, are usually finite: for instance, a well-known database concept it that of *active domain* [40], meaning the set of all objects that exist at a particular state of a database, which is obviously a finite set, since satabases are stored on (finite) computer hardware. The active domain is therefore a (finite set) function of the database state, and can be seen, in a relational context, as the union of the projections $\pi_A(R)$ for all relations R in the database schema and all the attributes A in R. Let us adapt this concept to the situation calculus by assuming that set $adom(s)$ represents the active domain at state s.

Given the existence of the $adom(s)$ active domain, the concept of *safe formula* can be defined as follows: a first–order formula $\varphi(x, \vec{y}, s)$ is *safe* on variable x if

$$(\forall x, \vec{y}, s)\varphi(x, \vec{y}, s) \supset x \in adom(s) \tag{3.14}$$

is a logical consequence of the database specification. We are again adapting the usual relational safe formula definition [40] to the situation calculus. The intuitive idea is that a formula is safe whenever it returns elements from the active domain only. This concept is fundamental in database theory, since unsafe queries (for instance $\neg r(x)$, where r is a database relation) are in general undesirable since it is not clear which domain these objects should belong to, so that the query can return an infinite number of objects in infinte domains. Instead, for example $p(x) \wedge \varphi(x)$ is a safe query, since it returns objects within relation p, all of which by definition belong to the active domain.

Given that the determination of whether an arbitrary formula φ is safe depends –rougly speaking– on (3.14)'s proof, the safeness property on formulae is undecidable, even in a relational context [40]. Sufficient syntactical safeness criteria for formulae have been defined, however. For instance, in [27], Nicolas defines the (syntactical) *range restricted property* on prenex conjunctive normal form formulae.

Range restricted formulae can be easily shown to be safe. For example, functional dependencies such as

$$(\forall s, x, y, z)(F(x, y, s) \wedge F(x, z, s) \supset y = z)$$

satisfy the range restricted syntactical safeness criterium[7] since all universally quantified variables appear in at least one negated atom. Further discussion on syntactical safeness criteria for formulae, and on alternative safeness definitions can be found in [36, 3, 12, 29, 15].

In this setting, assuming equation (3.14), the safeness of formula φ on variable x, it becomes possible to replace (3.13) by

$$(\exists x) x \in adom(s) \wedge \varphi(x, \vec{y}, s) \tag{3.15}$$

since variable x in φ returns values in $adom(s)$ only, and given this hypothesis, $(\exists x)\varphi(x, \vec{y}, s)$ is easily shown to be logically equivalent to $(\exists x) x \in adom(s) \wedge \varphi(x, \vec{y}, s)$.

Given the finiteness of the active domain $adom(s)$, it becomes possible to define a list function $bdom(s)$, where s is of type state, returning a finite list containing all the elements from the active domain $adom(s)$ at state s (and possibly some other irrelevant elements outside $adom(s)$). We give the axiomatization of function $bdom(s)$ further on. Assuming then the existence of list function $bdom(s)$ containing at least all elements from $adom(s)$, it becomes possible to define an exhaustive search within $bdom(s)$ that checks the truth of the above quantified formula $(\exists x)\varphi(x, \vec{y}, s)$, for any value of $\vec{y}$, and for the objects x in list $bdom(s)$, as follows:

First, consider $(\exists x) x \in bdom(s) \wedge \varphi(x, \vec{y}, s)$[8] instead of $(\exists x)\varphi(x, \vec{y}, s)$, assuming the safeness of φ and the inclusion of all elements in $adom(s)$ in list $bdom(s)$. Next, represent $(\exists x) x \in bdom(s) \wedge \varphi(x, \vec{y}, s)$ by boolean term $bexist(\vec{y}, s, bdom(s))$

[7] We omit the syntactical definition and a discussion of the range restricted formula property, for space's sake.

[8] We abuse a little the notation by using the $\in$ symbol for list membership, besides the usual set-membership meaning, whenever no confusion is possible.

(for <u>b</u>*ounded* <u>*exist*</u>) that has the same arguments as $(\exists x)\varphi(x, \vec{y}, s)$, namely $\vec{y}$ and s[9], plus a list argument, which is in this case set to list $bdom(s)$. The inductive definition of boolean function $bexist$ for an arbitrary list L reads:

$$\begin{aligned} &bexist(\vec{y}, s, []) = false \\ &bexist(\vec{y}, s, [x|L]) = (\varphi(x, \vec{y}, s) \vee bexist(\vec{y}, s, L)) \end{aligned} \tag{3.16}$$

This is nothing but the usual 'LISP-like' way of going through lists. We replace first order quantification by bounded quantification by recursively checking the property φ from the first element of the (domain) list on. It is clear that $bexist(\vec{y}, s, L)$ is a suitable representation for $(\exists x \in L)\varphi(x, \vec{y}, s)$.

Similar definitions can be given for bounded universal quantifiers in terms of a boolean function $ball$ by replacing formula $(\forall x)\psi(x, \vec{y}, s)$ by $(\forall x)(x \in bdom(s) \supset \psi(x, \vec{y}, s))$. The safeness definition in this case (universal quantification) is slightly different from the above existential quantifier safeness definition, in which $\varphi(x, \vec{y}, s)$ was required to return values for x within $adom(s)$ only. We require for universal quantification $\neg\psi$ (as opposed to ψ, as for existential quantification) to return values within $adom(s)$ only, that is, to be safe. The representation of universally quantified literals as bounded quantifications is analogous to the existential case: assuming the safeness hypothesis for formula $\neg\psi$, formula $(\forall x)\psi(x, \vec{y}, s)$ is represented as boolean term $ball(\vec{y}, s, bdom(s))$, inductively defined for an arbitrary list L as:

$$\begin{aligned} &ball(\vec{y}, s, []) = true \\ &ball(\vec{y}, s, [x|L]) = \varphi(x, \vec{y}, s) \wedge ball(\vec{y}, s, L). \end{aligned}$$

As an illustration for bounded quantification, consider formula (2.2) from the example in section 2, which is represented, using bounded quantification, by

[9] Variable x, being quantified, is not an argument of $(\exists x)\varphi(x, \vec{y}, s)$.

equation

$$Poss(order(prod), s) = InStock(prod, s) \vee bexist_1(prod, s, bdom(s)) \qquad (3.17)$$

where the axiomatization of the new symbol $bexist_1(prod, s, L)$ for an arbitrary domain list L is as follows:

$$bexist_1(prod, s, []) = false$$
$$bexist_1(prod, s, [x|L]) = bexist_1(prod, s, L) \vee$$
$$Provider(x, s) \wedge Provides(x, prod, s)$$

Similarly, $OrderedA$'s successor state axiom (2.3) is represented as

$$OrderedA(prod, do(a, s)) = OrderedA(prod, s) \vee (a = order(prod)) \wedge ball_2(prod, s, bdom(s))$$

where $ball_2(prod, s, L)$ is inductively defined by equations

$$ball_2(prod, s, [x|L]) = ball_2(prod, s, L) \wedge \qquad (3.18)$$
$$(\neg Provider(x, s) \vee \neg Provides(x, prod, s))$$
$$ball_2(prod, s, []) = true \qquad (3.19)$$

for an arbitrary list.

Should in general a quantified formula $(\forall x)\varphi(x, \vec{z}, s)$ contain, as a subformula, another quantified formula $(\forall y)\psi(x, y, \vec{z}, s)$ for instance, the bounded quantification method explained above would need to be recursively applied to ψ by replacing, within φ, $(\forall y)\psi(x, y, \vec{z}, s)$ by a new literal $ball_3(x, \vec{z}, s, bdom(s))$ and by giving an inductive definition for symbol $ball_3$ following (3.19).

The last remaining issue in the methodology presented in [6] is the inductive definition for the domain list function $bdom(s)$, which in this example reads:

$$bdom(S_0) = [p_1, p_2] \tag{3.20}$$

$$bdom(do(order(prod), s)) = [prod|bdom(s)] \tag{3.21}$$

$$bdom(do(newItem(prod, pl), s)) = [prod|bdom(s)] \tag{3.22}$$

that is, the objects in the active domain at state S_0 are those from equivalences (2.1); and objects are added to the active domain as they appear in actions[10]. The general $bdom(s)$ axiomatization (for any set of actions) will be given in section 3.4. The proof of this definition's soundness will be given in section 3.5. Several technical issues (such as the fact the objects appearing as action parameters may indeed be deleted (and not added) to the active domain, and such as the fact that objects may appear twice in the domain list) will be solved section 3.5 as part of the soundness proof.

In [6] the authors showed also the need for a set of inductive lemmata in the inductive CTRS proof process of situation calculus integrity using the above bounded quantification definition. Those inductive lemmata can be considered as "logical lemmata" in the sense that they capture part of the semantics of bounded quantification. They are of the form:

$$ball_{\varphi}(\vec{y}, s, bdom(s)) \supset \varphi(x, \vec{y}, s) \tag{3.23}$$

where $ball_{\varphi}(\vec{y}, s, bdom(s))$ represents $(\forall x)\varphi(x, \vec{y}, s)$[11]. This issue, in a wider setting, will be further illustrated and studied in detail in this chapter.

[10] Since we are omitting the axiomatization of the *Provider* fluent, for space's sake, we omitted also the specification of actions whose effect would be the eventual insertion of new providers into the *Provider* fluent. The parameters of such actions (the new providers) should be also added to list *bdom* in a bigger example, and the corresponding axioms should be included in the specification, besides (3.20), (3.21) and (3.22).

[11] The logical axiom for universal quantifiers $(\forall x)\varphi(x) \supset \varphi(y)$ is an element of Hilbert's logical system [13]. Notice its similarity with (3.23).

3.3 Eliminating Quantifiers

We propose in this section a method that allows the *elimination* of some quantifiers from Reiter's database updates specification, and in particular the existential quantifiers that are introduced into the specification by Reiter's transformation process of Effect Axioms into Successor State Axioms [30]. It is obvious that the quantification problem outlined in section 3.1 is solved for those quantifiers if they are somehow removed from the specification, but also, although we prove in section 3.6 the soundness of the bounded quantification method, the introduction of new *ball* and *bexist* symbols and their axiomatization into the specification clearly introduces an additional amount of complexity into any IC proof, and therefore it is always interesting to be able to eliminate some quantifiers from the specification before translating it using bounded quantification. We will recall Reiter's transformation process [30] and typify the quantifiers it introduces. We will then present a simple method that allows the elimination of those quantifiers from the SSAs specification.

Reiter's transformation of Effect Axioms (EA, for short) into SSAs is what constitutes in fact his solution to the Frame Problem [30]. The database applications defined in [33], which are the basis for this thesis, are an (important) subproduct of Reiter's solution to the Frame Problem, which can (the Frame Problem) be roughly stated as follows: given a set of Effect Axioms of the form

$$Poss(A(\vec{x}), s) \wedge \varphi(\vec{x}, \vec{y}, s) \supset [\neg]F(\vec{y}, do(A(\vec{x}), s)) \tag{3.24}$$

that specify *sufficient* conditions for change to occur in the truth values of fluent F (from state s to state $do(a, s)$), how can *necessary and sufficient* conditions for the change in fluent F be (efficiently) provided assuming the *Frame Hypothesis* that reads (roughly): "for any assignment σ of values to the variables of fluent F, it is assumed that $F(s)\sigma \equiv F(do(a, s))\sigma$, for every action a whenever this assumption is not contradictory with the effect axioms"?

It is clear that a succesor state axiom of the form

$$(\forall a, s, \vec{y}) Poss(a, s) \supset F(\vec{y}, do(a, s)) \equiv \varphi(\vec{y}, a, s)$$

is indeed a solution to the Frame Problem whenever it is consistent with the effect axioms and the Frame Assumption.

Example: Recalling the example from section 2, the SSA for fluent $InStock$ (equation (2.4)):

$$\begin{aligned}(\forall prod, a, s)(Poss(a, s) \supset (InStock(prod, do(a, s)) \equiv InStock(prod, s)\\ \vee\, a = newItem(prod)))\end{aligned}$$

is the result of the transformation of effect axiom

$$(\forall prod, s) InStock(prod, do(newItem(prod), s)) \tag{3.25}$$

using Reiter's transformation.

It is clear that, according to effect axiom (3.25), a sufficient condition for $InStock(prod, do(a, s))$ being true is $a = newItem(prod)$. According to the Frame Assumption, in any other case (i.e. if $a \neq newItem(prod)$), the truth value of fluent $InStock$ is the same at state s and at successor state $do(a, s)$. This set of hypothesis is compiled into successor state axiom (2.4).

Example: In the same way,

$$F(y, do(a, s)) \equiv (\exists x)(a = A(x, y) \wedge \varphi(x, y, s)) \vee F(y, s)$$

is the successor state axiom corresponding to effect axiom

$$\varphi(x, y, s) \supset F(y, do(A(x, y), s))$$

Notice that an example of the quantifier symbols that can be eliminated by this section's method (the ones introduced by the transformation of effect axioms into successor state axioms) is shown in the above example.

Rather than recalling the details in [30] from Reiter's transformation, we will only point out what is relevant for this section, namely:

Lemma 1. If Σ_{EA} is a set of effect axioms of the form (3.24); and if Σ_{SSA} is the set of successor state axioms derived from Σ_{EA} by Reiter's transformation [30], then: A) There exists an isomorphism between the set Q_{EA} of quantifiers symbols in Σ_{EA} and a subset Q' of the set Q_{SSA} of quantifier symbols in Σ_{SSA}. B) All quantifiers in $Q_{SSA} - Q'$ are existential and the variables they quantify are action parameters.

(Observation: the quantifiers being eliminated by this section's method are precisely those in the set $Q_{SSA} - Q'$).

Proof: We make explicit Reiter's transformation steps in [30] for an arbitrary set of effect axioms. We conclude the lemma from those steps' explicitation (which does not appear in [30]: the following steps are implicitly understood in that paper).

The transformation starts, as said earlier, with effect axioms of the form (3.24):

$$Poss(A_i(\vec{x}), s) \wedge \varphi(\vec{x}, \vec{y}, s) \supset [\neg]F(\vec{y}, do(A_i(\vec{x}), s))$$

possibly with 0 or more axioms for each action symbol A_i or for each fluent F and also possibly with more than one axiom for a combination of a fluent F and an action symbol A_i. The elements in vectors $\vec{x}$ and $\vec{y}$ are constants or variables, possibly repeated within each vector or between both vectors.

The first step of the transformation consists in rewriting (3.24) into the logically equivalent axiom:

$$Poss(a,s) \wedge a = A_i(\vec{x}) \wedge \varphi(\vec{x},\vec{y},s) \supset [\neg]F(\vec{y},do(a,s)) \tag{3.26}$$

This axiom is again logically equivalent to:

$$Poss(a,s) \wedge (\exists\vec{z})(a = A_i(\vec{x}) \wedge \varphi(\vec{x},\vec{y},s)) \supset [\neg]F(\vec{y},do(a,s)) \tag{3.27}$$

where the variables in $\vec{z}$ are those in $\vec{x}$ that are not in $\vec{y}$.

Next, recall that terms in $\vec{y}$ may be constants or repeated variables. Let $F(\vec{w},do(a,s))$ be a literal such that all of the variables in $\vec{w}$ are different, with no constant in it. Let $\frac{\vec{w}}{\vec{y}}$ be a conjunction of equalities that represents a substitution σ such that $F(\vec{w},do(a,s))\sigma$ is the same term as $F(\vec{y},do(a,s))$[12]. Then (3.27) is replaced by the logically equivalent axiom:

$$Poss(a,s) \wedge \frac{\vec{w}}{\vec{y}} \wedge (\exists\vec{z})(a = A_i(\vec{x}) \wedge \varphi(\vec{x},\vec{y},s)) \supset [\neg]F(\vec{w},do(a,s)) \tag{3.28}$$

Next, the positive effect axioms for each fluent are gathered into one single axiom, of the form:

$$Poss(a,s) \wedge (\bigvee_{j=1}^{n} \frac{\vec{w}}{\vec{y}_j} \wedge (\exists\vec{z}_j)a = A_i(\vec{x}_j) \wedge \varphi(\vec{x}_j,\vec{y}_j,s)) \supset F(\vec{w},do(a,s)) \tag{3.29}$$

n is the number of positive effect axioms for fluent F. $\vec{x}_j$ is the vector $\vec{x}$ of the j-th positive effect axiom. Idem for $\vec{z}_j$ and $\vec{y}_j$. Again logical equivalence is

[12]For instance, if $\vec{w}$ is $< x,y,z >$ and $\vec{y}$ is $< a,u,u >$, then $\frac{\vec{w}}{\vec{y}}$ is $(x=a) \wedge (y=u) \wedge (z=u)$. u is a variable and a is a constant.

preserved.

The same is done for the negative axioms, getting:

$$Poss(a,s) \wedge (\bigvee_{j=1}^{m} \frac{\vec{w}}{\vec{y}_j} \wedge (\exists \vec{z}_j) a = A_i(\vec{x}_j) \wedge \varphi(\vec{x}_j, \vec{y}_j, s)) \supset \neg F(\vec{w}, do(a,s)) \quad (3.30)$$

Call (3.29)

$$Poss(a,s) \wedge \gamma^{+}(a, \vec{w}, s) \supset F(\vec{w}, do(a,s))$$

and call (3.30)

$$Poss(a,s) \wedge \gamma^{-}(a, \vec{w}, s) \supset \neg F(\vec{w}, do(a,s))$$

Reiter then applies a minimization process (see [30] for details), getting the following successor state axiom:

$$Poss(a,s) \supset F(\vec{w}, do(a,s)) \equiv \gamma^{+}(a, \vec{w}, s) \vee F(\vec{w}, s) \wedge \neg\gamma^{-}(a, \vec{w}, s) \quad (3.31)$$

This formula, (3.31), is the final result of Reiter's algorithm. The lemma follows from the following observations:

- The φs in (3.29)+(3.30) and those in (3.24) are the same.
- The additional quantifiers in (3.29)+(3.30) are existential (as expressed in the theorem) and its variables, the $\vec{z}_j$s, are subsets of the $\vec{x}_j$s (as expressed above), which are action parameters.

■

We present now a simple way of eliminating the quantifiers from the above defined set $Q_{SSA} - Q'$. We will in general apply the bounded quantification method only on the quantifiers that remain *after* the application of the following procedure (if any)[13]:

Recalling chapter 2, the Unique Names Axioms state that for every action symbol $A(\vec{x})$ and for every parameter vectors $\vec{x}$ and $\vec{y}$, if $A(\vec{x}) = A(\vec{y})$ then all x_i in vector $\vec{x}$ are equal to the corresponding y_i in $\vec{y}$. This is basically the same as stating the existence of functions f_i such that $x_i = f_i(a)$, whenever $a = A(\vec{x})$, for every x_i in $\vec{x}$. Or, in a more "mathematical" language, that equation $a = A(\vec{x})$ has at most one solution for every x_i in $\vec{x}$, which can be called $f_i(a)$. Notice that in this case f_i does not depend on the x_j for $i \neq j$, because all x_j themselves are functions of a. With this in mind, it becomes easy to eliminate the quantifiers we are interested in. Retaking example 2 above:

$$F(y, do(a, s)) \equiv (\exists x)(a = A(x, y) \wedge \varphi(x, y, s)) \vee F(y, s) \tag{3.32}$$

in which action A has two parameters, x and y, two functions f_x and f_y can be defined and axiomatized as follows:

$$f_x(A(x, y)) = x$$
$$f_y(A(x, y)) = y$$

(given the Unique Names Axioms for action function symbol A). Notice that these equations are of course expressible in an equational language. Formula (3.32) can be then rewritten as:

$$F(y, do(a, s)) \equiv a = A(f_x(a), y) \wedge \varphi(f_x(a), y, s) \vee F(y, s)$$

and this last formula has a direct translation into equational languages by represent-

[13]And after the elimination of trivial quantifications, such as $(\exists x)(x = x)$, but since the set of trivial quantifications is undecidable, only partial solutions to this problem are possible such as the use for example of a finite list of some frequent trivial quantifications.

ing both sides of the equivalence as boolean terms and the equivalence itself as a term equality. It can be easily seen that the above idea of defining a function for every argument of every action symbol (that exists because of UNA) will eliminate all existential quantifiers introduced by Reiter's algorithm, leaving the quantifiers already present in the effect axioms only, which was the goal of this section, as expressed in its initial paragraph.

3.3.1 Generalization and Formalization of the Method

We formalize in this subsection the above informally stated quantifier elimination method, so as to prove its soundness. We first make a simple generalization of the method that allows among other goals the simplification of the soundness theorem's proof.

We generalize subformula $a = A(x, y) \wedge \varphi(x, y, s)$ in (3.32) to an arbitrary conjunction $\psi(x, \vec{y}) \wedge \varphi(x, \vec{y})$. Correspondingly, we generalize the Unique Names Axiom $A(x_1, y_1) = A(x_2, y_2) \supset x_1 = x_2$, which can be also expressed as

$$a = A(x_1, y) \wedge a = A(x_2, y) \supset x_1 = x_2$$

to:

$$\psi(x_1, \vec{y}) \wedge \psi(x_2, \vec{y}) \supset x_1 = x_2 \tag{3.33}$$

and we prove the following (more general)[14] result for the method informally presented in the previous subsection:

Theorem 1. If $(\exists x)\psi(x, \vec{y})$ is a subformula of an arbitrary first-order formula χ; if $\Sigma \cup \{\chi\} \models (\forall x_1, x_2, \vec{y})(\psi(x_1, \vec{y}) \wedge \psi(x_2, \vec{y}) \supset x_1 = x_2)$; and if the symbol f does not

[14]In the sense that it holds for an arbitrary formula ψ instead as for a formula of the form $a = A(\vec{u})$ only.

appear in Σ or χ, then $\Sigma \cup \{\chi\}$ is equiconsistent with

$$\Sigma \cup \{\chi|^{(\exists x)\psi(x,\vec{y})\wedge\varphi(x,\vec{y})}_{\psi(f(\vec{y}),\vec{y})\wedge\varphi(f(\vec{y}),\vec{y})}, (\forall x, \vec{y})(\psi(x, \vec{y}) \supset x = f(\vec{y}))\}$$

■

(The latter formula, $(\forall x, \vec{y})(\psi(x, \vec{y}) \supset x = f(\vec{y}))$, can be seen as the axiomatization of function symbol f as "ψ's inverse function".)

Example According to the theorem, an arbitrary specification $\Sigma\cup\{\neg(\exists x)\psi(x, \vec{y})\}$ such that $\Sigma\cup\{\neg(\exists x)\psi(x, \vec{y})\} \models (\forall x_1, x_2, \vec{y})(\psi(x_1, \vec{y})\wedge\psi(x_2, \vec{y}) \supset x_1 = x_2)$ is equiconsistent with $\Sigma \cup \{\neg(\exists x)\psi(f(\vec{y}), \vec{y}), (\forall x, \vec{y})(\psi(x, \vec{y}) \supset x = f(\vec{y}))\}$.

Our "skolemization procedure" is therefore a particular case of usual skolemization in which an additional hypothesis, namely in this example $\Sigma\cup\{\neg(\exists x)\psi(x, \vec{y})\} \models (\forall x_1, x_2, \vec{y})(\psi(x_1, \vec{y}) \wedge \psi(x_2, \vec{y}) \supset x_1 = x_2)$, is required. Additional advantages are obtained from this extra hypothesis, such as the possibility of skolemization within universal quantifications as seen in the example, besides the usual existential ones, when using ordinary skolemization.

Proof (of theorem 1)**:** We first prove that

$$\Sigma \cup \{\chi|^{(\exists x)\psi(x,\vec{y})\wedge\varphi(x,\vec{y})}_{\psi(f(\vec{y}),\vec{y})\wedge\varphi(f(\vec{y}),\vec{y})}, (\forall x, \vec{y})(\psi(x, \vec{y}) \supset x = f(\vec{y}))\} \models \Sigma \cup \{\chi\}$$

From $(\forall x, \vec{y})(\psi(x, \vec{y}) \supset x = f(\vec{y}))$ the following formula is easily derived: $(\exists x)(\psi(x, \vec{y}) \wedge \varphi(x, \vec{y})) \supset (\psi(f(\vec{y}), \vec{y}) \wedge \varphi(f(\vec{y}), \vec{y}))$. Since $(\psi(f(\vec{y}), \vec{y}) \wedge \varphi(f(\vec{y}), \vec{y})) \supset (\exists x)(\psi(x, \vec{y}) \wedge \varphi(x, \vec{y}))$ is tautologic, we have that:

$$(\forall x, \vec{y})(\psi(x, \vec{y}) \supset x = f(\vec{y})) \models (\exists x)(\psi(x, \vec{y}) \wedge \varphi(x, \vec{y})) \equiv (\psi(f(\vec{y}), \vec{y}) \wedge \varphi(f(\vec{y}), \vec{y}))$$

and given formula $(\exists x)(\psi(x, \vec{y}) \wedge \varphi(x, \vec{y})) \equiv (\psi(f(\vec{y}), \vec{y}) \wedge \varphi(f(\vec{y}), \vec{y}))$ it becomes possible to replace $\psi(f(\vec{y}), \vec{y}) \wedge \varphi(f(\vec{y}), \vec{y})$ by $(\exists x)\psi(x, \vec{y}) \wedge \varphi(x, \vec{y})$ within

$$\chi|^{(\exists x)\psi(x,\vec{y})\wedge\varphi(x,\vec{y})}_{\psi(f(\vec{y}),\vec{y})\wedge\varphi(f(\vec{y}),\vec{y})}$$

getting χ itself. This ends the first part of the proof.

We now prove that if $\Sigma \cup \{\chi\}$ has a model $\mathcal{M}$ that is also a model of $(\forall x_1, x_2, \vec{y})(\psi(x_1, \vec{y}) \supset \psi(x_2, \vec{y}))$, then a model $\mathcal{M}_1$ of

$$\Sigma \cup \{\chi|^{(\exists x)\psi(x,\vec{y}) \wedge \varphi(x,\vec{y})}_{\psi(f(\vec{y}),\vec{y}) \wedge \varphi(f(\vec{y}),\vec{y})}, (\forall x, \vec{y})(\psi(x, \vec{y}) \supset x = f(\vec{y}))\}$$

exists, if symbol f does not appear in $\Sigma \cup \{\chi\}$.

Let the interpretation of all the symbols in $\mathcal{M}$ be the same in $\mathcal{M}_1$. Let the interpretation of f in $\mathcal{M}_1$ be such that, for every $\vec{y}$, if $\mathcal{M} \models (\exists x)\psi(x, \vec{y})$, then $\mathcal{M}_1 \models \psi(f(\vec{y}), \vec{y})$. This is possible given the axiom of choice. This condition on f can be expressed as:

$$(\forall x, \vec{y})(\psi(x, \vec{y}) \supset \psi(f(\vec{y}), \vec{y})) \tag{3.34}$$

We will first prove that

$$\mathcal{M}_1 \models (\forall x, \vec{y})(\psi(x, \vec{y}) \supset x = f(\vec{y})) \tag{3.35}$$

We have that $\mathcal{M}$, and therefore $\mathcal{M}_1$ are models of $(\forall x_1, x_2, \vec{y})(\psi(x_1, \vec{y}) \wedge \psi(x_2, \vec{y}) \supset x_1 = x_2)$. From this and (3.34), (3.35) follows.

Given (3.35) we have easily that $\mathcal{M}_1 \models (\forall \vec{y})((\exists x)\psi(x, \vec{y}) \equiv \psi(f(\vec{y}), \vec{y}))$, and from this, we can replace χ by

$$\chi|^{(\exists x)\psi(x,\vec{y}) \wedge \varphi(x,\vec{y})}_{\psi(f(\vec{y}),\vec{y}) \wedge \varphi(f(\vec{y}),\vec{y})}$$

in Σ, and still have $\mathcal{M}_1 \models \Sigma$. This ends the second part of the proof and the proof itself.

■

Lemma 2. In the situation calculus, every formula χ containing a subformula of the form $(\exists x_1,...,x_n)(a = A(x_1,...,x_n,\vec{z}) \wedge \varphi(x_1,...,x_n,\vec{y},a,s))$, where φ is an arbitrary subformula within χ, can be replaced within a set Σ of formulae containing the Unique Names Axiom for function symbol A by

$$\varphi|^{(\exists x_1,...,x_n)(a=A(x_1,...,x_n,\vec{z})\wedge\varphi(x_1,...,x_n,\vec{y},a,s))}_{a=A(f_{x_1}(a),...,f_{x_n}(a),\vec{z})\wedge\varphi(f_{x_1}(a),...,f_{x_n}(a),\vec{y},a,s)}$$

Also, the following axiomatization for the $f_{x_i}(a)$ must be added to Σ whenever the former is not a logical consequence of the latter:

$$f_{x_i}(A(x_1,...,x_n)) = x_i$$

for every $i = 1..n$. The resulting set of formulae is equiconsistent with the original Σ.

Proof: The Unique Names Axioms for symbol A is $(\forall x_1...x_n,\vec{z},y_1...y_n,\vec{w})(A(x_1...x_n,\vec{z}) = A(y_1...y_n,\vec{w}) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n)$, from which the hypothesis of theorem 1 are verified for formula ψ (in theorem 1) being '$a = A(x_1...x_n,\vec{y})$', therefore getting the lemma.

■

Theorem 2. In a situation calculus database specification obtained from effect axioms by means of Reiter's solution to the Frame Problem all existential quantifiers introduced into the successor state axioms by this solution can be eliminated by replacing the quantified variable by a function symbol. The resulting specification is equiconsistent with the original one.

■

Proof: According to lemma 1, the hypothesis for lemma 2 are verified in Reiter's database updates specification.

■

The utility of the above equiconsistency results is of course that the fact that two specifications $\Sigma \cup \varphi$ and $\Sigma' \cup \varphi'$ are equiconsistent is equivalent to thet fact that $\Sigma \models \varphi$ if and only if $\Sigma' \models \varphi'$ and therefore proofs using specification Σ can be reproduced using Σ', which is generally a simpler specification.

3.3.2 Semantical elimination of quantifiers

We apply in this subsection the ideas presented in the previous subsection to an interesting particular case that eventually allows the elimination of further quantifiers from the specification:

In ordinary database applications, all table fields (in our case, all fluent parameters) are usually functionally dependant on a small subset of those table fields, which constitute the primary key of the table. This functional dependency property for an arbitrary (2-argument, for simplicity sake) fluent $F(x, y, s)$ can be expressed as

$$(\forall x, y_1, y_2, s)F(x, y_1, s) \wedge F(x, y_2, s) \wedge s \geq S_0 \supset y_1 = y_2 \tag{3.36}$$

considering a functional dependency in the direction $x \to y$, that is, considering y to be functionally dependent on x in F. This functional dependency is in itself an integrity constraint (a logical consequence of the database specification)[15]. We will however make abstraction of (3.36)'s proof process and assume that (3.36) is somehow known to or proven to be a logical consequence of the specification, that is, that y is somehow known to functionally depend on x in fluent F.

Then, directly applying theorem 1 from the previous subsection, we have that, for any formula $\varphi(x, y, \vec{z}, s)$, it is always possible to replace:

$$(\exists y)F(x, y, s) \wedge \varphi(x, y, \vec{z}, s)$$

[15]This IC should therefore in rigor be proved prior to the application of the method we are now suggesting, maybe using the bounded quantification method from the other sections.

by:

$$F(x, f(x,s), s) \wedge \varphi(x, f(x,s), \vec{z}, s)$$

within Σ of within a formula from Σ that contains $(\exists y)F(x,y,s) \wedge \varphi(x,y,\vec{z},s)$ as a subformula, provided that the above functional dependency property (3.36) is known to hold or has been proven to hold. Also, symbol f is, as stated above, a new function symbol not appearing elsewhere; and axiom

$$F(x,y,s) \supset F(x, f(y,s), s)$$

must be added to Reiter's specification. The interesting feature of this kind of skolemization is, as stated above, that it can be performed regardless of the quantifier scope of the subformula being skolemized, that is, this existentially quantified subformula does not need to appear in a universal quantification scope only.

In general, any quantified argument of a fluent that functionally depends on the primary key of that fluent (if the primary key exists) can be replaced by a function symbol, and the axiomatization for this function symbol must be added to the specification. This process eliminates one further quantifier from Reiter's specification, thus diminishing the proof overhead introduced by bounded quantification and maybe augmenting the proof's success probability[16].

3.4 Formalization of Bounded Quantification

We formalize in this section the transformation of an arbitrary situation calculus specification into a conditional equational specification by means of bounded quantification. We start by formally defining what a situation calculus specification is. We next formally define the bounded quantification transformation.

Definition 1. A situation calculus language is a sorted first–order language whose sorts are: the *action* sort, the *state* sort and the *object* sort. Two particular predicate symbols are defined in a situation calculus language: predicate symbol $Poss$, which has two arguments, the first of sort action and the other of sort state; and predicate

[16]Keeping in mind that the proof itself is an undecidable problem.

symbol is $\leq$, with two arguments of sort state. All other predicate symbols in the language have at most one argument of sort state (usually the last argument). They have no argument of sort action and they can have an arbitrary (possibly zero) finite number of arguments of sort object. Predicate symbols with one state argument are called *fluents*. Predicate symbols with no state argument are simply called predicates. A finite set of function symbols of sort *action* is defined, which we call $\mathcal{A}$. All parameters of all function symbols in $\mathcal{A}$ are of sort *object*.

Given the above definition of situation calculus language, we now define a situation calculus specification.

Definition 2. A situation calculus specification Σ is a finite set of formulae from a situation calculus language, which consists of the union of the following non–intersecting subsets:

– the set of successor state axioms $\mathcal{D}_{SSA}$, containing only, for every fluent symbol F in the language, one formula of the form:

$$(\forall \vec{x}, a, s) Poss(a, s) \supset F(\vec{x}, do(a, s)) \equiv \varphi(\vec{x}, a, s)$$

– the set of initial database axioms $\mathcal{D}_{S_0}$, containing only, for every fluent symbol F in the language, one formula of the form:

$$(\forall \vec{x}) F(\vec{x}, S_0) \equiv \varphi(\vec{x})$$

– the set of action precondition axioms $\mathcal{D}_{APA}$, containing only, for every action symbol $A \in \mathcal{A}$ one axiom of the form:

$$(\forall \vec{x})(\forall s) Poss(A(\vec{x}), s) \equiv \varphi(\vec{x}, s)$$

– $\mathcal{D}_{IP}$, containing one single second order formula, the induction principle IP for states

$$(\forall p)\{p(S_0) \wedge [(\forall a, s)(p(s) \supset p(do(a, s)))] \supset (\forall s) p(s)\}$$

– the set of unique names axioms $\mathcal{D}_{UNA}$, containing only, for each pair of distinct action symbols $A, B \in \mathcal{A}$ the axiom

$$(\forall \vec{x}, \vec{y})A(\vec{x}) \neq B(\vec{y});$$

for each action symbol $A \in \mathcal{A}$ the axiom

$$(\forall x_1 ... x_n, y_1 ... y_n)A(x_1, ..., x_n) = A(y_1, ..., y_n) \supset x_1 = y_1 \wedge ... \wedge x_n = y_n;$$

and the following unique names axioms for states

$$(\forall a, s)do(a, s) \neq S_0$$
$$(\forall a_1, a_2, s_1, s_2)do(a_1, s_1) = do(a_2, s_2) \supset (a_1 = a_2 \wedge s_1 = s_2)$$

– the predicates definition axiom set $\mathcal{D}_P$, containing only, for every predicate symbol P in the language (recall it has no state argument), one formula of the form:

$$(\forall \vec{x})P(\vec{x}) \equiv \varphi(\vec{x})$$

– and the definition of the $\leq$ predicate, which contains only the following formulae:

$$(\forall s)s \leq s$$
$$(\forall s_1, s_2, a)s_1 \leq do(a, s_2) \equiv Poss(a, s_2) \wedge s_1 \leq s_2$$
$$(\forall a, s)\neg do(a, s) \leq S_0$$

All quantifications within the above φs in $\mathcal{D}_{SSA}$ and $\mathcal{D}_{APA}$ are first-order quantifications on variables of sort object. Also, all variables in all vectors $\vec{x}$ above are distinct, and we call vector $\vec{x}$ in $\mathcal{D}_{SSA}$, $\mathcal{D}_{APA}$, $\mathcal{D}_{S_0}$ and $\mathcal{D}_P$'s formulae the *prefix* of those formulae.

■

We now give the formal definition of the bounded quantification transformation from Σ into a conditional equational specification Σ_{equ} by means of an algorithm we call *btrans*. We call Σ_{equ}'s language an *equational situation calculus language*, which we define as follows:

Definition 3. An equational situation calculus language is a situation calculus language, with an additional *domain* sort. Its predicates are the same situation calculus language predicates ($Poss$, $\leq$, fluents and predicates) plus a finite set of *ball* and *bexist* predicate symbols, which have one argument of sort state and one argument of sort domain. They have no argument of sort action and they can have an arbitrary finite number of arguments of sort object.

Definition 4. Transformation *btrans* is as follows:

a) First, let Σ_{equ} be equal to Σ, the original situation calculus specification, plus another two set of formulae Σ_{BD} (for 'bexist definitions'), initially empty, and Σ_{DD} (for 'domain definitions').

b) Evaluate formulae φ in $\mathcal{D}_{S_0}$ and $\mathcal{D}_P$ to sets S_F and S_P of tuples and replace the φs in $\mathcal{D}_P$ and $\mathcal{D}_{S_0}$ by disjunctions of tuple equalities, obtaining formulae of the form

$$F(x_1, ..., x_n, S_0) \equiv \bigvee_{<o_1,...,o_n>\in S_F} x_1 = o_1 \wedge ... \wedge x_n = o_n$$

and of the form

$$P(x_1, ..., x_m) \equiv \bigvee_{<o_1,...,o_m>\in S_P} x_1 = o_1 \wedge ... \wedge x_m = o_m$$

(it is assumed that the φs in $\mathcal{D}_{S_0}$ and in $\mathcal{D}_P$ represent database relations, so that they can be evaluated to sets of tuples).

c) Σ_{DD} consists, for every action symbol $A \in \mathcal{A}$, of formula

$$bdom(do(A_i(x_1, x_2, ..., x_m), s)) = [x_1|[x_2|...[x_m|bdom(s)]...]] \qquad (3.37)$$

and of the single formula

$$bdom(S_0) = bdom_{S_0} \tag{3.38}$$

where $bdom_{S_0}$ is a domain list containing all constants appearing in tuples from all of the above evaluated S_P and S_F sets.

d) Repeat the following steps:

i) Pick up an arbitrary subformula $(\forall x)\varphi(x, \vec{y}, s)$ or $(\exists x)\varphi(x, \vec{y}, s)$ of an arbitrary formula ψ in $\mathcal{D}_{SSA}$ or $\mathcal{D}_{APA}$ such that variable x does not belong to the ψ's prefix, and such that $\varphi(x, \vec{y}, s)$ contains no quantifier symbol. If no such subformula exists, exit the loop.

ii) If the chosen subformula is universally quantified, choose in the language an arbitrary unused *ball* symbol, $ball_\varphi$, and add the two formulae

$$\begin{aligned}&(\forall \vec{y}, s) ball_\varphi(\vec{y}, s, []) \equiv true\\ &(\forall \vec{y}, s, x, dom) ball_\varphi(\vec{y}, s, [x|dom]) \equiv \varphi(x, \vec{y}, s) \land ball_\varphi(\vec{y}, s, dom))\end{aligned} \tag{3.39}$$

to Σ_{BD}. If the chosen subformula is existentially quantified, choose in the language an arbitrary unused *bexist* symbol, $bexist_\varphi$ and add the two formulae

$$\begin{aligned}&(\forall \vec{y}, s) bexist_\varphi(\vec{y}, s, []) \equiv false\\ &(\forall \vec{y}, s, x, dom) bexist_\varphi(\vec{y}, s, [x|dom]) \equiv \varphi(x, \vec{y}, s) \lor bexist_\varphi(\vec{y}, s, dom)\end{aligned} \tag{3.40}$$

to Σ_{BD}.

iii) Replace within ψ (in Σ_{equ}) subformula $(\forall x)\varphi(x, \vec{y}, s)$ (respectively $(\exists x)\varphi(x, \vec{y}, s)$) by atom $ball_\varphi(\vec{y}, s, bdom(s))$ (respectively $bexist_\varphi(\vec{y}, s, bdom(s))$).

We note that no transformation is required for sets $\mathcal{D}_{UNA}$ and $\mathcal{D}_{IP}$, since those sets of formulae follow from the declaration of *do* and S_0 as free constructors for sort *state* and of the $A \in \mathcal{A}$ as free constructors for sort *action*. The following result is easily shown by induction on formulae:

Theorem 3. As a result of transformation *btrans*, all formulae in Σ_{equ} are of the form $lhs \equiv rhs$, where lhs is an atom and rhs is a first order formula containing no quantifier symbol. Moreover, all variables in lhs are distinct and universally quantified.

■

As a corollary of the theorem, since Σ_{equ} contains only equivalences of an atom and a formula with no quantifier, those equivalences can be trivially transformed into equations by considering its left and right hand sides as boolean terms. We abuse a little from language by making no distinction between Σ_{equ} considered as a set of first order equivalences and Σ_{equ} considered as a set of first order (boolean term) equalities.

3.5 Safeness Criteria on Situation Calculus Specifications

We devote this section to the definition of sufficient safeness criteria on Reiter's database updates specification formulae that guarantee the soundness of bounded quantification. We argue that in database contexts, where interpretations are usually finite, those sufficient conditions are reasonable.

Recalling section 3.2, an arbitrary existentially quantified formula $(\exists x)\varphi(x, \vec{y}, s)$ is roughly speaking replaced, when using bounded quantification, by

$$(\exists x) x \in bdom(s) \wedge \varphi(x, \vec{y}, s) \tag{3.41}$$

where $bdom(s)$ is the bounded quantification domain at state s. (3.41) is then translated into a conditional equational language by representing $bdom(s)$ as a finite list. Clearly, not every possible quantified subformula can be soundly translated using this method, since first order quantifications are defined in general w.r.t. arbitrary quantification domains (possibly infinite ones).

A sufficient hypothesis for the soundness of the above transformation is the *bounded quantification safeness hypothesis* which we define as follows:

$$(\forall \vec{y}, s, x)\varphi(x, \vec{y}, s) \supset x \in bdom(s) \tag{3.42}$$

since given (3.42), formulae $(\exists x)\varphi(x, \vec{y}, s)$ and $(\exists x)x \in bdom(s) \wedge \varphi(x, \vec{y}, s)$ are easily shown to be logically equivalent.

(3.42) follows from our situation calculus adaptation (3.14) of the usual safeness hypothesis and from the inclusion of the active domain in the bounded domain $bdom(s)$ for every reachable state s[17]:

$$(\forall s \geq S_0)adom(s) \subseteq bdom(s) \tag{3.43}$$

We will therefore require two hypothesis on the situation calculus database updates specification in order for bounded quantification to be sound: first, the safeness on variable x of formula φ for every existential quantification $(\exists x)\varphi$ and every universal quantification $(\forall x)\neg\varphi$ that need to be translated into bounded quantification terms. This hypothesis is reasonable, since most database formulae, being for example generated by means of relational operators, are safe. Moreover, as seen in section 3.2, many syntactical (and therefore decidable) criteria have been defined that guarantee formula safeness.

The second hypothesis, (3.43), is guaranteed whenever the database updates specification successor state axioms satisfy the *safeness hypothesis for successor state axioms*, which we define as follows:

Definition 5. A successor state axiom $F(x, \vec{y}, do(a, s)) \equiv \varphi(x, \vec{y}, a, s)$ for an arbi-

[17] Again, abusing a little the notation by relating a list to a set by means of symbol $\subseteq$.

trary fluent F from the database updates specification is *safe* whenever formula

$$\varphi(x, \vec{y}, A(x_1, ..., x_n), s) \land x \neq x_1 \land ... \land x \neq x_n \tag{3.44}$$

satisfies the safeness hypothesis (3.14), for every action symbol $A \in \mathcal{A}$.

Intuitively, we require φ in F's SSA to be safe, that is, not to return arbitrary objects outside the active domain, with the exception of the objects mentioned in the action symbols themselves, which can in general be added to fluents. Again, this safeness hypothesis on successor state axioms is reasonable, sice unsafe sucessor state axioms such as for example $F(x, do(a, s)) \equiv \neg F(x, s)$ or $F(x, do(a, s)) \equiv x > 4$ would imply infinite fluents (within infinite domains), making it impossible not only bounded quantification but also the storage of such fluents on computer hardware. We prove the following result:

Theorem 4. If all successor state axioms in $\mathcal{D}_{SSA}$ satisfy the SSA safeness hypothesis (3.44) and if the bounded domain list $bdom(s)$ is constructed according to equations (3.38) and (3.37), then the active domain is included within the bounded quantification domain, for every reachable state s.

■

Proof: Since $adom(s)$ is the union of all objects in all fluents at state s, (3.43) is logically equivalent to the conjunction of:

$$F(x, \vec{y}, s) \supset x \in bdom(s) \tag{3.45}$$

for every fluent F and every variable x of fluent F. We prove (3.45) by induction in s.

At state $s = S_0$, (3.45) is deduced from (3.38) and the hypothesis that list $bdom_{S_0}$ contains all objects in sets S_F and S_P, and in particular the objects that satisfy F at state S_0.

In the inductive step, we prove

$$F(x, \vec{y}, do(A(x_1, ..., x_n), s)) \supset x \in bdom(do(A(x_1, ..., x_n), s)) \tag{3.46}$$

from the inductive hypothesis $adom(s) \subseteq bdom(s)$, for all actions symbols $A \in \mathcal{A}$. Recalling equation (3.37), $bdom(s)$ is inductively defined by:

$$bdom(x, do(A(x_1, ..., x_n), s)) = bdom(x, s) \vee x = x_1 \vee ... \vee x = x_n \tag{3.47}$$

for every action symbol $A \in \mathcal{A}$. Assuming that fluent F's SSA is $Poss(a, s) \supset F(x, \vec{y}, do(a, s)) \equiv \varphi(x, \vec{y}, a, s)$, and given (3.47), (3.46) is logically equivalent to:

$$\varphi(x, \vec{y}, A(x_1, .., x_n), s) \supset (bdom(x, s) \vee x = x_1 \vee ... \vee x = x_n)$$

or:

$$\varphi(x, \vec{y}, A(x_1, ..., x_n), s) \wedge x \neq x_1 \wedge ... \wedge x \neq x_n \supset bdom(x, s) \tag{3.48}$$

Given the inductive hypothesis $adom(s) \subseteq bdom(s)$, (3.48) is a logical consequence of

$$\varphi(x, \vec{y}, A(x_1, ..., x_n), s) \wedge x \neq x_1 \wedge ... \wedge x \neq x_n \supset adom(x, s), \tag{3.49}$$

which is the theorem's successor state axiom safeness hypothesis (3.44).

■

To sum up, we will require every quantified formula φ in $(\exists x)\varphi$ or $(\forall x)\neg\varphi$ to satisty the safeness hypothesis (3.14) if this formula is to be translated using bounded quantification; and we will require all successor state axioms in the specification to satisfy the SSA safeness hypothesis (3.44).

3.6 Formal Results for Bounded Quantification

We prove in this section the soundness and completeness of bounded quantification (provided that the safeness hypothesis from the previous section hold for Σ_{Reiter}), namely the following result:

$$\Sigma_{Reiter} \models \varphi \; iff \; \Sigma_{equ} \models \varphi_{equ} \tag{3.50}$$

that is, all logical consequences of Σ_{equ} are logical consequences of Σ_{Reiter} (in the original situation calculus language), and all logical consequences of Σ_{Reiter} are also logical consequences of Σ_{equ} (in the equational situation calculus language). The proof will be based on the following lemma: $\Sigma_{Reiter} \cup \neg\varphi$ is equiconsistent with $\Sigma_{equ} \cup \neg\varphi_{equ}$.

However, from the integrity constraint proof–process point of view, (3.50) is a result a little too weak, because both Σ_{Reiter} and Σ_{equ} are second order theories and therefore little relationship exists between the concept $\Sigma_{Reiter} \models \varphi$ and the concepts $\Sigma_{Reiter} \vdash \varphi$ or $\Sigma_{equ} \vdash \varphi_{equ}$, since no deductive system is complete for inductive theories. Since we are interested in this thesis in proof processes (rather than in the notion of logical consequence only), we need a stronger result than (3.50) that somehow incorporates the $\vdash$ deduction concept. We propose the following approach to this problem:

Generally, inductive theorem provers use *inductive schemata* in their proof processes, such as Reiter's inductive schema ($IP_{S_0 \geq s}$) in [33], which consists basically of the following proof hint: from

$$\varphi(S_0) \tag{3.51}$$

$$(\forall a, s)\varphi(s) \wedge Poss(a, s) \supset \varphi(do(a, s)), \tag{3.52}$$

plus the inductive principle for states IP, $(\forall s \geq S_0)\varphi(s)$ is derivable. Therefore, when trying to prove $(\forall s \geq S_0)\varphi(s)$, Reiter's suggestion is to try proving (3.51) and (3.52).

In general, let us consider an arbitrary transformation $\mathcal{T}$ on formulae. We call this transformation an *inductive schema* (for the proof of static IC's in the situation calculus) iff, for every formula $\varphi(s)$ that has only one free variable, of sort state, $IP \models \mathcal{T}(\varphi) \supset (\forall s \geq S_0)\varphi$.

In the above case, the transformation is (unformally) $\mathcal{T}(\varphi) = \varphi(S_0) \wedge (\forall a, s)(\varphi(s) \wedge Poss(a, s) \supset \varphi(do(a, s)))$. Formally speaking, the transformation reads:

$$\mathcal{T}(\varphi) = \varphi|^s_{S_0} \wedge (\forall a, s)(Poss(a, s) \wedge \varphi \supset \varphi|^s_{do(a,s)}) \tag{3.53}$$

Transformation (3.53) is, according to our definition, an inductive schema. In fact, Reiter proves in [33] precisely that, for every formula $\varphi(s)$:

$$IP \models \{\varphi(S_0) \wedge (\forall a, s)(Poss(a, s) \wedge \varphi(s) \supset \varphi(do(a, s))\} \supset (\forall s \geq S_0)\varphi(s)$$

Other transformations on formulae are also inductive schemata (for the proof of static IC's in the situation calculus). For instance,

$$\begin{aligned} &\mathcal{T}'(\varphi) = \varphi|^s_{S_0} \wedge (\forall a)\varphi|^s_{do(a,S_0)} \wedge \\ &(\forall a, b, s)(\varphi \wedge Poss(a, s) \wedge Poss(b, do(a, s)) \supset \varphi|^s_{do(b,do(a,s))}) \end{aligned} \tag{3.54}$$

is also an inductive schema, since it is possible to prove that, for every formula φ[18]:

$$IP \models \{\varphi(S_0) \wedge (\forall a)\varphi(do(a, S_0)) \wedge \\ (\forall a, b, s)(\varphi \wedge Poss(a, s) \wedge Poss(b, do(a, s)) \supset \varphi(do(b, do(a, s))))\} \supset \\ (\forall s \geq S_0)\varphi \tag{3.55}$$

Inductive theorem proof processes generally consist, then, in an inductive schema search by means of heuristics, such as those provided by the Cover Set Method, the Boyer and Moore heuristics [8, 26], structural induction [11, 2] or others; next, assuming that an inductive schema $\mathcal{T}$ was found, $\mathcal{T}(\varphi)$'s proof is attempted by means of a (first order complete[19]) set of deduction rules.

This inductive schemata search problem makes it difficult to provide results on proof processes since should there exist an (in an appropriate sense) complete search method for inductive schemata, then a complete deductive system for inductive theories could be defined by coupling the supposedly complete inductive schemata search method to an arbitrary first order complete deductive system for first order logic.

However, before the presentation of formal results, we point out that in [22] Fangzhen Lin proves a lemma that seems to contradict all of the above discussion. We devote therefore the following subsection to a discussion of that lemma.

3.6.1 On Lin's lemma

Fangzhen Lin's lemma in [22] reads:

$$\mathcal{D}_{SSA} \cup \mathcal{D}_{UNA} \cup \mathcal{D}_{IP} \cup \{IC(S_0)\} \models (\forall s \geq S_0)IC(s)$$

[18]We omit (3.55)'s proof for space's sake.

[19]In theory, of course, since theorem provers are computer programs and first order logic is undecidable.

iff

$$\mathcal{D}_{UNA} \models IC(s) \wedge Poss(a,s) \supset IC(do(a,s))$$

This lemma is stated in the context of the solution to the frame problem in presence of ramification constraints (ramification problem) and to the problem of deriving necessary and sufficient preconditions for action given a set of necessary preconditions for actions (qualification problem). In particular, the kind of specifications used in that paper are slightly different from the 'standard' ones, as described in [32], and in particular in the above lemma the precondition axioms $\mathcal{D}_{APA}$ are not considered as part of the specification. It is easy to adapt Fangzhen Lin's proof in [22] so as to produce the following version of the above lemma

Lemma 3. $\mathcal{D}_{SSA} \cup \mathcal{D}_{UNA} \cup \mathcal{D}_{IP} \cup \{IC(S_0)\} \cup \mathcal{D}_{APA} \models (\forall s \geq S_0) IC(s)$ iff $\mathcal{D}_{UNA} \models IC(s) \wedge Poss(a,s) \supset IC(do(a,s))$

which is closer to the 'standard' situation calculus specifications as described in [32]. We briefly prove this lemma first, and discuss it afterwards:

Proof: (adapted from Fangzhen Lin's proof of Lemma 5 in [22])

The 'if' part of the lemma is straightforward by induction.

We prove the 'only if' part of the lemma by assuming that

$$\mathcal{D}_{SSA} \cup \mathcal{D}_{UNA} \cup \mathcal{D}_{IP} \cup \mathcal{D}_{APA} \models (\forall s \geq S_0) IC(s) \tag{3.56}$$

and by considering a model $\mathcal{M}$ of $\mathcal{D}_{UNA}$ such that:

$$\mathcal{M}, \sigma_s, \sigma_a \models Poss(a,s) \wedge IC(s) \tag{3.57}$$

where σ_a and σ_s are arbitrary assignments of values to variables a and s. We prove that

$$\mathcal{M} \models \mathcal{R}[IC(do(a,s))] \tag{3.58}$$

From this fact, the lemma clearly follows. We prove (3.58) by considering a model $\mathcal{N}$ of $\mathcal{D}_{SSA}$, $\mathcal{D}_{APA}$, $\mathcal{D}_{IP}$ and $\mathcal{D}_{UNA}$, that is, a model of the situation calculus whose interpretation for fluent symbols at state S_0 is the same as $\mathcal{M}$'s at state s (given assignment σ_s). That is, for each fluent F and for every assignment σ_o of values to variables in vector $\vec{x}$:

$$\mathcal{M}, \sigma_o, \sigma_s \models F(\vec{x}, s) \;\; iff \;\; \mathcal{N}, \sigma_\wr \models \mathcal{F}(\vec{\S}, \mathcal{S}_\prime)$$

or, in other words, $\mathcal{M}$ in state s, given assignment σ_s is isomorphic to $\mathcal{N}$ in state S_0. From (3.57), trivially $\mathcal{M}, \sigma_a, \sigma_s \models do(a,s) \geq s$, from which, since $\mathcal{M}$ and $\mathcal{N}$ are isomorphic (in the above sense):

$$\mathcal{N}, \sigma_a \models do(a, S_0) \geq S_0 \tag{3.59}$$

and from (3.59) and (3.56)

$$\mathcal{N}, \sigma_a \models IC(do(a, S_0)) \tag{3.60}$$

Since $\mathcal{M}$ and $\mathcal{N}$ are isomorphic (in the above sense), from (3.60) we get (3.58).

■

The apparent contradiction between the above lemma and the previous subsection relies in the following fact, derived from the above lemma: if

$$\mathcal{D}_{SSA} \cup \mathcal{D}_{UNA} \cup \mathcal{D}_{IP} \cup IC(S_0) \cup \mathcal{D}_{APA} \models (\forall s \geq S_0) IC(s) \tag{3.61}$$

then

$$\mathcal{D}_{UNA} \models IC(s) \wedge Poss(a, s) \supset IC(do(a, s)),$$

which can be interpreted as the fact that if an IC is derivable from the database updates specification, then it is derivable by means of Reiter's inductive schema $IP_{s \geq S_0}$. The apparent contradiction is solved by observing that the left side of the '$\models$' symbol in (3.61) is not the entire database specification, as presented in [32], but rather a specification that is limited to $IC(S_0)$ in the initial state. In a standard database updates specification, as presented in [32], the description of the initial database is in general a categorical one, and therefore a stronger theory that $IC(S_0)$ (for most ICs). In particular, an IC that is provable by means of inductive schema (3.54) for instance requires *two* base case hypothesis: $IC(S_0)$ and $(\forall a)IC(do(a, s))$, of which the second is in general not a logical consequence of the first, and therefore it might be the case that such an integrity constraint follow from the entire specification, but not from the a specification that is limited to formula $IC(S_0)$ at state S_0, as required by (our adapted version of) Lin's lemma. There is no contradiction, therefore, between Lin's lemma and the discussion from the previous subsection.

3.6.2 The results

We will assume from now on that the problem of finding an appropriate inductive schema $\mathcal{T}(\varphi)$ has somehow been solved, for instance by the Cover Set Method or by another inductive schemata search method, or even suggested by a user. Indeed, our little experience shows that Reiter's inductive schema (3.51)+(3.52) is in most practical cases suitable —in the context, of course, of situation calculus database specifications.

More precisely, we will concentrate on the problem of proving $\mathcal{T}(\varphi)$ from Σ^-, defined as Σ_{Reiter} minus the induction principle IP[20], from which $(\forall s \geq S_0)\varphi(s)$ follows (whenever $\mathcal{T}$ is indeed an inductive schema). Since φ is an arbitrary situation calculus formula and $\mathcal{T}$ can be also more or less arbitrary[21], we abstract $\mathcal{T}(\varphi)$ to an

[20] Σ^- is therefore a shorthand for $\mathcal{D}_{SSA} \cup \mathcal{D}_{APA} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_P \cup \mathcal{D}_{UNA}$.

[21] For instace, proving that $\varphi(s)$ implies $\varphi(do(a, do(b, do(c, s))))$, or inductive schemata analogous

arbitrary first order formula φ, without making an overgeneralization (but keeping in mind that in the following discussion, formulae, for instance φ, are mostly intended to represent inductive schemata applied to formulae, for instance $\mathcal{T}(\varphi)$, since the interest of this thesis is more on inductive rather than just first-order proof processes).

We face therefore the following problem: is it the case that

$$\Sigma^{-}_{equ} \models \varphi_{equ} \; iff \; \Sigma^{-} \models \varphi \,?$$

The answer is no, and a very simple counter-example can be given: for example, $(\forall x)p(x, s) \supset p(o_1, s)$ (o_1 is the name of a constant). It is clear that this formula is not only a logical consequence of some Σ, but moreover a tautology and therefore a logical consequence of any Σ, and in particular of the $\emptyset$ set of formulae. Then, assuming that Σ^{-} is empty, or that it contains irrelevant formulae, it is clearly the case that $\Sigma^{-} \models \varphi$ if φ is $(\forall x)p(x, s) \supset p(o_1, s)$.

Let us now consider our version of φ, φ_{equ}, which reads: $ball(s, bdom(s)) \supset p(o_1, s)$. It is clear that, *ball* and p being two distinct predicate symbols, this formula is in no sense a tautology (only an 'ordinary' formula). Therefore, assuming that Σ^{-}_{equ} is the empty set (or a set of irrelevant formulae), it is clear that in this case $\Sigma^{-} \models \varphi$ whereas $\Sigma^{-}_{equ} \not\models \varphi_{equ}$. It is easy to see that the problem remains, even considering that the *ball* symbol definition (3.39) is included in Σ^{-}_{equ}.

The problem is simple: the above formula φ is, in Reiter's specification, a (first–order) logical consequence of Σ^{-} (in this particular counter–example, a tautology, but this is not the case, of course, in general). In our specification, such a simple fact is an *inductive consequence* of the specification Σ_{equ}. This fact is not surprising, since the *ball* and *bexist* symbols definitions (3.39) and (3.40) are inductive. That is, *inductive proofs* are necessary for bounded quantification specifications for formulae that are proved for Reiter's specification by means of first–order mechanisms only. In particular, we are again facing the inductive schemata search problem for those inductive proofs, which we supposed being solved above. We solve it by providing two sets of inductive lemmata (which we call LL and DR) such that: if the situation

to (3.4) through (3.7).

calculus database updates specification satisfies the safeness hypothesis presented in section 3.5, $\Sigma_{Reiter} \models LL \cup DR$; and

$$\Sigma^{-} \models \varphi \;\; iff \;\; \Sigma^{-}_{equ} \cup LL \cup LD \models \varphi_{equ} \tag{3.62}$$

We give further on (3.50) and (3.62)'s proof, which are actually (almost) the same proof, but we first prove the lemmata (we discuss in the next section implementation issues related to these lemmata).

Lemma 4. If

a) Σ_{Reiter} satisfies safeness hypothesis (3.44) for all its successor state axioms;

b) Σ_{equ} is constructed from Σ_{Reiter} using transformation *btrans*; and

c) an arbitrary formula $\varphi(\vec{x}, y, s)$ in Σ_{Reiter} satisfies the (3.14) safeness criterium for variable y, then formula

$$\varphi(x, \vec{y}, s) \supset bexist_{\varphi}(\vec{y}, s, bdom(s)) \tag{3.63}$$

(resp.

$$ball_{\neg\varphi}(\vec{y}, s, bdom(s)) \supset \neg\varphi(x, \vec{y}, s) \;\;) \tag{3.64}$$

is an (inductive) consequence of Σ_{equ}.

We call LL the set of all such lemmata, for every formula φ that needs, following algorithm *btrans*, to be translated into a bounded quantification term. As mentioned in the introduction, this lemma can be compared to Hilbert's logical axiom for universal quantifiers $(\forall x)\psi(x) \supset \psi(y)$. This fact is not surprising either, since we are somehow giving a semantics to bounded quantifiers. We will see in next section that the following proof is easily reproduced using RRL.

Proof: We give a generical proof for Lemma 4, for a generic formula φ for the universal quantification case. The proof is analogous for the existential one.

(3.64) follows easily from the following two sublemmata:

$$\varphi(x, \vec{y}, s) \supset appears(x, bdom(s)) \tag{3.65}$$

$$ball_{\neg\varphi}(\vec{y}, s, bdom(s)) \wedge appears(x, bdom(s)) \supset \neg\varphi(x, \vec{y}, s) \tag{3.66}$$

where $appears(x, l)$ is the membership predicate of an object in a list, which we informally called $x \in l$ in preceeding sections[22]. Its axiomatization is

$$\begin{aligned} &appears(x, []) \equiv false \\ &appears(x, [y|l]) \equiv (x = y) \vee appears(x, l). \end{aligned} \tag{3.67}$$

(3.65), as explained in section 3.5, follows from Theorem 4, whose hypothesis are the first two hypothesis of this lemma, and from φ's safeness, which is the third hypothesis[23]. (3.66) is in turn a logical consequence of:

$$ball_{\neg\varphi}(\vec{y}, s, dom(s_1)) \wedge appears(x, bdom(s_1)) \supset \neg\varphi(x, \vec{y}, s) \tag{3.68}$$

(a generalization of (3.66)), which is provable for any φ induction on s_1. We briefly

[22]We switch to the $appears(x, l)$ notation here, since the $x \in l$ notation leads to confusion in this proof.

[23]Recall that this last hypothesis is undecidable, but that it can be established by means of sufficient syntactical criteria, as explained in section 3.5.

show (3.68)'s proof: the base case is:

$$ball_{\neg\varphi}(\vec{y}, s, bdom(S_0)) \wedge appears(x, bdom(S_0)) \supset \neg\varphi(x, \vec{y}, s) \tag{3.69}$$

Given (3.67) and (3.38), $appears(x, bdom(S_0))$ reduces to the disjunction of the literals $(x = c_i)$ for all constants c_i in $bdom_{S_0}$, and therefore (3.69) reduces to the set of sublemmata

$$ball_{\neg\varphi}(\vec{y}, s, bdom(S_0)) \wedge (x = c_i) \supset \neg\varphi(x, \vec{y}, s)$$

or

$$ball_{\neg\varphi}(\vec{y}, s, bdom(S_0)) \supset \neg\varphi(c_i, \vec{y}, s) \tag{3.70}$$

for each constant c_i in $bdom_{S_0}$.

In turn, $ball_{\neg\varphi}(\vec{y}, s, bdom(S_0))$ reduces, given (3.38) and (3.39), to

$$\bigwedge_{appears(c_j, bdom_{S_0})} \neg\varphi(c_j, \vec{y}, s),$$

from which $\neg\varphi(c_i, \vec{y}, s)$ follows for each particular c_i, as stated in (3.70). This ends the base case of the inductive proof.

Next, the inductive step:

$$(ball_{\neg\varphi}(\vec{y}, s, bdom(s_1)) \wedge appears(x, bdom(s_1)) \supset \neg\varphi(x, \vec{y}, s)) \supset$$
$$(ball_{\neg\varphi}(\vec{y}, s, bdom(do(a, s_1))) \wedge appears(x, bdom(a, s_1)) \supset \neg\varphi(x, \vec{y}, s))$$

which can be divided into three subgoals:

$$\neg ball_{\neg\varphi}(\vec{y}, s, bdom(s_1)) \wedge ball_{\neg\varphi}(\vec{y}, s, bdom(do(a, s_1))) \wedge$$
$$appears(x, bdom(do(a, s_1))) \supset \neg\varphi(x, \vec{y}, s) \tag{3.71}$$
$$\neg appears(x, bdom(s_1)) \wedge ball_{\neg\varphi}(\vec{y}, s, bdom(do(a, s_1))) \wedge$$
$$appears(x, bdom(do(a, s_1))) \supset \neg\varphi(x, \vec{y}, s) \tag{3.72}$$
$$\neg\varphi(x, \vec{y}, s) \wedge ball_{\neg\varphi}(\vec{y}, s, bdom(do(a, s_1))) \wedge$$
$$appears(x, bdom(do(a, s_1))) \supset \neg\varphi(x, \vec{y}, s) \tag{3.73}$$

(3.73) is trivially tautologic, since the consequent appears as an antecedent's literal. Next, given (3.39) and (3.37), (3.71) can be rewritten as:

$$\neg ball_{\neg\varphi}(\vec{y}, s, bdom(s_1)) \wedge ball_{\neg\varphi}(\vec{y}, s, bdom(s_1)) \wedge \bigwedge_{x_i = x_1, \ldots, x_n} \neg\varphi(x_i, \vec{y}, s) \wedge$$
$$appears(x, bdom(do(A(x_1, \ldots, x_n), s_1)) \supset \neg\varphi(x, \vec{y}, s)$$

for each $A(x_1, ..., x_n) \in \mathcal{A}$. Each such formula is tautologic since its antecedent is contradictory.

Finally, (3.72) can be rewritten as:

$$\neg appears(x, bdom(s_1)) \wedge ball_{\neg\varphi}(\vec{y}, s, bdom(do(A(x_1, \ldots, x_n), s_1))) \wedge$$
$$(appears(x, bdom(s_1)) \vee (x = x_1) \vee \ldots \vee (x = x_n)) \supset \neg\varphi(x, \vec{y}, s)$$

for each $A(x_1, ..., x_n) \in \mathcal{A}$, which, after the cancellation of the $appears(x, bdom(s_1))$ and the $\neg appears(x, bdom(s_1))$ becomes the conjunction of:

$$\neg appears(x, bdom(s_1)) \wedge ball_{\neg\varphi}(\vec{y}, s, bdom(do(A(x_1, \ldots, x_n), s_1))) \wedge (x = x_j) \supset \neg\varphi(x, \vec{y}, s)$$

or:

$$\neg appears(x_j, bdom(s_1)) \wedge ball_{\neg\varphi}(\vec{y}, s, bdom(do(A(x_1, ..., x_n), s_1))) \supset \neg\varphi(x_j, \vec{y}, s) \tag{3.74}$$

for each variable $x_j = x_1...x_n$. $ball_{\neg\varphi}(\vec{y}, s, bdom(do(A(x_1, ..., x_n), s_1)))$ is in turn

$$ball_{\neg\varphi}(\vec{y}, s, bdom(s_1)) \wedge \bigwedge_{x_i = x_1, \ldots, x_n} \neg\varphi(x_i, \vec{y}, s)$$

which therefore includes (3.74)'s consequent, $\neg\varphi(x_j, \vec{y}, s)$.

■

Lemma 5. If Σ_{equ} is constructed from Σ_{Reiter} using transformation $btrans$, then for every pair of situation calculus formulae χ and φ

$$\Sigma_{equ} \cup \{\chi \supset \varphi(x, \vec{y}, s)\} \models \chi \supset ball_{\varphi}(\vec{y}, s, bdom(s)) \tag{3.75}$$

if variable x does not appear free in χ.

Notice that it is not possible to express this lemma as a single formula such as for example $(\neg\chi \vee (\forall x)\varphi(x, \vec{y}, s)) \supset (\neg\chi \vee ball_{\varphi}(\vec{y}, s, bdom(s)))$ (or, more simply, $(\forall x)\varphi(x, \vec{y}, s) \supset ball_{\varphi}(\vec{y}, s, bdom(s))$), since the quantifier symbol in the antecedent of both formulae has no counterpart in (conditional) equational languages, as seen in section 3.1. In the same way Lemma 4 can be considered as Hilbert's logical axiom's bouded quantification counterpart, Lemma 5 can be considered as Hilbert's deduction rule counterpart. Recall that Hilbert's deduction rule for quantifiers [13] is

$$\frac{\varphi \supset \psi}{\varphi \supset (\forall x)\psi}$$

whenever variable x does not appear free in φ.

Proof: As for Lemma 4, we give a generical proof for Lemma 5 for an arbitrary situation calculus formula φ. From Σ_{equ} and

$$\chi \supset \varphi(x, \vec{y}, s) \tag{3.76}$$

formula

$$\chi \supset ball_{\varphi}(\vec{y}, s, bdom(s)) \tag{3.77}$$

should be derivable if variable x does not appear in χ. We therefore assume (3.76) and prove (3.77), by generalizing (3.77) to

$$\chi \supset ball_{\varphi}(\vec{y}, s, bdom(s_1)) \tag{3.78}$$

and proving (3.78) by induction on s_1.

The base case in (3.78)'s proof is $\chi \supset ball_{\varphi}(\vec{y}, s, bdom_{S_0})$ which reduces to

$$\chi \supset (ball_{\varphi}(\vec{y}, s, []) \wedge \bigwedge_{c_i \in bdom_{S_0}} \varphi(c_i, \vec{y}, s))$$

in which $ball_{\varphi}(\vec{y}, s, [])$ is identically true, according to (3.39), and the other consequents follow from χ, given (3.76) (recalling that variable x in (3.76) is implicitly universally quantified).

The inductive step is:

$$(\chi \supset ball_{\varphi}(\vec{y}, s, bdom(s_1))) \supset (\chi \supset ball_{\varphi}(\vec{y}, s, bdom(do(a, s_1))))$$

which can be divided into two subgoals:

$$\neg\chi \wedge \chi \supset ball_\varphi(\vec{y}, s, bdom(do(a, s_1))) \tag{3.79}$$

$$ball_\varphi(\vec{y}, s, bdom(s_1)) \wedge \chi \supset ball_\varphi(\vec{y}, s, bdom(do(a, s_1))) \tag{3.80}$$

(3.79) is trivially tautologic. (3.80), for each $A(x_1, ..., x_n) \in \mathcal{A}$ is:

$$ball_\varphi(\vec{y}, s, bdom(s_1)) \wedge \chi \supset ball_\varphi(\vec{y}, s, bdom(do(A(x_1, ..., x_n), s_1))),$$

which, given (3.39), becomes:

$$ball_\varphi(\vec{y}, s, bdom(s_1)) \wedge \chi \supset (ball_\varphi(\vec{y}, s, bdom(s_1)) \wedge \bigwedge_{x_j = x_1 \ldots x_n} \varphi(x_j, \vec{y}, s))$$

which, after the simplification of the $ball_\varphi(\vec{y}, s, bdom(s_1))$ literal, becomes:

$$ball_\varphi(\vec{y}, s, bdom(s_1)) \wedge \chi \supset \varphi(x_j, \vec{y}, s)$$

(repeated n times), which is a logical consequence of (3.76), recalling that variable x in (3.76) is implicitly universally quantified.

■

Anticipating the implementation sections, lemmata 4 and 5's proof can be reproduced almost as presented here by the RRL theorem prover. The following theorem follows from lemmata 4 and 5:

Theorem 5. If the situation calculus database updates specification Σ_{Reiter} satisfies the safeness criteria defined in section 3.5; and if Σ_{equ} and φ_{equ} are constructed from Σ_{Reiter} and φ using transformation *btrans* from section 3.4, then

– (A) $\Sigma_{Reiter} \models \varphi$ iff $\Sigma_{equ} \models \varphi_{equ}$, and

– (B) $\Sigma^- \models \varphi$ iff $\Sigma^-_{equ} \cup LL \cup DR \models \varphi_{equ}$

■

Proof: Let us consider first step 2 in transformation $btrans$: since we assume that the φ in $\mathcal{D}_{S_0}$ and in $\mathcal{D}_P$ represent finite relations, those φ can be evaluated to sets of tuples and they are therefore logically equivalent to

$$\bigvee_{<o_1,\ldots,o_n>\in S_F} x_1 = o_1 \wedge \ldots \wedge x_n = o_n$$

so equiconsistency (and moreover logical equivalence) is preserved by step 2.

Next, considering step 3, since $bdom$ is a new symbol within Σ_{equ}, given a model $\mathcal{M}$ for Σ_{equ}, $bdom$'s interpretation can always be defined within an extension of $\mathcal{M}$ so as to satisfy (3.37) and (3.38), whenever those two formula are consistent, which can be easily shown. The equiconsistency between Σ_{equ} prior to, and after, step 3 follows easily from this observation.

Finally, we consider step 4, which is an iteration. Before each iteration step, we have a specification of the form $\Sigma \cup \{\psi\}$ and after the iteration step we have

$$\Sigma \cup \{\psi|^{(\forall x)\varphi(x,\vec{y},s)}_{ball(\vec{y},s,bdom(s))}\}. \tag{3.81}$$

W.r.t (A) above, since the hypothesis of lemmata 4 and 5 are verified (because they are the hypothesis of the theorem itself, in particular for formula φ), both lemmata hold for φ. In case (B) both lemmata are hypothesis, so in either case both lemmata hold, and the following formula is easily shown to be a logical consequence of both lemmata:

$$ball_\varphi(\vec{y}, s, bdom(s)) \equiv (\forall x)\varphi(x, \vec{y}, s) \tag{3.82}$$

Theorem 5's proof is therefore similar, from now on, to Theorem 1's proof: given (3.82), $\Sigma \cup \{\psi\}$ is proved equiconsistent with (3.81) through the same steps in Theorem 1's proof, which we then omit here for space's sake.

Finally, by induction on the number of iterative steps in step 4 of transformation $btrans$, the final equiconsistency result is proved.

■

3.7 Implementation Issues

We discuss in this section the implementation of integrity constraint proof processes using the RRL theorem prover within the SCDBR reasoning system.

We first follow in the implementation the following three steps:

- A syntactical translation process, by means of transformation $btrans$, of Reiter's situation calculus specification into the equational situation calculus specification, using RRL's syntax. In particular, the equivalences are replaced by boolean term equalities. We describe this process in appendix B, which contains the implementation's documentation.
- The declaration of the S_0 and do symbols as free constructors for data type $state$.
- The syntactical translation of a conjectured integrity constraint $(\forall s \geq S_0)IC(s)$ and the activation of RRL with this input formula, so as to attempt its proof from the equational specification.

Within the proof process, RRL first orients the situation calculus equations into rewrite rules. This orientation process is quite simple in our case, since it reduces to defining a lexicographical ordering between fluent symbols, the $Poss$ and $reach$ symbols and the $ball$ and $bexist$ symbols. Being a quite simple but highly technical issue, we omit it here for space's sake.

After the equation orientation process, RRL attepts to prove the conjecture by searching for an inductive schema $\mathcal{T}$ for conjecture $(\forall s \geq S_0)IC(s)$ by means of the Cover Set Method. Having found a candidate inductive schema $\mathcal{T}$, RRL attempts to prove $\mathcal{T}(\varphi)$ by means of first order procedures (whenever possible), and our inductive lemmata will in general help within this first order proof process, since they guarantee the ('first order') equiconsistence between $\Sigma_{Reiter} \cup \{\neg\varphi\}$ and $\Sigma_{equ} \cup LL \cup DR \cup \{\neg\varphi_{equ}\}$.

With respect to the inductive lemmata LL, it is reasonable to prove those lemmata *prior* to the integrity constraint's proof itself by means of an (RRL) proof analogous to the one that was given for Lemma 4. This process involves no significant difficulty. Alternatively, the specifyier could simply 'decret' the lemmata LL by adding them to the equational situation calculus specification. This approach implies a slight risk, which is the fact that the lemmata follow from the equational specification *provided that* the original situation calculus specification satisfies the safeness hypothesis from section 3.5. Otherwise, no guarantee exists that the lemmata are logical consequences of, or even consistent with, the equational specification, and therefore they should be added with care to the equational specification (or at least section 3.5's safeness hypothesis should be verified).

With respect to the other set of inductive lemmata DR, since those lemmata do not consist of first order formulae, but rather of the fact that a formula follows from another, as seen in the previous section, it is not possible to prove them from the specification prior to the integrity constraint proof process, since they are not logic formulae. Moreover, even if DR was somehow meaningfull as a set of formulae, there would infinitely many of them, one for each possible wff χ in (3.75), therefore making a proof for each one of them an impossible task. We propose the following approach to this problem:

The set DR can be considered, as mentioned in the previous section, as a set of deduction rules, analogous to Hilbert's deduction rule for first order quantifiers. That is, the set of lemmata DR states that, whenever $\chi \supset \varphi(x, \vec{y}, s)$ has been derived from Σ_{equ}, $\chi \supset ball_{\varphi}(\vec{y}, s, bdom(s))$ is also derivable. The RRL theorem prover, however, behaves in a top-down manner, that is: given a conjecture, it generates a set of lemmata from which the conjecture follows. Next, in order to prove each lemmata, it generates a set of sub-lemmata from which the lemma follows, and so on, until a tautologic lemma is generated, or a lemma that is a formula belonging to Σ_{equ}. In this context, the concept of 'deduction rule' must be considered backwards, that is, an arbitrary deduction rule $\frac{X}{Y}$ (X and Y being for instance first order formulae), is to be interpreted as the possibility of generating sublemma X from lemma Y (and not, as usual, as the possibility of deriving Y once X has been derived, which would be the case in a forward reasoning theorem prover). Our implementation for the deductive rules DR with the RRL theorem prover must therefore somehow produce sublemma $\chi \supset \varphi(x, \vec{y}, s)$ whenever RRL generates lemma $\chi \supset ball_\varphi(\vec{y}, s, bdom(s))$. A natural way for doing so consists precisely in emulating the proof of Lemma 5, as follows:

a) Given a failed[24] sublemma of the form $\chi \supset ball_\varphi(\vec{y}, s, bdom(s))$, suggest RRL to generalize s in $bdom(s)$ to an arbitrary state s_1, giving the following sublemma to be proved: $\chi \supset ball_\varphi(\vec{y}, s, bdom(s_1))$.

b) Suggest then proving $\chi \supset ball_\varphi(\vec{y}, s, bdom(s_1))$ by induction on s_1.

It is easy to see that the above two steps will emulate the proof of Lemma 5, and should therefore always succeed in generating $\chi \supset \varphi(x, \vec{y}, s)$ as a sublemma for lemma $\chi \supset ball_\varphi(\vec{y}, s, bdom(s))$.

Now, a natural question to be asked is whether RRL (or any theorem prover of its kind) is able to find the above generalization and induction hints by itself, that is, to find appropriate inductive lemmata for $\chi \supset ball_\varphi(\vec{y}, s, bdom(s))$. The generation of inductive lemmata is a difficult problem. Many heuristics have been proposed for this problem [26, 16], but since no deductive system exists for inductive theories, the problem is likely to remain. In the particular case of RRL, this theorem prover implements many generalization heuristics [16], among which dropping a literal from a clause, and a kind of generalization similar to the one that we propose and which consists basically in generating a term t from a conjecture to a (new) variable (for example, generalizing conjecture '$x + s(y) = s(y) + x$' to

[24] This proof mechanism makes of course no sense for successfull sublemmata, since, being successfull, they are already proved.

'$x+z=z+x$'). Clearly, the main risk in such a generalization is overgeneralization, such as for example generalizing '$y+s(s(x))=s(s(y+x))$' to '$y+z=s(s(y+x))$' (in this example, generalizing $s(s(x))$ to the new variable z), which is obviously too strong a lemma. In order to avoid overgeneralization, RRL's heuristics generalize a term to a variable *only* if this term appears twice in the conjecture, once in the left hand side of the equation (remember that the language is conditional equational, with formulae of the form (3.1)) and once in the right hand side; or once among the antecedent literals and once either in the left or the right hand side [16]. All those cases occur seldom within situation calculus specifications, and therefore our generalization and induction heuristics are relevant.

To sum up, we implement the bounded quantification method with the RRL theorem prover by translating (using transformation $btrans$) the situation calculus database updates into an equational situation calculus language; we declare symbols S_0 and do as free constructors for the state data type and we provide a simple lexicographic ordering for the symbols in the equational language that allows the orientation of the equational situation calculus formulae into rewrite rules; we prove prior to the IC proof itself the set LL of lemmata; we start the IC proof and suggest the generalization and induction heuristics 1 and 2 above for failed lemmata, whenever possible. The soundness of this implementation is based on the soundness result for bounded quantification (Theorem 5) and by the completeness results for the RRL deduction procedures [43, 45] (if the input situation calculus specification satisfies seccion 3.5's safeness criteria).

3.8 Example of the Method

We show in this section the important parts of the stock database example's script generated by the the RRL system, which consists of an automatic proof for integrity constraint (2.5). The example illustrates the need for the LL and DR lemmata. The whole script can be found in appendix A.

We first show the system's input, namely the equational specification, written using RRL's syntax. Most elements from this input are self-explanatory:

```
Equations read in are:
```

```
 1. REACH(S0)  [DEF, 1]
 2. REACH(DO(XA, XS)) == REACH(XS) & POSS(XA, XS)  [DEF, 2]
 3. ORDEREDA(XPROD, DO(XA, XS)) == COND(ORDEREDA(XPROD, XS), TRUE, ((XA = ORDER(XPROD)) AND
    BALL(XPROD, XS, BDOM(XS))))  [DEF, 3]
 4. INSTOCK(XPROD, DO(XA, XS)) == COND(INSTOCK(XPROD, XS), TRUE, (XA = NEWITEM(XPROD)))  [DEF, 4]
 5. ORDEREDA(XPROD, S0) == (XPROD = P2)  [DEF, 5]
 6. INSTOCK(XPROD, S0) == COND((XPROD = P1), TRUE, (XPROD = P2))  [DEF, 6]
 7. POSS(ORDER(XPROD), XS) == COND(INSTOCK(XPROD, XS), TRUE, BEXIST(XPROD, XS, BDOM(XS)))  [DEF, 7]
 8. POSS(NEWITEM(XPROD), XS)  [DEF, 8]
 9. BALL(XPROD, XS, NOTHING)  [DEF, 9]
10. BALL(XPROD, XS, C(X, XDOM)) == BALL(XPROD, XS, XDOM) & COND(PROVIDER(X, XS),
    PROVIDES(X, XPROD, XS), TRUE)  [DEF, 10]
11. BEXIST(XPROD, XS, NOTHING) == FALSE  [DEF, 11]
12. BEXIST(XPROD, XS, C(X, XDOM)) == COND(BEXIST(XPROD, XS, XDOM), TRUE, (PROVIDER(X, XS) AND
    PROVIDES(X, XPROD, XS)))  [DEF, 12]
13. BDOM(S0) == C(P1, C(P2, NOTHING))  [DEF, 13]
14. BDOM(DO(XA, XS)) == C(PRODUCTIN(XA), BDOM(XS))  [DEF, 14]
15. BEXIST(XPROD, XS, BDOM(XS)) == TRUE  if  PROVIDER(X, XS) & PROVIDES(X, XPROD, XS)  [DEF, 15]
16. BALL(XPROD, XS, BDOM(XS)) == FALSE  if  PROVIDES(X, XPROD, XS) & PROVIDER(X, XS)  [DEF, 16]
```

Some syntactical observations will help however in this input's understanding: the '`==`' symbol stands for term equality. Variables names start with letters `U` through `Z`. Function and constant names start with letters from `A` through `T`. Recall that predicates are considered as boolean functions. The language is a typed one, so that each argument of each function has a specific type[25]. We defined two functions in this specification, `C` and `NOTHING` which play the role of list constructors, equivalent to Prolog's `[.|.]` and `[]` or to Lisp's CONS and NIL. Finally, the logical 'or' is expressed in RRL by means of the *cond* operator, which is the same as Lisp's: $cond(x, y, z)$ returns y if x is evaluated to $true$ and z otherwise. x is therefore normally a boolean expression. If boolean expressions x and y represent formulae φ and ψ, it is clear that the expression $cond(x, true, y)$ returns the boolean value associated to $\varphi \vee \psi$, since $cond(x, true, y)$ is equal (equivalent) to $true$ if x is true and equal (equivalent) to y if x is false.

Equations `[15]` and `[16]` constitute the set LL of lemmata for this specification. We skipped its proof by including them within the specification, for space's sake. That proof would be essentialy the same as Lemma 3's proof.

We next recall the constructor declaration process:

[25]The type declaration can be found in in RRL's script in appendix A.

```
Type operators you wish to be CONSTRUCTORS: DO S0 ORDER NEWITEM C NOTHING
Is 'DO' a free constructor ? (Y,N,YES,NO) Y
Is 'ORDER' a free constructor ? (Y,N,YES,NO) Y
Is 'NEWITEM' a free constructor ? (Y,N,YES,NO) Y
Is 'C' a free constructor ? (Y,N,YES,NO) Y
```

The above answers are provided by the user. We omit the –rather long– equation orientation process that follows, whose output is a set of rewrite rules, since in this example it reduces to accepting a lexicographic ordering suggested by RRL. The user inputs next a lemma, which we explain further on:

```
RRL-> PROVE
Type your equation in the format:  L == R [if C]
Enter a ']' to exit when no equation is given.
BEXIST(XPROD, XS, BDOM(XS1)) == FALSE if (BALL(XPROD, XS, BDOM(XS)) AND (REACH(XS) AND
(NOT(INSTOCK(XPROD, XS)) AND (NOT(ORDEREDA(XPROD, XS)) AND HINT(INDUC(BDOM(XS1)))))))

Proving [main] BEXIST(XPROD, XS, BDOM(XS1)) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (HINT(INDUC(BDOM(XS1)))) and
          (BALL(XPROD, XS, BDOM(XS)))
    by HINT(INDUC(BDOM(XS1)))

Let P(XS1) be [main] BEXIST(XPROD, XS, BDOM(XS1)) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS)))

The induction will be done on XS1 in BDOM(XS1), and will follow the scheme:
    [#1] P(S0)
    [#2] P(DO(XA, XS1)) if  { P(XS1) }

By rules [13], [12], [11],
     [#1] BEXIST(XPROD, XS, BDOM(S0)) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS)))
     is reduced to
     [#1] COND((PROVIDER(P2, XS) AND PROVIDES(P2, XPROD, XS)), TRUE, (PROVIDER(P1, XS) AND
                        PROVIDES(P1, XPROD, XS))) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS)))

Proving [#1] COND((PROVIDER(P2, XS) AND PROVIDES(P2, XPROD, XS)), TRUE, (PROVIDER(P1, XS) AND
```

```
                    PROVIDES(P1, XPROD, XS))) == FALSE if
        (REACH(XS)) and
        (not(INSTOCK(XPROD, XS))) and
        (not(ORDEREDA(XPROD, XS))) and
        (BALL(XPROD, XS, BDOM(XS)))
        under the condition (PROVIDER(P2, XS) AND PROVIDES(P2, XPROD, XS))
        and its negation.

By rule [16],
    [#1.1] TRUE == FALSE if
        (REACH(XS)) and
        (not(INSTOCK(XPROD, XS))) and
        (not(ORDEREDA(XPROD, XS))) and
        (PROVIDER(P2, XS)) and
        (PROVIDES(P2, XPROD, XS)) and
        (BALL(XPROD, XS, BDOM(XS)))
    is reduced to true.

By rule [16],
    [#1.2] (PROVIDER(P1, XS) AND PROVIDES(P1, XPROD, XS)) == FALSE if
        (REACH(XS)) and
        (not(INSTOCK(XPROD, XS))) and
        (not(ORDEREDA(XPROD, XS))) and
        (not(PROVIDER(P2, XS))) and
        (not(PROVIDES(P2, XPROD, XS))) and
        (BALL(XPROD, XS, BDOM(XS)))
    is reduced to true.

Following equation is proven successfully by case-split:
    [#1] COND((PROVIDER(P2, XS) AND PROVIDES(P2, XPROD, XS)), TRUE, (PROVIDER(P1, XS) AND
                    PROVIDES(P1, XPROD, XS))) == FALSE if
        (REACH(XS)) and
        (not(INSTOCK(XPROD, XS))) and
        (not(ORDEREDA(XPROD, XS))) and
        (BALL(XPROD, XS, BDOM(XS)))

By rules [14], [12], [16],
    [#2] BEXIST(XPROD, XS, BDOM(DO(XA, XS1))) == FALSE if
        (REACH(XS)) and
        (not(INSTOCK(XPROD, XS))) and
        (not(ORDEREDA(XPROD, XS))) and
        (BALL(XPROD, XS, BDOM(XS))) and
        (not(BEXIST(XPROD, XS, BDOM(XS1))))
    is reduced to true.

All subgoals of [main] are proven, hence
   [main] BEXIST(XPROD, XS, BDOM(XS1)) == FALSE if
        (REACH(XS)) and
        (not(INSTOCK(XPROD, XS))) and
        (not(ORDEREDA(XPROD, XS))) and
        (BALL(XPROD, XS, BDOM(XS)))
```

```
    is an inductive theorem.

Following equation
    BEXIST(XPROD, XS, BDOM(XS1)) == FALSE  if
        { HINT(INDUC(BDOM(XS1))),
          REACH(XS),
          BALL(XPROD, XS, BDOM(XS)),
          not(INSTOCK(XPROD, XS)),
          not(ORDEREDA(XPROD, XS)) }   [USER, 17]
    is an inductive theorem in the current system.

Adding Rule:  [17] BEXIST(XPROD, XS, BDOM(XS1)) ---> FALSE  if
        { REACH(XS),
          not(INSTOCK(XPROD, XS)),
          not(ORDEREDA(XPROD, XS)),
          BALL(XPROD, XS, BDOM(XS)) }   [USER, 17]
```

And finally the user inputs the considered integrity constraint, which is automatically proved by RRL:

```
RRL-> PROVE
Type your equation in the format:  L == R [if C]
Enter a ']' to exit when no equation is given.
INSTOCK(XPROD, XS) == TRUE if (ORDEREDA(XPROD, XS) AND REACH(XS))

Let P(XS) be [main] INSTOCK(XPROD, XS) == TRUE if
          (REACH(XS)) and
          (ORDEREDA(XPROD, XS))

The induction will be done on XS in REACH(XS), and will follow the scheme:
    [#1] P(S0)
    [#2] P(DO(XA, XS1)) if  { P(XS1) }

By rules [1], [5], [6],
     [#1] INSTOCK(XPROD, S0) == TRUE if
          (REACH(S0)) and
          (ORDEREDA(XPROD, S0))
     is reduced to true.

Subgoal [#2] is split into:
 [#2.1] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS))) and
          (not(REACH(XS)))
 [#2.2] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS))) and
          (not(ORDEREDA(XPROD, XS)))
 [#2.3] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (REACH(DO(XA, XS))) and
```

```
         (ORDEREDA(XPROD, DO(XA, XS))) and
         (INSTOCK(XPROD, XS))

By rule [2],
     [#2.1] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (not(REACH(XS))) and
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS)))
     is reduced to true.

By rules [2], [3], [4], [7], [17], and by switching the head and the premises,
     [#2.2] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (not(ORDEREDA(XPROD, XS))) and
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS)))
     is reduced to true.

By rules [2], [3], [4],
     [#2.3] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (INSTOCK(XPROD, XS)) and
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS)))
     is reduced to true.

All subgoals of [main] are proven, hence
    [main] INSTOCK(XPROD, XS) == TRUE if
          (REACH(XS)) and
          (ORDEREDA(XPROD, XS))
    is an inductive theorem.

Following equation
    INSTOCK(XPROD, XS) == TRUE  if
        { REACH(XS),
          ORDEREDA(XPROD, XS) }   [USER, 18]
    is an inductive theorem in the current system.
```

An important observation here is the use of lemma (rule) `[17]` in sublemma `[#2.2]`'s proof. This sublemma

```
     [#2.2] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (not(ORDEREDA(XPROD, XS))) and
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS)))
```

can be reduced through rewriting to

```
BEXIST(XPROD, XS, BDOM(XS)) ---> FALSE  if
        { REACH(XS),
```

```
not(INSTOCK(XPROD, XS)),
not(ORDEREDA(XPROD, XS)),
BALL(XPROD, XS, BDOM(XS)) }   [USER, 17]
```

and fails if lemma `[17]` is not previously suggested (after an attempt among others to perform induction on `XS`). The underlying reason for this failure is the one explained in section 3.6 concerning the inductive nature of the *ball* and *bexist* symbol definition, which produce the need for lemmata LL and DR.

Our proof hints are in this case to generalize, as explained in the previous section, variable `XS` in `BDOM(XS)` to an arbitrary state variable `XS1` and to perform induction on that new variable. The output of those proof hints is in this example lemma `[17]`. Notice also the use of the LL lemma (rule) `[16]` within lemmata `[#1.1]` and `[#1.2]`'s proof (within the first proof, of lemma `[17]`), which would fail in a similar way to `[#2.2]` from the other proof process if lemma `[16]` had not been provided.

3.9 Comparison to Other Works

Much work has already been done on topics that are similar or somehow related to this chapter's.

In [1], Abiteboul and Vianu consider transactions that are sequences of insertions and deletions on relational tables. They prove the undecidability of the preservation of arbitrary constraints by such transactions and the decidability of the preservation of a particular set of IC, which they call *p-bounded dependencies*, that can be seen as a generalization of functional dependencies. In [38], Su applies those results to a set of particular cases of p-bounded depencies, namely functional dependencies and inclusion dependencies. He shows that the decision of whether such dependencies are preserved by the above kind of transactions is co-np complete. These two papers provide decidability results, which are therefore stronger that ours, but which are valid for particular classes of integrity constraints and transactions only.

In [37], Sheard and Stemple provide a solution to the problem that we address, namely the proof of database integrity constraints, in the context of relational databases. However, they solve the problem for a particular syntactical class of integrity constraints only, namely what they call *domain constraints*, *tuple constraints*, *relational constraints* or *inter-relational constraints*; moreover the only possible transactions are tuple insertion, deletion, multiple inserts, updates and selective updating or deleting, which are particular cases of the action effects expressible in Reiter's database specification formalism.

In [19], Matt Kaufmann provides a solution to first order quantification within the NQTHM theorem prover. His solution consists basically in a skolemization procedure, after a reduction of the scope of the quantifiers, so as to reduce each Skolem function's number of arguments (whenever possible). Being a skolemization procedure, it would have the drawbacks presented in section 3.1 if it was applied to term rewriting systems.

Boyer and Moore's solution [10] is indeed similar to ours, and our work can be seen as an adaptation of Boyer and Moore's ideas to the situation calculus, along with a proof of correctness results for the bounded quantification mechanism.

Generally speaking, the problem addressed in this chapter must somehow be solved in every application that uses term-rewriting theorem provers, and the user of such systems must find in most cases an ad-hoc solution to this problem for the application he is interested in.

IV. ON DERIVING INDUCTIVE LEMMATA IN THE SITUATION CALCULUS

We present in this chapter a further discussion on subsection 3.6.1's topic, namely the problem of the generation of stronger lemmata for integrity constraints in the situation calculus. We present a simple result on the topic along with a set of simple heuristics for the generation of inductive lemmata for integrity constraints in the situation calculus.

We start the discussion by recalling lemma 3:

$$\mathcal{D}_{SSA} \cup \mathcal{D}_{UNA} \cup \mathcal{D}_{IP} \cup \{IC(S_0)\} \cup \mathcal{D}_{APA} \models (\forall s \geq S_0) IC(s)$$

iff

$$\mathcal{D}_{UNA} \models IC(s) \wedge Poss(a, s) \supset IC(do(a, s))$$

According to subsection 3.6.1's discussion, this lemma cannot be considered as implying that an integrity constraint (inductively) follows from Reiter's database updates specification only if it follows from that specification by means of Reiter's inductive schema $IP_{s \geq S_0}$, in particular because a stronger hypothesis than just $IC(S_0)$ could be needed to hold at state S_0 in order for $(\forall s \geq S_0) IC(s)$ to follow. An important question is therefore: which one? or what can be said about this stronger condition? Let us assume that a stronger condition $SL(S_0)$ holds at state S_0 that is sufficient for the derivation of $(\forall s \geq S_0) IC(s)$ (the meaning of 'stronger' being of course $\models (\forall s) SL(s) \supset IC(s)$). Let us assume that $SL(s)$ is an inductive lemma for IC in the sense that it can be proved from itself at state S_0, that is

$$\mathcal{D}_{SSA} \cup \mathcal{D}_{UNA} \cup \mathcal{D}_{IP} \cup \{SL(S_0)\} \cup \mathcal{D}_{APA} \models (\forall s \geq S_0) SL(s)$$

or in other words no further lemmata are needed in SL's proof. Then, by directly applying Lin's lemma, it is clear that SL is provable by means of Reiter's inductive schema, or in other words

$$\mathcal{D}_{UNA} \models SL(s) \wedge Poss(a,s) \supset SL(do(a,s)).$$

We have therefore shown that:

Lemma 6. If $SL(s)$ is a stronger lemma for integrity constraint $IC(s)$ in the above specified sense then SL is provable by means of Reiter's inductive schema.

There remains the problem of actually finding this lemma. We propose for that purpose the following set of (heuristical) steps:

a) Call IC (the integrity constraint needed to be proved) SL_0. Let i be 0.

b) Verify whether $\Sigma_{Reiter} \models SL_i(S_0)$. This process is in general decidable, since Σ_{Reiter} is categorical, and therefore only one (finite) model exists for the initial state S_0 (up to isomorphisms), so the process reduces to checking the truth of a formula in a finite model.

c) If $\Sigma_{Reiter} \not\models SL_i(S_0)$, the original IC is disproved;

d) else, try proving

$$\Sigma_{Reiter} \vdash Poss(a,s)SL_i(s) \supset SL_i(do(a,s)) \tag{4.1}$$

by means of first order proof methods. This process is in general undecidable, since the original IC is an arbitrary first-order formula. Should a positive result arise (i.e. a proof for (4.1)), the whole process is successfull. Should a non-conclusive result or a first-order refutation for (4.1) be found, consider (4.1) in itself as an inductive goal, i.e. let SL_{i+1} be $Poss(a,s) \wedge SL_i(s) \supset SL_i(do(a,s))$.

e) Let i be $i+1$, and go to step b).

We have provided in this chapter a simple result regarding lemmata for integrity constraints in the situation calculus, along with a set of heuristical steps that can guide the generation of such lemmata.

V. PROVING INTEGRITY CONSTRAINTS USING RESOLUTION-BASED THEOREM PROVERS

We devote this chapter to the discussion of integrity constraint proofs in the situation calculus by means of resolution–based theorem provers. The resolution principle is a single deductive rule proved complete for first-order logic by Robinson in [35]. Resolution–based theorem provers, such as Otter [42] or Prolog, are therefore strongly oriented towards first–order theorem proving and do not provide in general induction mechanisms (such as the Cover Set Method or Boyer and Moore's approach, mentioned in the previous sections). We recall from the previous chapter that the use of induction is in most cases necessary in situation calculus integrity constraint proofs. We solve that problem simply by providing an inductive schema, such as Reiter's schema $IP_{s \geq S_0}$ for instance, to the theorem provers. Our little experience and the structure of Reiter's specification (successor state axioms and so on) suggest that Reiter's inductive schema is appropriate in most cases. We therefore assume throughout this chapter that we intend to prove

$$\Sigma_{Reiter} \models IC(S_0) \tag{5.1}$$

and

$$\Sigma_{Reiter} \models IC(s) \wedge Poss(a, s) \supset IC(do(a, s)) \tag{5.2}$$

by means of resolution-based theorem provers. The results from this chapter can be easily extended to other inductive schemata such as (3.54). We first briefly recall a set of results related to the resolution principle that are relevant to this chapter.

5.1 Overview of Resolution Results

The *resolution principle* is a deductive rule that derives

$$(L_1 \vee ... \vee L_{i-1} \vee L_{i+1} \vee ... \vee L_n \vee M_1 \vee ... \vee M_{j-1} \vee M_{j+1} \vee M_m)\theta$$

from

$$L_1 \vee ... \vee L_n$$

and

$$M_1 \vee ... \vee M_m,$$

where the L_k and the M_k are literals, and θ is a most general unifier of literals L_i and M_j. A *resolution derivation* of a clause C from a set of clauses is a sequence of applications of the resolution rule that produces C as a conclusion.

A *definite* program is a set of clauses of the form $A \leftarrow B_1 \wedge ... \wedge B_n$, where A and the B_i are positive literals (atoms). The '$\leftarrow$' symbol stands for first-order implication. A *definite goal* is a clause of the form $\leftarrow A_1 \wedge ... \wedge A_m$. An *SLD-resolution* derivation of the empty clause from a definite program and a goal G is a resolution derivation in which one of the resolvents in each step of the derivation is a clause from the definite program. In other words, an SLD-derivation is a sequence of goals G_i in which G_0 is G and: if G_i is $\leftarrow A_1 \wedge ... \wedge A_n$; if A_k is an atom in G_i (called the *selected atom*); if $D \leftarrow B_1 \wedge ... \wedge B_m$ is a clause in P; and if θ is a MGU of atoms A_k and D, then G_{i+1} is $(\leftarrow A_1 \wedge ... \wedge A_{i-1} \wedge B_1 \wedge ... \wedge B_m \wedge A_{i+1} \wedge ... \wedge A_n)\theta$. Clearly, all possible SLD-derivations for a given goal G and a given definite program P can be gathered in a (possibly infinite) tree structure, with goal G at its root, called an SLD-tree, in which nodes at level l are goals G_l for some possible SLD-derivation. SLD-resolution is a complete deductive system for definite programs.

A *normal program* is a set of clauses, called *normal clauses*, of the form $A \leftarrow L_1 \wedge ... \wedge L_n$, where the L_i are literals, that is, atoms or negated atoms. Likewise, a *normal goal* is a clause of the form $\leftarrow L_1 \wedge ... \wedge L_n$.

A SLDNF-failed tree (of rank 0) for a *ground atom* A is an SLD-tree for A and program P that does not contain the empty clause as a leaf (recalling that an SLD-tree contains all possible SLD-derivations for a given goal). An SLDNF-derivation (of rank $n+1$) of the empty clause from a goal G is an SLD-derivation with the following additional deductive rule: from goal $\leftarrow L_1 \wedge ... \wedge L_{i-1} \wedge \neg A_i \wedge L_{i+1} \wedge ... \wedge L_m$, where A is a ground atom, derive $\leftarrow L_1 \wedge ... \wedge L_{i-1} \wedge L_{i+1} \wedge ... \wedge L_m$, if a SLDNF failed tree (of rank n) can be produced. Clearly, this definition makes sense only for finitely failed SLDNF trees. An SLDNF-tree (of rank n) for a goal G is the set of all SLDNF-derivations of the empty clause from goal G, gathered in a tree-like structure in analogy to the above definition for an SLD-tree.

A *general program* is a set of *general clauses*, of the form $A \leftarrow \varphi$, where φ is an arbitrary first order formula. The precise semantics of general programs and of correct answers to general queries are given in [23]. We omit them here, since they involve the concepts of minimal Herbrand models, stratified programs, program completion and others, which would be too long to be covered here.

5.2 The Inductive Step

We implement (5.2)'s proof by means of the resolution–based Otter theorem prover. Otter uses two sets of first order formulae (translated into clausal form), usually known to be consistent, called the *usable* set of formulas and the *set of support*, usually conjectured to be inconsistent with the usable set of formulas. Otter tries to derive a contradiction (the empty clause) from both sets. It is therefore clear that, in order for Otter to prove (5.2), one should define the usable set of formulae as Σ_{Reiter} in (5.2), tranformed into clausal form (minus the induction axiom, since this axiom is not expressible in clausal form, not being a first–order formula). Also, the set of support is defined as the negation of $IC(s) \wedge Poss(a,s) \supset IC(do(a,s))$ in (5.2), translated into clausal form.

Otter posesses a rather large set of flags, including which kind of resolution to use (SLD-resolution, LU-resolution, hyperresolution, etc), whether to use paramodulation or not, the assignment of weights to clauses (Otter always chooses the "lightest" clauses first in its reasoning process) and others. Our heuristics regarding the setting of those flags consist in setting paramodulation on, since (5.2) can in general contain equalities. We use binary resolution and hyperresolution, and we set a 'high' weight for the *do* symbol so as to produce whenever possible a 'single state' reasoning by Otter, by avoiding the *do* symbol to appear too many times (since this symbol, according to out weight setting, will generate 'heavy' clauses, which are last considered by Otter).

We refer to appendix B for details related to the implementation.

5.3 The Base Case

The implementation of proofs for the base case (5.1) offers a slight difficulty, which is the following: the integrity constraint at state S_0 will in general follow from the initial database $\mathcal{D}_{S_0}$. Recalling section 3.4, $\mathcal{D}_{S_0}$ consists of a set of axioms of the form

$$F(\vec{x}, S_0) \equiv (x_1 = o_1 \wedge ... \wedge x_n = o_n) \vee ... \vee (x_1 = o_{k-n+1} \wedge ... \wedge x_n = o_k), \quad (5.3)$$

one for each fluent symbol F, where vector $\vec{x}$ has variables $x_1 ... x_n$ and the o_i are constant names (not necessarily distinct). Equivalence (5.3)'s transformation into clausal form starts by dividing (5.3) into

$$(x_1 = o_1 \wedge ... \wedge x_n = o_n) \vee ... \vee (x_1 = o_{k-n+1} \wedge ... \wedge x_n = o_k) \supset F(\vec{x}, S_0) \quad (5.4)$$

and

$$F(\vec{x}, S_0) \supset (x_1 = o_1 \wedge ... \wedge x_n = o_n) \vee ... \vee (x_1 = o_{k-n+1} \wedge ... \wedge x_n = o_k) \quad (5.5)$$

Formula (5.4) is easily represented in clausal form by the logically equivalent set of facts

$$F(o_i, ..., o_{i+n-1}) \tag{5.6}$$

It easy to prove, however, that formula (5.5)'s translation into clausal form yields an exponential number of clauses (proportional to c^m, where m is the number of facts in (5.4), or k/n in the example). It is also clear that a resolution process on an exponential number of clauses is unfeasible, moreover if the exponent is roughly speaking the number of tuples in a relation in the initial database, which is in general a 'large' number.

A natural solution to this problem is to avoid the translation of (5.5) into clauses by using only the (5.6) set of facts, plus some negation-as-failure mechanism. A natural alternative to be considered for the proof of the base case (5.1) is therefore the use of Prolog as a theorem prover. Since IC is an arbitrary first order formula, we implemented Lloyd and Topor's algorithm [23] for its translation into logic programming. We review that algorithm in the following section.

5.4 Overview of Lloyd and Topor's algorithm

Lloyd and Topor's algorithm transforms a general program P and a general query Q into a normal program P' and a normal query Q' that produce the same answers (through SLDNF resolution) as the original P and Q. The algorithm consists of the application of the following set of syntactic transformation rules (which we denote by '$\Longrightarrow$') to the general program and goal, until no further rule can be applied (the process is proved terminating in [23]):

$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg(V \wedge W) \wedge W_{i+1} \wedge ... \wedge W_m$$
$$\Longrightarrow A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg V \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge (\forall x_1...x_k)W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$\Longrightarrow A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg(\exists x_1...x_k)\neg W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg(\forall x_1...x_k)W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$\Longrightarrow A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge (\exists x_1...x_k)\neg W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge (V \leftarrow W) \wedge W_{i+1} \wedge ... \wedge W_m$$
$$\Longrightarrow A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge V \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg(V \leftarrow W) \wedge W_{i+1} \wedge ... \wedge W_m$$
$$\Longrightarrow A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge W \wedge \neg V \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge (V \vee W) \wedge W_{i+1} \wedge ... \wedge W_m$$
$$\Longrightarrow A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge V \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg(V \vee W) \wedge W_{i+1} \wedge ... \wedge W_m$$
$$\Longrightarrow A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg V \wedge \neg W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg\neg W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$\Longrightarrow A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge (\exists x_1...x_n)W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$\Longrightarrow A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg(\exists x_1...x_n)W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$\Longrightarrow A \leftarrow W_1 \wedge ... \wedge W_{i-1} \wedge \neg p(y_1, ..., y_k)W \wedge W_{i+1} \wedge ... \wedge W_m$$
$$p(y_1, ..., y_k) \leftarrow (\exists x_1...x_n)W$$

It is clear that the resulting program is a normal program, since any clause that is not normal falls into some of the above syntactic cases. Furthermore, Lloyd and Topor prove that the SLDNF resolution process associated to the resulting normal program does not flounder if this resulting program is *allowed*. We give the definition of the (syntactic) allowed condition on normal programs, as presented in [23]:

A program clause $A \leftarrow L_1 \wedge ... \wedge L_n$ is *admissible* if every variable that occurs in the clause occurs either in the head A or in a positive literal among the L_i. A clause $A \leftarrow L_1 \wedge ... \wedge L_n$ is *allowed* if every variable in the clause appears in a positive literal of the body $L_1 \wedge ...L_n$. A goal G is *allowed* if every variable in the goal appears in some positive literal in the goal. A set of clauses $P \cup \{G\}$ is *allowed* if the following conditions are satisfied:

a) Every clause in P is admissible.

b) Every clause in the definition of a predicate symbol occuring in a positive literal in the body of G or in a positive literal in the body of a clause in P is allowed.

c) G is allowed.

Topor defines in [39] the (syntactic) concept of *allowed formula* and proves that the application of his and Lloyd's algorithm to allowed formulae always produces allowed programs. The definition of *allowed formula* is as follows:

A variable is *pos* (positive) in a formula φ if one of the following cases holds:

- x is pos in $p(t_1, ..., t_n)$ if x ocuurs in $p(t_1, ..., t_n)$ and p is not $=$;
- x is pos in $x = c$ or $c = x$ if c is a constant;
- x is pos in $\neg\psi$ if x is neg in ψ;
- x is pos in $\psi \wedge \chi$ if x is pos in ψ or x is pos in χ;
- x is pos in $\psi \vee \chi$ if x is pos in χ or x is pos in χ;
- x is pos in $(\exists y)\psi$ if it is pos in ψ;

– x is pos in $(\forall y)\psi$ if it is neg in ψ.

Similarly, x is *neg* (negative) in φ if one of the following cases hold:

– x is *neg* in $\neg\psi$ if x is *pos* in ψ;

– x is *neg* in $\psi \wedge \chi$ if x is *neg* in ψ or x is *neg* in χ;

– x is *neg* in $\psi \vee \chi$ if x is *neg* in ψ or x is *neg* in χ;

– x is *neg* in $\psi \supset \chi$ if x is *pos* in in ψ or x is *neg* in χ;

– x is *neg* in $(\forall x)\psi$ if x is *neg* in ψ.

A formula φ is *allowed* if the following conditions hold:

a) every free variable in φ is pos in φ;

b) for every subformula $(\exists x)\psi$ os φ, x is pos in ψ;

c) for every subformula $(\forall x)\psi$ of φ, x is neg in ψ.

5.5 Implementation Issues

Our implementation of the above algorithm consists essentially in the application of the above rules by means of a Prolog (meta-)program that generates a Prolog (object) program from formula IC and Σ_{S_0}, considered as the set (5.6) of facts. Literals in the object program are ordered as follows: positive literals are put first, then equalities and finally negative atoms. It is easy to see that Prolog's strategy, applied to such a program, emulates SLDNF resolution. In fact, in SLDNF resolution any literal in a goal can be chosen at each step or the derivation, with the exception of non-ground negative atoms. Since positive atoms bind variables, it is clear that by putting the negative atoms at the end of every clause we ensure that Prolog will never choose a negative non-ground atom unless the same choice would arise using SLDNF-resoultion. The only (relative) drawback of Prolog is its use of depth-first search in SLDNF-trees, which is known to be non-optimal in some cases.

We can conclude that if an integrity constraint is an allowed formula then the application of our implementation of Lloyd and Topor's algorithm will never flounder and produce the correct answer to the query $IC(S_0)$ (in this case, 'true' or 'false'), thus producing a correct proof or refutation of $IC(S_0)$ from the database updates specification.

VI. CONCLUSIONS

We provided in this thesis a general methodology for proving situation calculus database integrity constrints by means of inductive conditional term rewriting systems. We proved the mothodology to be sound and complete. The methodology solves among others the problem of first order quantification using term rewriting in the situation calculus database context. This solution can be further extended to other contexts or provide a starting point for a more general solution to the quantification problem in rewriting systems (besides situation calculus specifications). We implemented the solutions within the $SCDBR$ reasoner. Finally, we implemented also an alternative proof method for integrity constraints using resoultion based theorem provers, strongly relying on Topor and Lloyd's algorithm.

BIBLIOGRAPHY

[1] S. Abiteboul and V. Vianu. A Transaction–Based Approach to Relational Database Specification. *Journal of the ACM*, Vol. 36(Num. 4):pp. 758–789, October 1989.

[2] R. Aubin. Mechanizing structural induction. *Theoretical Computer Science*, 9:347–362, 1979.

[3] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Readings in Artificial Intelligence & Databases*, pages 376–430. Morgan Kaufmann, 1989.

[4] L. Bertossi, M. Arenas, and C. Ferretti. SCDBR: An Automated Reasoner for Database Updates (Extended Abstract). In *Proc. VXI International Conference of the Chilean Computer Science Society (SCCC'96), M. V. Zelkowitz and P. Straub (eds.)*, pages 43–55, 1996.

[5] L. Bertossi and J. C. 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*, pages 543–545. Springer, 1994.

[6] 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*, pages 212–222. Springer, LNAI 1079, 1996.

[7] R. S. Boyer and J S. Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM*, 31:441–458, 1984.

[8] R.S. Boyer and J S. Moore. *A Ccomputational Logic*. Academic Press, 1979.

[9] R.S. Boyer and J S. Moore. *A Ccomputational Logic Handbook*. Academic Press, 1988.

[10] 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.

[11] R. M. Burstall. Proving properties of programs by structural induction. *Comput. J.*, 12(1), 1969.

[12] C. J. Date. *An Introduction to Database Systems.* Addison Wesley, 4 edition, 1986.

[13] H. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[14] Leopoldo Bertossi et al. El Sistema SCDBR: Manual de Uso e Instalacion. Version 4. Technical report, Universidad Catolica de Chile, 1996.

[15] A. Van Gelder and R. Topor. Safety and Correct Translation of Relational Calculus Formulas. In *Proc. ACM Symposium on Database Systems (PODS)*, 1987.

[16] D. Kapur. Mechanizing Induction. Preliminary draft, Dept. Computer Science, State University of New York at Albany, 1995.

[17] D. Kapur and H. Zhang. RRL: A Rewrite Rule Laboratory. In Lusk and Overbeek, editors, *Proc. Ninth International Conference on Automated Deduction (CADE-9), LNCS 310*, pages 768–769. Springer, 1988.

[18] D. Kapur and H. Zhang. An Overview of Rewrite Rule Laboratory (RRL). *J. of Computer and Mathematics with Applications*, 29(2):91–114, 1995.

[19] Matt Kaufmann. First–Order Quantification with NQTHM. *Journal of Automated Reasoning*, Vol. 9(Number 3):pp. 355–372, December 1992.

[20] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[21] D. S. Lankford. A simple explanation of inductionless induction. Technical Report MTP-14, Mathematics Department, Lousiana Technical University, 1981.

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

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

[24] 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, pages 463–502, Edinburgh, Scotland, 1969. Edinburgh University Press.

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

[26] J. Strother Moore. The mechanization of induction in the boyer-moore theorem prover. In *Notes from the 2nd International Summer School in Logic for Computer Science*. Université de Savoie, Chambéry, France, 1994.

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

[28] Jean Marie Nicolas and K. Yazdanian. Integrity checking in deductive data bases. *Logic and databases*, 1978.

[29] François Bancilhon Raghu Ramakrishnan and Avi Silberschatz. Safety of recursive horn clauses with infinite relations. *PODS*, 87.

[30] 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*, pages 359–380. Academic Press, 1991.

[31] R. Reiter. Formalizing database evolution in the situation calculus. In *Proceedings of the Fifth Generation Computer Systems*, Tokyo, Japan, June 1992.

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

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

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

[35] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[36] R. Hull S. Abiteboul and V. Vianu. *Foundations of Databases.* Addison Wesley, 1995.

[37] T. Sheard and D. Stemple. Automatic Verification of Database Transaction Safety. *ACM Transactions on Database Systems*, 14(3):322–368, 1989.

[38] Jianwen Su. Dependency Preservation in Semantic Databases. *Acta Informatica*, 31:pp. 27–54, 1994.

[39] R. Topor. Domain-independent formulas and databases. *Theoretical Computer Science*, 52:281–306, 1987.

[40] J. Ullman. *Principles of Database and Knowledge-Base Systems, Vols. I and II.* Computer Science Press, 1988.

[41] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, chapter 6. Elsevier Science Publishers B. V., 1990.

[42] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning.* Mc Graw Hill, 2 edition, 1992.

[43] H. Zhang. Reduction, Superposition & Induction: Automated Reasoning in an Equational Logic. Technical report, Ph.D. Thesis, Dept. Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180–3590, 1988.

[44] H. Zhang and G. X. Hua. Proving Ramsey theorem by the cover set induction: A case and comparison study. *The Annals of Mathematic and Artificial Intelligence*, 8, 1992.

[45] H. Zhang and D. Kapur. First–Order Theorem Proving using Conditional Rewrite Rules. In Lusk and Overbeek, editors, *Proc. of Ninth International Conference on Automated Deduction Deduction (CADE-9), LNCS 310*, pages 1–20. Springer, 1988.

[46] H. Zhang, D. Kapur, and M.S. Krishnamoorthy. A Mechanizable Induction Principle for Equational Specifications. In Lusk and Overbeek, editors, *Proc. of Ninth International Conference on Automated Deduction Deduction (CADE-9), LNCS 310*, pages 162–181. Springer, 1988.

[47] Hantao Zhang and Xin Hua. Proving the chinese remainder theorem by the cover set induction. In *Proc. International Conference of Automated Deduction*, pages 431–445. Springer Verlag, 1992.

Appendixes

A. RRL'S SCRIPT

```
AKCL (Austin Kyoto Common Lisp)  Version(1.623) Tue Dec 21 23:54:10 -0300 1993
Contains Enhancements by W. Schelter

>
****************************************************************************
****                                                                    ****
****        WELCOME  TO  REWRITE  RULE  LABORATORY  (RRL 4.1)           ****
****                                                                    ****
****************************************************************************
Start at: 20:55:19 Sep 16 1999

Type Add, Akb, Auto, Break, Clean, Delete, Disable, Enable, Grammar, History,
     Init, Kb, List, Load, Log, Makerule, Manager, Namerule, Norm, Option,
     Operator, Prove, Quit, Read, Refute, Save, Stats, Suffice, Time, Undo,
     Unlog, Write or Help.
RRL-> Time for this command = 0.000 sec      Total time = 0.000 sec

Type Add, Akb, Auto, Break, Clean, Delete, Disable, Enable, Grammar, History,
     Init, Kb, List, Load, Log, Makerule, Manager, Namerule, Norm, Option,
     Operator, Prove, Quit, Read, Refute, Save, Stats, Suffice, Time, Undo,
     Unlog, Write or Help.
RRL-> OPTION

Type HELP, ACKB, AUTOORD, BRAKE, CRITICAL, CYCLE, FASTKB, FOPC, HISTORY,
     INDUC, LIST, MANUAL, NORM, OPERATOR, ORDER, PROVE, REDUCE, INSTANCE,
     SUPPORT, TRACE, TYPE, QUIT --> PROVE
Following methods are available:
  Deduction:
     (f) forward reasoning --- using completion
     (s) refutation        --- using boolean-ring method
     (c) refutation        --- using conditional rules
  Induction:
     (e) explicit          --- using the cover-set method
     (s) inductionless     --- using the test set method
     (g) inductionless     --- using the ground-reducibility method
Please make your choice:
(current value: S) (F,C,E,S,G) E

The maximal depth of conditional rewriting is set to 2.
Time for this command = 0.017 sec      Total time = 0.017 sec

Type Add, Akb, Auto, Break, Clean, Delete, Disable, Enable, Grammar, History,
     Init, Kb, List, Load, Log, Makerule, Manager, Namerule, Norm, Option,
     Operator, Prove, Quit, Read, Refute, Save, Stats, Suffice, Time, Undo,
     Unlog, Write or Help.
RRL-> ADD
Type your equations, enter a ']' when done.
```

```
New operators have the arities:
      [BEXIST : OBJECT, STATE, DOMAIN -> BOOL ]
      [BALL : OBJECT, STATE, DOMAIN -> BOOL ]
      [C : OBJECT, DOMAIN -> DOMAIN ]
      [NOTHING : -> DOMAIN ]
      [BDOM : STATE -> DOMAIN ]
      [NEWITEM : OBJECT -> ACT ]
      [ORDER : OBJECT -> ACT ]
      [REACH : STATE -> BOOL ]
      [DO : ACT, STATE -> STATE ]
      [S0 : -> STATE ]
      [POSS : ACT, STATE -> BOOL ]
      [PROVIDES : OBJECT, OBJECT, STATE -> BOOL ]
      [PROVIDER : OBJECT, STATE -> BOOL ]
      [INSTOCK : OBJECT, STATE -> BOOL ]
      [ORDEREDA : OBJECT, STATE -> BOOL ]

New constant set is: { P1, P2, NOTHING, S0 }
Equations read in are:
 1. REACH(S0)  [DEF, 1]
 2. REACH(DO(XA, XS)) == REACH(XS) & POSS(XA, XS)  [DEF, 2]
 3. ORDEREDA(XPROD, DO(XA, XS)) == COND(ORDEREDA(XPROD, XS), TRUE, ((XA = ORDER(XPROD)) AND BALL(XPROD, XS, BDOM(XS))))  [DEF, 3]
 4. INSTOCK(XPROD, DO(XA, XS)) == COND(INSTOCK(XPROD, XS), TRUE, (XA = NEWITEM(XPROD)))  [DEF, 4]
 5. ORDEREDA(XPROD, S0) == (XPROD = P2)  [DEF, 5]
 6. INSTOCK(XPROD, S0) == COND((XPROD = P1), TRUE, (XPROD = P2))  [DEF, 6]
 7. POSS(ORDER(XPROD), XS) == COND(INSTOCK(XPROD, XS), TRUE, BEXIST(XPROD, XS, BDOM(XS)))  [DEF, 7]
 8. POSS(NEWITEM(XPROD), XS)  [DEF, 8]
 9. BALL(XPROD, XS, NOTHING)  [DEF, 9]
10. BALL(XPROD, XS, C(X, XDOM)) == BALL(XPROD, XS, XDOM) & COND(PROVIDER(X, XS), PROVIDES(X, XPROD, XS), TRUE)  [DEF, 10]
11. BEXIST(XPROD, XS, NOTHING) == FALSE  [DEF, 11]
12. BEXIST(XPROD, XS, C(X, XDOM)) == COND(BEXIST(XPROD, XS, XDOM), TRUE, (PROVIDER(X, XS) AND PROVIDES(X, XPROD, XS)))  [DEF, 12]
13. BDOM(S0) == C(P1, C(P2, NOTHING))  [DEF, 13]
14. BDOM(DO(XA, XS)) == C(PRODUCTIN(XA), BDOM(XS))  [DEF, 14]
15. BEXIST(XPROD, XS, BDOM(XS)) == TRUE  if  PROVIDER(X, XS) & PROVIDES(X, XPROD, XS)  [DEF, 15]
16. BALL(XPROD, XS, BDOM(XS)) == FALSE  if  PROVIDES(X, XPROD, XS) & PROVIDER(X, XS)  [DEF, 16]
Time for this command = 0.350 sec      Total time = 0.383 sec

Type Add, Akb, Auto, Break, Clean, Delete, Disable, Enable, Grammar, History,
     Init, Kb, List, Load, Log, Makerule, Manager, Namerule, Norm, Option,
     Operator, Prove, Quit, Read, Refute, Save, Stats, Suffice, Time, Undo,
     Unlog, Write or Help.
RRL-> OPERATOR

Type HELP, ACOPERATOR, COMMUTATIVE, CONSTRUCTOR, DISPLAY, DIVISIBLE,
     EQUIVALENCE, PRECEDENCE, QUIT, STATUS, TRANSITIVE, WEIGHT, QUIT --> CONSTRUCTOR
Type operators you wish to be CONSTRUCTORS: DO S0 ORDER NEWITEM C NOTHING
Is 'DO' a free constructor ? (Y,N,YES,NO) Y
Is 'ORDER' a free constructor ? (Y,N,YES,NO) Y
Is 'NEWITEM' a free constructor ? (Y,N,YES,NO) Y
Is 'C' a free constructor ? (Y,N,YES,NO) Y
Time for this command = 0.017 sec      Total time = 0.400 sec
```

```
Type Add, Akb, Auto, Break, Clean, Delete, Disable, Enable, Grammar, History,
     Init, Kb, List, Load, Log, Makerule, Manager, Namerule, Norm, Option,
     Operator, Prove, Quit, Read, Refute, Save, Stats, Suffice, Time, Undo,
     Unlog, Write or Help.
RRL-> MAKERULE

Adding Rule:   [1] REACH(S0) ---> TRUE  [DEF, 1]

Trying to orient equation:
  REACH(DO(XA, XS)) == REACH(XS) & POSS(XA, XS)  [DEF, 2]

  To prove:  REACH(DO(XA, XS))  >  REACH(XS) & POSS(XA, XS)
  Here are some precedence suggestions:
       1.  REACH > POSS
Either type a list of numbers or
Type Abort, Display, Drop, Equiv, LR, MakeEq, Operator, Postpone, Quit, RL,
     Status, Twoway, Undo or Help.
RRL>ORDER> 1

Adding Rule:   [2] REACH(DO(XA, XS)) ---> REACH(XS) & POSS(XA, XS)  [DEF, 2]

Trying to orient equation:
  ORDEREDA(XPROD, DO(XA, XS)) == COND(ORDEREDA(XPROD, XS), TRUE, ((ORDER(XPROD) = XA) AND BALL(XPROD, XS, BDOM(XS))))  [DEF, 3]

  To prove:  ORDEREDA(XPROD, DO(XA, XS))  >  COND(ORDEREDA(XPROD, XS), TRUE, ((ORDER(XPROD) = XA) AND BALL(XPROD, XS, BDOM(XS))))
  Here are some precedence suggestions:
       1.  ORDEREDA > BALL
       2.  ORDEREDA > BDOM
       3.  DO > ORDER
Either type a list of numbers or
Type Abort, Display, Drop, Equiv, LR, MakeEq, Operator, Postpone, Quit, RL,
     Status, Twoway, Undo or Help.
RRL>ORDER> 1 2 3

Adding Rule:   [3] ORDEREDA(XPROD, DO(XA, XS)) ---> COND(ORDEREDA(XPROD, XS), TRUE, ((ORDER(XPROD) = XA) AND BALL(XPROD, XS, BDOM(XS))))  [DEF, 3]

Adding Rule:   [4] INSTOCK(XPROD, DO(XA, XS)) ---> COND(INSTOCK(XPROD, XS), TRUE, (NEWITEM(XPROD) = XA))  [DEF, 4]

Adding Rule:   [5] ORDEREDA(XPROD, S0) ---> (P2 = XPROD)  [DEF, 5]

Adding Rule:   [6] INSTOCK(XPROD, S0) ---> COND((P1 = XPROD), TRUE, (P2 = XPROD))  [DEF, 6]

Trying to orient equation:
  POSS(ORDER(XPROD), XS) == COND(INSTOCK(XPROD, XS), TRUE, BEXIST(XPROD, XS, BDOM(XS)))  [DEF, 7]

  To prove:  POSS(ORDER(XPROD), XS)  >  COND(INSTOCK(XPROD, XS), TRUE, BEXIST(XPROD, XS, BDOM(XS)))
  Here are some precedence suggestions:
       1.  POSS > INSTOCK
       2.  POSS > BEXIST
       3.  POSS > BDOM
```

```
Either type a list of numbers or
Type Abort, Display, Drop, Equiv, LR, MakeEq, Operator, Postpone, Quit, RL,
     Status, Twoway, Undo or Help.
RRL>ORDER> 1 2 3

Adding Rule:   [7] POSS(ORDER(XPROD), XS) ---> COND(INSTOCK(XPROD, XS), TRUE, BEXIST(XPROD, XS, BDOM(XS)))  [DEF, 7]

Adding Rule:   [8] POSS(NEWITEM(XPROD), XS) ---> TRUE  [DEF, 8]

Adding Rule:   [9] BALL(XPROD, XS, NOTHING) ---> TRUE  [DEF, 9]

Trying to orient equation:
  BALL(XPROD, XS, C(X, XDOM)) == BALL(XPROD, XS, XDOM) & COND(PROVIDER(X, XS), PROVIDES(X, XPROD, XS), TRUE)  [DEF, 10]

  To prove:  BALL(XPROD, XS, C(X, XDOM))  >  BALL(XPROD, XS, XDOM) & COND(PROVIDER(X, XS), PROVIDES(X, XPROD, XS), TRUE)
  Here are some precedence suggestions:
       1.  BALL > PROVIDER
       2.  BALL > PROVIDES
Either type a list of numbers or
Type Abort, Display, Drop, Equiv, LR, MakeEq, Operator, Postpone, Quit, RL,
     Status, Twoway, Undo or Help.
RRL>ORDER> 1 2 3

Adding Rule:  [10] BALL(XPROD, XS, C(X, XDOM)) ---> BALL(XPROD, XS, XDOM) & COND(PROVIDER(X, XS), PROVIDES(X, XPROD, XS), TRUE)  [DEF, 10]

Adding Rule:  [11] BEXIST(XPROD, XS, NOTHING) ---> FALSE  [DEF, 11]

Trying to orient equation:
  BEXIST(XPROD, XS, C(X, XDOM)) == COND(BEXIST(XPROD, XS, XDOM), TRUE, (PROVIDER(X, XS) AND PROVIDES(X, XPROD, XS)))  [DEF, 12]

  To prove:  BEXIST(XPROD, XS, C(X, XDOM))  >  COND(BEXIST(XPROD, XS, XDOM), TRUE, (PROVIDER(X, XS) AND PROVIDES(X, XPROD, XS)))
  Here are some precedence suggestions:
       1.  BEXIST > PROVIDER
       2.  BEXIST > PROVIDES
Either type a list of numbers or
Type Abort, Display, Drop, Equiv, LR, MakeEq, Operator, Postpone, Quit, RL,
     Status, Twoway, Undo or Help.
RRL>ORDER> 1 2

Adding Rule:  [12] BEXIST(XPROD, XS, C(X, XDOM)) ---> COND(BEXIST(XPROD, XS, XDOM), TRUE, (PROVIDER(X, XS) AND PROVIDES(X, XPROD, XS)))  [DEF, 12]

Adding Rule:  [13] BDOM(S0) ---> C(P1, C(P2, NOTHING))  [DEF, 13]

Trying to orient equation:
  BDOM(DO(XA, XS)) == C(PRODUCTIN(XA), BDOM(XS))  [DEF, 14]

  To prove:  BDOM(DO(XA, XS))  >  C(PRODUCTIN(XA), BDOM(XS))
  Here are some precedence suggestions:
       1.  BDOM > PRODUCTIN
       2.  DO > C
Either type a list of numbers or
```

```
Type Abort, Display, Drop, Equiv, LR, MakeEq, Operator, Postpone, Quit, RL,
     Status, Twoway, Undo or Help.
RRL>ORDER> 1 2

Adding Rule:  [14] BDOM(DO(XA, XS)) ---> C(PRODUCTIN(XA), BDOM(XS))  [DEF, 14]

Following rule has extra variables in its condition:
     [15] BEXIST(XPROD, XS, BDOM(XS)) ---> TRUE  if
        { PROVIDER(X, XS),
          PROVIDES(X, XPROD, XS) }   [DEF, 15]

Following rule has extra variables in its condition:
     [16] BALL(XPROD, XS, BDOM(XS)) ---> FALSE  if
        { PROVIDER(X, XS),
          PROVIDES(X, XPROD, XS) }   [DEF, 16]
Time for this command = 0.217 sec      Total time = 0.617 sec

Type Add, Akb, Auto, Break, Clean, Delete, Disable, Enable, Grammar, History,
     Init, Kb, List, Load, Log, Makerule, Manager, Namerule, Norm, Option,
     Operator, Prove, Quit, Read, Refute, Save, Stats, Suffice, Time, Undo,
     Unlog, Write or Help.
RRL-> PROVE
Type your equation in the format:  L == R [if C]
Enter a ']' to exit when no equation is given.
BEXIST(XPROD, XS, BDOM(XS1)) == FALSE if (BALL(XPROD, XS, BDOM(XS)) AND (REACH(XS) AND (NOT(INSTOCK(XPROD, XS)) AND (NOT(ORDEREDA(XPROD, XS)) AND HINT(INDUC(BDOM(XS1)))))))


Specification of 'ORDEREDA' is completely defined.

Specification of 'INSTOCK' is completely defined.

Specification of 'POSS' is completely defined.

Specification of 'REACH' is completely defined.

Specification of 'BDOM' is completely defined.

Specification of 'BALL' is completely defined.

Specification of 'BEXIST' is completely defined.

Proving [main] BEXIST(XPROD, XS, BDOM(XS1)) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (HINT(INDUC(BDOM(XS1)))) and
          (BALL(XPROD, XS, BDOM(XS)))
    by HINT(INDUC(BDOM(XS1)))

Let P(XS1) be [main] BEXIST(XPROD, XS, BDOM(XS1)) == FALSE if
          (REACH(XS)) and
```

```
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS)))

The induction will be done on XS1 in BDOM(XS1), and will follow the scheme:
    [#1] P(S0)
    [#2] P(DO(XA, XS1)) if  { P(XS1) }

By rules [13], [12], [11],
     [#1] BEXIST(XPROD, XS, BDOM(S0)) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS)))
     is reduced to
     [#1] COND((PROVIDER(P2, XS) AND PROVIDES(P2, XPROD, XS)), TRUE, (PROVIDER(P1, XS) AND PROVIDES(P1, XPROD, XS))) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS)))

Proving [#1] COND((PROVIDER(P2, XS) AND PROVIDES(P2, XPROD, XS)), TRUE, (PROVIDER(P1, XS) AND PROVIDES(P1, XPROD, XS))) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS)))
          under the condition (PROVIDER(P2, XS) AND PROVIDES(P2, XPROD, XS))
          and its negation.

By rule [16],
     [#1.1] TRUE == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (PROVIDER(P2, XS)) and
          (PROVIDES(P2, XPROD, XS)) and
          (BALL(XPROD, XS, BDOM(XS)))
     is reduced to true.

By rule [16],
     [#1.2] (PROVIDER(P1, XS) AND PROVIDES(P1, XPROD, XS)) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (not(PROVIDER(P2, XS))) and
          (not(PROVIDES(P2, XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS)))
     is reduced to true.

Following equation is proven successfully by case-split:
     [#1] COND((PROVIDER(P2, XS) AND PROVIDES(P2, XPROD, XS)), TRUE, (PROVIDER(P1, XS) AND PROVIDES(P1, XPROD, XS))) == FALSE if
```

```
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS)))

By rules [14], [12], [16],
     [#2] BEXIST(XPROD, XS, BDOM(DO(XA, XS1))) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS))) and
          (not(BEXIST(XPROD, XS, BDOM(XS1))))
     is reduced to true.

All subgoals of [main] are proven, hence
    [main] BEXIST(XPROD, XS, BDOM(XS1)) == FALSE if
          (REACH(XS)) and
          (not(INSTOCK(XPROD, XS))) and
          (not(ORDEREDA(XPROD, XS))) and
          (BALL(XPROD, XS, BDOM(XS)))
    is an inductive theorem.

Following equation
    BEXIST(XPROD, XS, BDOM(XS1)) == FALSE  if
        { HINT(INDUC(BDOM(XS1))),
          REACH(XS),
          BALL(XPROD, XS, BDOM(XS)),
          not(INSTOCK(XPROD, XS)),
          not(ORDEREDA(XPROD, XS)) }   [USER, 17]
    is an inductive theorem in the current system.

Adding Rule:  [17] BEXIST(XPROD, XS, BDOM(XS1)) ---> FALSE  if
        { REACH(XS),
          not(INSTOCK(XPROD, XS)),
          not(ORDEREDA(XPROD, XS)),
          BALL(XPROD, XS, BDOM(XS)) }   [USER, 17]


Time for this command = 0.217 sec      Total time = 0.833 sec

Type Add, Akb, Auto, Break, Clean, Delete, Disable, Enable, Grammar, History,
     Init, Kb, List, Load, Log, Makerule, Manager, Namerule, Norm, Option,
     Operator, Prove, Quit, Read, Refute, Save, Stats, Suffice, Time, Undo,
     Unlog, Write or Help.
RRL-> PROVE
Type your equation in the format:  L == R [if C]
Enter a ']' to exit when no equation is given.
INSTOCK(XPROD, XS) == TRUE if (ORDEREDA(XPROD, XS) AND REACH(XS))


Let P(XS) be [main] INSTOCK(XPROD, XS) == TRUE if
```

```
          (REACH(XS)) and
          (ORDEREDA(XPROD, XS))

The induction will be done on XS in REACH(XS), and will follow the scheme:
    [#1] P(S0)
    [#2] P(DO(XA, XS1)) if  { P(XS1) }

By rules [1], [5], [6],
     [#1] INSTOCK(XPROD, S0) == TRUE if
          (REACH(S0)) and
          (ORDEREDA(XPROD, S0))
     is reduced to true.

Subgoal [#2] is split into:
 [#2.1] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS))) and
          (not(REACH(XS)))
 [#2.2] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS))) and
          (not(ORDEREDA(XPROD, XS)))
 [#2.3] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS))) and
          (INSTOCK(XPROD, XS))

By rule [2],
     [#2.1] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (not(REACH(XS))) and
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS)))
     is reduced to true.

By rules [2], [3], [4], [7], [17], and by switching the head and the premises,
     [#2.2] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (not(ORDEREDA(XPROD, XS))) and
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS)))
     is reduced to true.

By rules [2], [3], [4],
     [#2.3] INSTOCK(XPROD, DO(XA, XS)) == TRUE if
          (INSTOCK(XPROD, XS)) and
          (REACH(DO(XA, XS))) and
          (ORDEREDA(XPROD, DO(XA, XS)))
     is reduced to true.

All subgoals of [main] are proven, hence
    [main] INSTOCK(XPROD, XS) == TRUE if
          (REACH(XS)) and
```

```
         (ORDEREDA(XPROD, XS))
    is an inductive theorem.

Following equation
    INSTOCK(XPROD, XS) == TRUE  if
        { REACH(XS),
          ORDEREDA(XPROD, XS) }   [USER, 18]
    is an inductive theorem in the current system.

Adding Rule:  [18] INSTOCK(XPROD, XS) ---> TRUE  if
        { REACH(XS),
          ORDEREDA(XPROD, XS) }   [USER, 18]


Time for this command = 0.083 sec      Total time = 0.933 sec

Type Add, Akb, Auto, Break, Clean, Delete, Disable, Enable, Grammar, History,
     Init, Kb, List, Load, Log, Makerule, Manager, Namerule, Norm, Option,
     Operator, Prove, Quit, Read, Refute, Save, Stats, Suffice, Time, Undo,
     Unlog, Write or Help.
RRL-> QUIT
No log file.

Goodbye NIL.
Finish at: 20:55:33 Sep 16 1999


Bye.
```

B. SOTWARE DOCUMENTATION

We include in this appendix the source code of our implementation, along with its documentation.

B.1 RRL

Module: `inter-rrl.pl`

Description: Provides an elementary interface between SCDBR and RRL and manages the interaction between both programs.

```
for(implies(and(above(obj1,obj2,s),above(obj3,obj2,s)),equal(obj1,obj3))).
prove_rrl(F,VV,AAA,BBB,CCC,DDD) :-
  retractall(constr(_,_)),
  trad_spec(AAA),
  write_ball(F,VV,[0,2],AAA),
  write_file(BBB,'option prove e'),
  nl_file(BBB),
  write_file(BBB,'read '),
  write_file(BBB,AAA),
  nl_file(BBB),
  write_file(BBB,'ope con'),
  findall(X,constr(X,_),L),
  write1_int(L,BBB),
  findall(Y,constr_dom(Y,_),LL),
  write4_int(LL,BBB),
  nl_file(BBB),
  findall(Z,constr(Z,free),L1),
  findall(W,constr_dom(W,free),LL1),
  append(L1,LL1,L2),
  write2_int(L2,BBB),
  write_file(BBB,'make'),
  nl_file(BBB),
  makeeq(BBB,40),
  write_file(bbb,'(start-rrl)'),
  nl_file(bbb),
  write_file(bbb,'auto |'),
  write_file(bbb,BBB),
  write_file(bbb,'|'),
  nl_file(bbb),
  write_file(bbb,'auto |'),
  write_file(bbb,CCC),
```

```
  write_file(bbb,'|'),
  nl_file(bbb),
  write_file(CCC,'prove'),
  nl_file(CCC),
  write_file(CCC,'reach(xs) == false if not('),
  write1_rrl(F,VV,[0,2],CCC),
  write_file(CCC,')'),
  nl_file(CCC),
  write_file(CCC,'abort'),
  nl_file(CCC),
  write_file(CCC,'quit'),
  nl_file(CCC),
  text_append(['rrl.exe < bbb > ',DDD],E),
  unix_call(E).

write1_int([],_).
write1_int([H|T],A) :-
  write_file(A,' a'),
  write_file(A,H),
  write1_int(T,A).

write4_int([],_).
write4_int([H|T],A) :-
  write_file(A,' '),
  write_file(A,H),
  write4_int(T,A).

write2_int([],_).
write2_int([_|T],A) :-
  write_file(A,'y'),
  nl_file(A),
  write2_int(T,A).

makeeq(_,0).
makeeq(A,N) :-
  write3_int(20,A),
  nl_file(A),
  N1 is N - 1,
  makeeq(A,N1).

write3_int(0,_).
write3_int(X,O) :-
  X1 is X-1,
  write3_int(X1,O),
  write_file(O,X),
  write_file(O,' ').
```

Module: `trad.pl`

Description: Translates a situation calculus specification into a conditional equational language, using the syntax of the RRL theorem prover. It behaves according to algorithm *btrans* described in section 3.4.

```
trad_spec(A) :-
  assert(constr(s0,nfree)),
  assert(constr(do,free)),
  findall(X,fluent(X,_,_),L),
  write_file(A,'[as0 : astate]'),
  nl_file(A),
  write_file(A,'[ado : act , astate -> astate]'),
  nl_file(A),
  write_ssa_rrl(L,0,A),
  write_file(A,'[aposs : act , astate -> bool]'),
  nl_file(A),
  findall(X,action(X,_,_),L2),
  write_poss_rrl(L2,0,A),
  write_file(A,'[reach : astate -> bool]'),
  nl_file(A),
  write_file(A,'reach(as0) := true'),
  nl_file(A),
  write_reach_rrl(L2,A),
  findall([Z,Y],fluent(Z,_,Y),L3),
  write_s0_rrl(L3,A),
  c_initial(LI),
  uno(LI,UNO),
  write_uno(UNO,A).

write_ssa_rrl([X|T],N,A) :-
  ssa(X,all(a,implies(poss(a,s),iff(H,F)))),
  write_file(A,'[a'),
  write_file(A,X),
  write_file(A,' : '),
  fluent(X,_,Y),
  write6_rrl(Y,A),
  write_file(A,' -> bool]'),
  nl_file(A),
  write_term_rrl(H,A),
  write_file(A,' := '),
  v_f(H,V),
  write1_rrl(F,V,[N,0],A),
  nl_file(A),
  write_ball(F,V,[N,0],A),
  N1 is N + 1,
  write_ssa_rrl(T,N1,A).
write_ssa_rrl([],_,_).

write_poss_rrl([X|T],N,A) :-
  action(X,_,Y),
  write_file(A,'[a'),
  write_file(A,X),
```

```
  write_file(A,' : '),
  write6_rrl(Y,A),
  write_file(A,' -> act]'),
  nl_file(A),
  ap(X,all(_,iff(H,F))),
  write_term_rrl(H,A),
  write_file(A,' := '),
  v_f(H,V),
  write1_rrl(F,V,[N,1],A),
  nl_file(A),
  write_ball(F,V,[N,1],A),
  N1 is N + 1,
  write_poss_rrl(T,N1,A).
write_poss_rrl([],_,_).

write_reach_rrl([X|T],A) :-
  ap(X,all(_,iff(poss(Y,_),_))),
  type_constr(Y,Ty),
  assert(constr(X,Ty)),
  write_file(A,'reach(ado('),
  write_term_rrl(Y,A),
  write_file(A,',xs)) := aposs('),
  write_term_rrl(Y,A),
  write_file(A,',xs) and reach(xs)'),
  nl_file(A),
  write_reach_rrl(T,A).
write_reach_rrl([],_).

type_constr(Y,Ty) :-
  v_f(Y,L),
  preg(L,Ty).

preg([],nfree).
preg([_|_],free).

write1_rrl(and(F1,F2),V,N,A) :-
  write1_rrl(F1,V,[0|N],A),
  write_file(A,' and '),
  write1_rrl(F2,V,[1|N],A).
write1_rrl(or(F1,F2),V,N,A) :-
  write_file(A,'cond('),
  write1_rrl(F1,V,[0|N],A),
  write_file(A,',true,'),
  write1_rrl(F2,V,[1|N],A),
  write_file(A,')').
write1_rrl(not(F),V,N,A) :-
  write_file(A,'not('),
  write1_rrl(F,V,N,A),
  write_file(A,')').
write1_rrl(implies(F1,F2),V,N,A) :-
  write_file(A,'cond('),
```

```
  write1_rrl(F1,V,[0|N],A),
  write_file(A,','),
  write1_rrl(F2,V,[1|N],A),
  write_file(A,',true)').
write1_rrl(iff(F1,F2),V,N,A) :-
  write_file(A,'cond('),
  write1_rrl(F1,V,[0|N],A),
  write_file(A,','),
  write1_rrl(F2,V,[1|N],A),
  write_file(A,',not('),
  write1_rrl(F2,V,[1|N],A),
  write_file(A,'))').
write1_rrl(all(_,_),V,N,A) :-
  write_file(A,'ball'),
  write2_rrl(N,A),
  write_file(A,'('),
  write3_rrl(V,A),
  write_file(A,',dom'),
  write2_rrl(N,A),
  write_file(A,'(xs))').
write1_rrl(some(_,_),V,N,A) :-
  write_file(A,'bexist'),
  write2_rrl(N,A),
  write_file(A,'('),
  write3_rrl(V,A),
  write_file(A,',dom'),
  write2_rrl(N,A),
  write_file(A,'(xs))').
write1_rrl(T,_,_,A) :-
  write_term_rrl(T,A).

write_ball(and(F1,F2),V,N,A) :-
  write_ball(F1,V,[0|N],A),
  write_ball(F2,V,[1|N],A).
write_ball(or(F1,F2),V,N,A) :-
  write_ball(F1,V,[0|N],A),
  write_ball(F2,V,[1|N],A).
write_ball(not(F),V,N,A) :-
  write_ball(F,V,N,A).
write_ball(implies(F1,F2),V,N,A) :-
  write_ball(F1,V,[0|N],A),
  write_ball(F2,V,[1|N],A).
write_ball(iff(F1,F2),V,N,A) :-
  write_ball(F1,V,[0|N],A),
  write_ball(F2,V,[1|N],A).
write_ball(all(X,F),V,N,A) :-
  write_domain(N,X,A),
  write_file(A,'[ball'),
  write_def(N,V,A),
  write_file(A,'ball'),
  write_initial(N,V,A),
```

```
  write_file(A,') := true'),
  nl_file(A),
  write_file(A,'ball'),
  write_ind_1(N,V,A,X),
  write_file(A,')) := ball'),
  write_ind_2(N,V,A),
  write_file(A,') and '),
  write1_rrl(F,[X|V],[0|N],A),
  nl_file(A),
  write_file(A,'ball'),
  write2_rrl(N,A),
  write_file(A,'('),
  write3_rrl(V,A),
  write_file(A,',dom'),
  write2_rrl(N,A),
  write_file(A,'(xs))'),
  write_file(A,':= false if not('),
  write1_rrl(F,[X|V],[0|N],A),
  write_file(A,')'),
  nl_file(A),
  write_ball(F,[X|V],[0|N],A).
write_ball(some(X,F),V,N,A) :-
  write_domain(N,X,A),
  write_file(A,'[bexist'),
  write_def(N,V,A),
  write_file(A,'bexist'),
  write_initial(N,V,A),
  write_file(A,') := false'),
  nl_file(A),
  write_file(A,'bexist'),
  write_ind_1(N,V,A,X),
  write_file(A,')) := bexist'),
  write_ind_2(N,V,A),
  write_file(A,') or '),
  write1_rrl(F,[X|V],[0|N],A),
  nl_file(A),
  write_file(A,'bexist'),
  write2_rrl(N,A),
  write_file(A,'('),
  write3_rrl(V,A),
  write_file(A,',dom'),
  write2_rrl(N,A),
  write_file(A,'(xs)) := true if '),
  write1_rrl(F,[X|V],[0|N],A),
  nl_file(A),
  write_ball(F,[X|V],[0|N],A).
write_ball(_,_,_,_).

write_ind_2(N,V,A) :-
  write2_rrl(N,A),
  write_file(A,'('),
```

```
  write3_rrl(V,A),
  write_file(A,',y'),
  write2_rrl(N,A).

write_initial(N,V,A) :-
  write2_rrl(N,A),
  write_file(A,'('),
  write3_rrl(V,A),
  write_file(A,',nothing'),
  write2_rrl(N,A).

write_def(N,V,A) :-
  write2_rrl(N,A),
  write_file(A,' : '),
  write4_rrl(V,A),
  write_file(A,' , domain'),
  write2_rrl(N,A),
  write_file(A,' -> bool]'),
  nl_file(A).

write_domain(N,X,A) :-
  write_file(A,'[nothing'),
  write2_rrl(N,A),
  write_file(A,' : domain'),
  write2_rrl(N,A),
  write_file(A,']'),
  nl_file(A),
  gen_name('nothing',N,Z),
  assert(constr_dom(Z,nfree)),
  write_file(A,'[c'),
  write2_rrl(N,A),
  write_file(A,' : a'),
  type_variable(X,T),
  write_file(A,T),
  write_file(A,' , domain'),
  write2_rrl(N,A),
  write_file(A,' -> domain'),
  write2_rrl(N,A),
  write_file(A,']'),
  nl_file(A),
  gen_name('c',N,ZZ),
  assert(constr_dom(ZZ,free)),
  write_file(A,'[dom'),
  write2_rrl(N,A),
  write_file(A,' : astate -> domain'),
  write2_rrl(N,A),
  write_file(A,']'),
  nl_file(A),
  findall(ZZZ,action(ZZZ,_,_),L),
  write_ind_dom(L,T,N,A),
  write_s0_dom(T,N,A).
```

```
write_ind_dom([X|Tail],T,N,A) :-
  ap(X,all(_,iff(poss(Ac,_),_))),
  v_f(Ac,V),
  write_file(A,'dom'),
  write2_rrl(N,A),
  write_file(A,'(ado('),
  write_term_rrl(Ac,A),
  write_file(A,',xs)) := '),
  write5_rrl(V,N,T,A),
  nl_file(A),
  write_ind_dom(Tail,T,N,A).
write_ind_dom([],_,_,_).

write5_rrl([V|Tail],N,T,A) :-
  type_variable(V,T),
  write_file(A,'c'),
  write2_rrl(N,A),
  write_file(A,'(x'),
  write_file(A,V),
  write_file(A,','),
  write5_rrl(Tail,N,T,A),
  write_file(A,')').
write5_rrl([V|Tail],N,T,A) :-
  \+ type_variable(V,T),
  write5_rrl(Tail,N,T,A).
write5_rrl([],N,_,A) :-
  write_file(A,'dom'),
  write2_rrl(N,A),
  write_file(A,'(xs)').

write_ind_1(N,V,A,X) :-
  write2_rrl(N,A),
  write_file(A,'('),
  write3_rrl(V,A),
  write_file(A,',c'),
  write2_rrl(N,A),
  write_file(A,'(x'),
  write_file(A,X),
  write_file(A,',y'),
  write2_rrl(N,A).

write2_rrl([N|T],A) :-
  write_file(A,'_'),
  write_file(A,N),
  write2_rrl(T,A).
write2_rrl([],_).

write3_rrl([V|[V1|T]],A) :-
  write_file(A,'x'),
  write_file(A,V),
```

```
  write_file(A,','),
  write3_rrl([V1|T],A).
write3_rrl([V],A) :-
  write_file(A,'x'),
  write_file(A,V).

write_3_rrl([V|[V1|T]],A) :-
  write_file(A,'x'),
  write_file(A,V),
  write_file(A,','),
  write_3_rrl([V1|T],A).
write_3_rrl([_],A) :-
  write_file(A,'as0').

write6_rrl([V|[V1|T]],A) :-
  write_file(A,'a'),
  write_file(A,V),
  write_file(A,' , '),
  write6_rrl([V1|T],A).
write6_rrl([V],A) :-
  write_file(A,'a'),
  write_file(A,V).

write4_rrl([V|[V1|T]],A) :-
  write_file(A,'a'),
  type_variable(V,Ty),
  write_file(A,Ty),
  write_file(A,' , '),
  write4_rrl([V1|T],A).
write4_rrl([V],A) :-
  write_file(A,'a'),
  type_variable(V,Ty),
  write_file(A,Ty).

write_s0_rrl([],_).
write_s0_rrl([[F,Ty]|T],A) :-
  gen_var(Ty,V,[]),
  initial(L),
  select_rrl(F,L,LL),
  write_s0_1(F,V,LL,A),
  write_s0_rrl(T,A).

gen_var([],[],_).
gen_var([H|T],[V|TT],L) :-
  belongs(H,L,N,LL),
  text_append([H,N],V),
  gen_var(T,TT,LL).

belongs(T,[],1,[[T,1]]).
belongs(T,[[T,N]|L],N1,[[T,N1]|L]) :-
  N1 is N+1.
```

```
belongs(T,[[TT,N]|L],N1,[[TT,N]|L1]) :-
  \+ (T = TT),
  belongs(T,L,N1,L1).

select_rrl(_,[],[]).
select_rrl(F,[T|L],L1) :-
  T =..[H|_],
  resul(F,T,H,L,L1).

resul(F,T,F,L,[T|L1]) :-
  select_rrl(F,L,L1).
resul(F,_,FF,L,L1) :-
  \+ (F = FF),
  select_rrl(F,L,L1).

write_s0_1(F,V,L,A) :-
  write_file(A,'a'),
  write_file(A,F),
  write_file(A,'('),
  write_3_rrl(V,A),
  write_file(A,') := '),
  write_7_rrl(V,L,A),
  nl_file(A).

write_7_rrl(_,[],A) :-
  write_file(A,'false').
write_7_rrl(V,[H|T],A) :-
  write_file(A,'cond('),
  H =..[_|TT],
  write_8_rrl(V,TT,A),
  write_file(A,',true,'),
  write_7_rrl(V,T,A),
  write_file(A,')').
write_8_rrl([_],[_],A) :-
  write_file(A,'true').
write_8_rrl([V|T1],[X|T2],A) :-
  write_file(A,'(x'),
  write_file(A,V),
  write_file(A,' = '),
  write_term_rrl(X,A),
  write_file(A,') and '),
  write_8_rrl(T1,T2,A).
write_8_rrl([],[],A) :-
  write_file(A,'true').

gen_name(X,N,Z) :-
  gen_name_1(N,NN),
  text_append([X,NN],Z).

gen_name_1([],'').
gen_name_1([H|T],Z) :-
```

```
  gen_name_1(T,ZZ),
  text_append(['_',H,ZZ],Z).

write_s0_dom(T,N,A) :-
  write_file(A,'dom'),
  write2_rrl(N,A),
  write_file(A,'(as0) := '),
  c_initial(C),
  write_s0_dom_1(T,N,C,A),
  nl_file(A).

write_s0_dom_1(_,N,[],A) :-
  write_file(A,'nothing'),
  write2_rrl(N,A).
write_s0_dom_1(T,N,[[Obj,T]|L],A) :-
  write_file(A,'c'),
  write2_rrl(N,A),
  write_file(A,'('),
  write_term_rrl(Obj,A),
  write_file(A,','),
  write_s0_dom_1(T,N,L,A),
  write_file(A,')').
write_s0_dom_1(T,N,[[_,Type]|L],A) :-
  \+ (T = Type),
  write_s0_dom_1(T,N,L,A).

write_uno([],_).
write_uno([not(equal(X,Y))|T],A) :-
  write_file(A,'('),
  write_term_rrl(X,A),
  write_file(A,' = '),
  write_term_rrl(Y,A),
  write_file(A,') := false'),
  nl_file(A),
  write_uno(T,A).

write_term_rrl(T,File):-
atomic(T),
variable(T),
text_append([x,T],NT),
writeq_file(File,NT).

write_term_rrl(T,File):-
atomic(T),
text_append([a,T],NT),
writeq_file(File,NT).

write_term_rrl(equal(T1,T2),File):-
write_file(File,'('),
write_term_rrl(T1,File),
write_file(File,' = '),
```

```
write_term_rrl(T2,File),
write_file(File,')').

write_term_rrl(T,File):-
T=..[N|A],
text_append([a,N],NN),
writeq_file(File,NN),
write_file(File,'('),
write_term_rrl_list(A,File),
write_file(File,')').

write_term_rrl_list([],_).
write_term_rrl_list([X],File):-
write_term_rrl(X,File).
write_term_rrl_list([X|Y],File):-
write_term_rrl(X,File),
write_file(File,','),
write_term_rrl_list(Y,File).
```

B.2 Otter and Prolog

Module: `proof.pl`

Description: Provides a simple interface between a user and SCDBR so as to allow integrity constraint proofs using resolution-based theorem provers (in this case Otter and Prolog).

```
prove_ic(X) :-
    write('Generating include file ...    '),
    ttyflush,
    generar_include([fluent,action,una,initial,uno],'temp/incl-otter'),
    write('Done.'), nl,
    c_subs_list([s],[s0],[X],[Y]),
    c_subs_list([s],[do(a,s)],[X],[Z]),
    write('Proof for the initial database:'),nl,
    write('------------------------------'),nl,
    initial(I),
    list_assert(I),
    sc2pr(Y,ppp,'temp/temp.pl'),
    ['temp/temp.pl'],
    say_result_s0,
    nl,
    write('Do you want to see the proof/refutation for'),
    nl,
    write('the initial state? (The prolog program that'),
```

```
    nl,
    write('will be traced is temp/temp.pl) (y./n.) '),
    read(Answer),
    show_proof(Answer),
    notrace,
    write('Inductive step:'),nl,
    write('--------------'),nl,
    otterize(all(s,all(a,implies(and(poss(a,s),X),Z))),
     ['set(binary_res).','set(hyper_res).','set(para_from).','set(para_into).'],
     ['temp/incl-otter'],'temp/proof-is','fg'),
    nl,
    write('The proof for the inductive step can be found in temp/proof-is'),
    nl,
    rm_file(['temp/incl-otter','temp/temp.pl']).

say_result_s0 :-
    ppp,
    write('The proof in the initial state was successful.').
say_result_s0 :-
    write('The proof in the initial state was unsuccessful.').

show_proof(y) :-
    trace,
    ppp.
show_proof(_).
```

Module:`initial_otter.pl`

Description: translates an IC instantiated according to Reiter's inductive schema $IP_{s \geq S_0}$ into the Otter syntax, using a predefined SCDBR module (`otterize`). Pleas refer to the SCDBR documentation for a description of the `otterize` module.

```
recorrer([H|T],X) :- closure(H,W),fotter_w(W,X),recorrer(T,X).
recorrer([],_).

generar_include(F,FILE):-
unix_mktemp('giXXXXXX',FILE),
write_file(FILE,'formula_list(usable).'),
        nl_file(FILE),
        write_file(FILE,'all x (x = x).'),
        nl_file(FILE),
        if(member(fluent,F),fluentes(FILE),true),
        if(member(action,F),acciones(FILE),true),
        if(member(una,F),unas(FILE),true),
        if(member(initial,F),iniciales(FILE),true),
if(member(uno,F),unos(FILE),true),
        write_file(FILE,'end_of_list.').
```

```
fluentes(FILE):-
predicates(Y),
all_ssa(Y,FILE).

all_ssa([],_).
all_ssa([X|Y],FILE):-
ssa(X,SSA),
closure(SSA,F),
fotter_w(F,FILE),
all_ssa(Y,FILE).

acciones(FILE):-
functors(X),
all_ap(X,FILE).

all_ap([],_).
all_ap([X|Y],FILE):-
ap(X,AP),
closure(AP,F),
fotter_w(F,FILE),
all_ap(Y,FILE).

unas(FILE):-
una(U),!,
recorrer(U,FILE).

iniciales(FILE):-
initial(T),
recorrer(T,FILE).

unos(FILE):-
c_initial(C),
remove_duplicates(C,L),
uno(L,U),!,
recorrer(U,FILE).

otter_initial(F) :-
write_now('Generating include file...'),
generar_include([initial,uno],FILE),
write('Done.'), nl,
write('Proof for the initial database:'),nl,
write('-------------------------------'),nl,
closure(F,CF),
otterize(CF,['set(binary_res).','set(hyper_res).','set(para_from).',
         'set(para_into).'],[FILE],'temp/proof-s0','fg'),
nl,
write('The proof can be found in temp/proof-s0'),
        nl,
rm_file([FILE]).
```

Module:`initial_prolog.pl`

Description: Proves an integrity constraint in the initial state using Prolog.

```
for2pr(F,P,A,V) :-
  v_quantify(F,V1),
  v_f(F,V2),
  diff(V2,V1,V),
  intr_neg(1,F,F1),
  dnf(F1,F2),
  order(F2,F3),
  write_or(V,0,F3,P,A).

list_assert([]).
list_assert([X|Y]):-
assert(X),
list_assert(Y).

prolog_initial(X):-
unix_mktemp('piXXXXXX',TMPFILE),
        nl,write_now('Transforming Query into Prolog Goal...'),
for2pr(X,pi_v1,TMPFILE,VT),
write('Done.'),nl,nl,
initial(I),
list_assert(I),
consult(TMPFILE),
read_file(TMPFILE,T1),
        rm_file([TMPFILE]),!,
        T1=..[_|[T|_]],
T=..[_|V],
setof(V,T,PR),
if(PR=[[]],write('true'),separate(PR,VT)).

separate(PR,VT):-
nl,
once(separate_1(PR,VT)).

separate_1([],_).
separate_1([L1|RL],LV):-
merge(LV,L1,R1),
        write(R1),
nl,
        separate_1(RL,LV).
```

Module:`pros04.pl`

Description: the important subroutine of module `initial-prolog`. It implements the Lloyd-Topor algorithm for translating a first order formula into logic programming.

```
% These are the two top-level predicates of
% this module. Predicate sc2pr takes as input
% a formula F that has no free variables and
% generates in file A a Prolog program that
% can check whether formula F is true or false.
% Typically, you should
%  - do something to instantiate the basic
% predicates of F, then
%   sc2pr(<formula>,<prefix>,<file>),
%   [<file>].
% Then, the subgoal <prefix> succeeds if formula
% F is true and fails otherwise. All subgoals in
% program A have <prefix> in the beginning of their
% names. In particular, the top-level goal of A,
% which is equivalent to F, is called <prefix>.
% intr_neg, dnf, order and write_or are the second-
% level goals. They are explained in what follows.

sc2pr(F,P,A) :-
  intr_neg(1,F,F1),
  dnf(F1,F2),
  order(F2,F3),
  write_or([],0,F3,P,A).

% This predicate, ssa2pr, is almost the same as sc2pr
% but with some differences. ssa2pr takes as input
% a successor state axiom, not a general formula,
% and returns in file A a program that computes
% the fluent in the successor state given all
% necessary fluents in the previous state.

ssa2pr(all(_,implies(_,iff(H,F))),P,A) :-
  v_f(H,L),
  intr_neg(1,F,F1),
  dnf(F1,F2),
  order(F2,F3),
  write_or(L,0,F3,P,A).

% intr_neg "introduces" recursively the negations
% into the second parameter, giving the third
% parameter. For example, intr_neg(1, not(forall(x,f)),F)
% gives F = (exists x not f), and introduces
% recursively the negation into f. The first
% argument in intr_neg is 1 if the current formula
% is not negated and 0 if it is negated.
% intr_neg also eliminates implications and
% equivalences, giving only conjunctions,
```

```
% disjunctions and bottom-level negations.

intr_neg(1,and(F1,F2),and(F3,F4)) :-
  intr_neg(1,F1,F3),
  intr_neg(1,F2,F4).
intr_neg(0,and(F1,F2),or(F3,F4)) :-
  intr_neg(0,F1,F3),
  intr_neg(0,F2,F4).
intr_neg(1,or(F1,F2),or(F3,F4)) :-
  intr_neg(1,F1,F3),
  intr_neg(1,F2,F4).
intr_neg(0,or(F1,F2),and(F3,F4)) :-
  intr_neg(0,F1,F3),
  intr_neg(0,F2,F4).
intr_neg(1,not(F),F1) :-
  intr_neg(0,F,F1).
intr_neg(0,not(F),F1) :-
  intr_neg(1,F,F1).
intr_neg(1,implies(F1,F2),or(F3,F4)) :-
  intr_neg(0,F1,F3),
  intr_neg(1,F2,F4).
intr_neg(0,implies(F1,F2),and(F3,F4)) :-
  intr_neg(1,F1,F3),
  intr_neg(0,F2,F4).
intr_neg(1,iff(F1,F2),or(and(F3,F4),and(F5,F6))) :-
  intr_neg(1,F1,F3),
  intr_neg(1,F2,F4),
  intr_neg(0,F1,F5),
  intr_neg(0,F2,F6).
intr_neg(0,iff(F1,F2),or(and(F3,F4),and(F5,F6))) :-
  intr_neg(1,F1,F3),
  intr_neg(0,F2,F4),
  intr_neg(0,F1,F5),
  intr_neg(1,F2,F6).
intr_neg(1,all(X,F),all(X,F1)) :-
  intr_neg(1,F,F1).
intr_neg(0,all(X,F),some(X,F1)) :-
  intr_neg(0,F,F1).
intr_neg(1,some(X,F),some(X,F1)) :-
  intr_neg(1,F,F1).
intr_neg(0,some(X,F),all(X,F1)) :-
  intr_neg(0,F,F1).
intr_neg(1,X,X).
intr_neg(0,X,not(X)).

% dnf is the second subgoal of sc2pr. Given
% a formula generated by intr_neg, dnf
% transforms it into disjunctive normal form.
% Subformulas existentially quantified are
% transformed into disjunctive normal form,
% but subformulas universally quantified are
```

```
% transformed into conjunctive normal form.
% Also, some other transformations are done
% on quantified subformulas which are explained
% later on.

dnf(or(F1,F2),F3) :-
  dnf(F1,F4),
  dnf(F2,F5),
  append_or(F4,F5,F3).
dnf(and(F1,F2),F3) :-
  dnf(F1,F4),
  dnf(F2,F5),
  mix_or(F4,F5,F3).
dnf(all(X,F),F1) :-
  trans(all(X,F),F2),
  dnf1(F2,F1).
dnf(some(X,F),F1) :-
  trans(some(X,F),F2),
  dnf1(F2,F1).
dnf(X,X).

% cnf is similar to dnf, but it transforms a formula
% into conjunctive normal form instead of disjunctive

cnf(and(F1,F2),F3) :-
  cnf(F1,F4),
  cnf(F2,F5),
  append_and(F4,F5,F3).
cnf(or(F1,F2),F3) :-
  cnf(F1,F4),
  cnf(F2,F5),
  mix_and(F4,F5,F3).
cnf(all(X,F),F1) :-
  trans(all(X,F),F2),
  cnf1(F2,F1).
cnf(some(X,F),F1) :-
  trans(some(X,F),F2),
  cnf1(F2,F1).
cnf(X,X).

% dnf1 and cnf1 do the same as dnf and cnf, but
% leave unchanged the quantified subformulas.
% This is because dnf1 and cnf1 receive as input
% formulas
% in which all the changes regarding the quantified
% subformulas are already done. Therefore, dnf1
% and cnf1 do not need to care about processing
% the quantified subformulas.

dnf1(or(F1,F2),F3) :-
  dnf1(F1,F4),
```

```
  dnf1(F2,F5),
  append_or(F4,F5,F3).
dnf1(and(F1,F2),F3) :-
  dnf1(F1,F4),
  dnf1(F2,F5),
  mix_or(F4,F5,F3).
dnf1(X,X).

cnf1(and(F1,F2),F3) :-
  cnf1(F1,F4),
  cnf1(F2,F5),
  append_and(F4,F5,F3).
cnf1(or(F1,F2),F3) :-
  cnf1(F1,F4),
  cnf1(F2,F5),
  mix_and(F4,F5,F3).
cnf1(X,X).

% append_or takes as input two lists of formulas
% related by or's and returns the disjunction of
% both lists. That is, it is an append operation.

append_or(or(F1,F2),F3,or(F1,F4)) :-
  append_or(F2,F3,F4).
append_or(X,F3,or(X,F3)).

% append_and is the same as append_or, but with and's.

append_and(and(F1,F2),F3,and(F1,F4)) :-
  append_and(F2,F3,F4).
append_and(X,F3,and(X,F3)).

% mix_or receives as input two formulas that are
% or-lists and return the conjunction of both,
% in a distributed way. For example,
% mix_or(or(f1,or(f2,f3)),or(f4,or(f5,f6)), F)
% returns
% F = or(and(f1,f4),or(and(f1,f5),or(and(f1,f6),
%     or(and(f2,f4),or(and(f2,f5),or(and(f2,f6),
%     or(and(f3,f4),or(and(f3,f5),and(f3,f6))))))))))))

mix_or(or(F1,F2),F3,F4) :-
  mix_or1(F1,F3,F5),
  mix_or(F2,F3,F6),
  append_or(F5,F6,F4).
mix_or(X,F3,F4) :-
  mix_or1(X,F3,F4).

mix_or1(F1,or(F2,F3),F4) :-
  mix_or1(F1,F3,F6),
  append_and(F1,F2,F5),
```

```
  append_or(F5,F6,F4).
mix_or1(F1,X,F4) :-
  append_and(F1,X,F4).

% mix_and is the same as mix_or, but returns
% the disjunction of two and-lists in a
% distributed way.

mix_and(and(F1,F2),F3,F4) :-
  mix_and1(F1,F3,F5),
  mix_and(F2,F3,F6),
  append_and(F5,F6,F4).
mix_and(X,F3,F4) :-
  mix_and1(X,F3,F4).

mix_and1(F1,and(F2,F3),F4) :-
  mix_and1(F1,F3,F5),
  append_or(F1,F2,F6),
  append_and(F5,F6,F4).
mix_and1(F1,X,F4) :-
  append_or(F1,X,F4).

% trans transforms the first formula giving the
% second. trans takes as input only quantified
% formulas and does the following processing:
% for universally (existentially) quantified
% formulas, trans transforms it into conjunctive
% (disjunctive) normal form. This is done by
% cnf (dnf). The resulting formula is therefore
% a universal (existential) quantification of
% conjunctions (disjunctions). So, intr_all
% (intr_some) transforms it into conjunctions
% (disjunctions) of universal (existential)
% quantifications. Finally, put_out_all
% (put_out_some) takes out from the quantified
% subformulas all the literals that do not
% depend on the quantified variable. For example,
% put_out_all(all(x,or(p(x,y),q(y,z))),F) gives
% F = or(all(x,p(x,y)),q(y,z)). That is, put_out_all
% took off q(y,z) because the quantified variable, x,
% does not appear in it.

trans(all(X,F),F1) :-
  cnf(F,F2),
  intr_all(X,F2,F3),
  put_out_all(F3,F1).
trans(some(X,F),F1) :-
  dnf(F,F2),
  intr_some(X,F2,F3),
  put_out_some(F3,F1).
```

```
intr_all(X,and(Y,Z),and(all(X,Y),W)) :-
  intr_all(X,Z,W).
intr_all(X,Y,all(X,Y)).

intr_some(X,or(Y,Z),or(some(X,Y),W)) :-
  intr_some(X,Z,W).
intr_some(X,Y,some(X,Y)).

put_out_all(and(X,Y),and(Z,W)) :-
  put_out_all1(X,Z),
  put_out_all(Y,W).
put_out_all(X,Y) :-
  put_out_all1(X,Y).

put_out_some(or(X,Y),or(Z,W)) :-
  put_out_some1(X,Z),
  put_out_some(Y,W).
put_out_some(X,Y) :-
  put_out_some1(X,Y).

put_out_all1(all(X,Y),Z) :-
  sep_all(X,Y,W,U),
  union_all(X,W,U,Z).

put_out_some1(some(X,Y),Z) :-
  sep_some(X,Y,W,U),
  union_some(X,W,U,Z).

% sep_all (sep_some) takes as input a disjunction
% (conjunction) of quantified formulas or literals
% (no conjunctions (disjunctions)) and returns
% two lists: the second (fourth argument) with all quantified formulas
% and with the literals that depend on the variable (the first
% parameter). The first list (third argument) contains all the literals
% that do not depend on the variable, that is, those
% that will be taken off from the quantification

sep_all(Y,or(all(X,F1),F2),L1,[all(X,F1)|L2]) :-
  sep_all(Y,F2,L1,L2).
sep_all(Y,or(some(X,F1),F2),L1,[some(X,F1)|L2]) :-
  sep_all(Y,F2,L1,L2).
sep_all(Y,or(X,F2),L1,[X|L2]) :-
  v_f(X,L),
  member(Y,L),
  sep_all(Y,F2,L1,L2).
sep_all(Y,or(X,F2),[X|L1],L2) :-
  sep_all(Y,F2,L1,L2).
sep_all(_,all(X,F1),[],[all(X,F1)]).
sep_all(_,some(X,F1),[],[some(X,F1)]).
sep_all(Y,X,[],[X]) :-
  v_f(X,L),
```

```
  member(Y,L).
sep_all(_,X,[X],[]).

sep_some(Y,and(all(X,F1),F2),L1,[all(X,F1)|L2]) :-
  sep_some(Y,F2,L1,L2).
sep_some(Y,and(some(X,F1),F2),L1,[some(X,F1)|L2]) :-
  sep_some(Y,F2,L1,L2).
sep_some(Y,and(X,F2),L1,[X|L2]) :-
  v_f(X,L),
  member(Y,L),
  sep_some(Y,F2,L1,L2).
sep_some(Y,and(X,F2),[X|L1],L2) :-
  sep_some(Y,F2,L1,L2).
sep_some(_,all(X,F1),[],[all(X,F1)]).
sep_some(_,some(X,F1),[],[some(X,F1)]).
sep_some(Y,X,[],[X]) :-
  v_f(X,L),
  member(Y,L).
sep_some(_,X,[X],[]).

% union_all (union_some) takes as input the two output lists
% from sep_all (sep_some), which are the second and the third
% arguments and concatenates them back into a formula, the
% fourth argument. Note that union_all must check the emptyness
% of both input arguments, because each possible case gives
% different results. The first argument is the quantified
% variable.

union_all(X,[],[H|T],all(X,Z)) :-
  union_all_1([H|T],Z).
union_all(_,[H|T],[],Z) :-
  union_all_1([H|T],Z).
union_all(X,[H|T],[H1|T1],or(Y,all(X,Z))) :-
  union_all_1([H|T],Y),
  union_all_1([H1|T1],Z).

union_all_1([H],H).
union_all_1([H1|[H2|T]],or(H1,X)) :-
  union_all_1([H2|T],X).

union_some(X,[],[H|T],some(X,Z)) :-
  union_some_1([H|T],Z).
union_some(_,[H|T],[],Z) :-
  union_some_1([H|T],Z).
union_some(X,[H|T],[H1|T1],and(Y,some(X,Z))) :-
  union_some_1([H|T],Y),
  union_some_1([H1|T1],Z).

union_some_1([H],H).
union_some_1([H1|[H2|T]],and(H1,X)) :-
  union_some_1([H2|T],X).
```

```
% This is the fourth top-level goal of the whole
% program. The third argument is an input formula
% that will be transformed into a prolog program
% that will be stored in the file whose name is the
% fourth argument. Notice that because of all the
% pre-processing that has been done on the formula,
% its translation into a prolog program is
% straightforward. An outline of the translation is
% the following: first, notice that as a result of
% the pre-processing, the formula is in disjunctive
% normal form, except for the fact that some of
% the "literals" can be quantified formulas. Notice
% that all universally (existentially) quantified
% formulas are disjunctions (conjunctions) of
% literals or quantified formulas. Therefore, the
% translation goes as follows: suppose that the
% top-level formula is for example
%
%       (F1 and F2) or (F3 and F4 and F5)
%
% Then the following program is generated:
%
%   ppp :- p-0.
%   ppp :- p-1.
%   ppp-0 :- F1, F2.
%   ppp-1 :- F3, F4, F5.
%
% (assuming that the prefix given in the third argument
% is ppp). If a quantified subformula appears it is
% replaced by ppp-<number>(<vars>). For example,
% suppose that F1 above is (exists x) p(x). Then,
% subgoal ppp-0 would look like:
%
%   ppp-0 :- ppp-0-0, F2.
%
% and the following clause is added:
%
%   ppp-0-0 :- p(x).
%
% and this process goes on recursively for all quantified
% subformulas. If a universally quantified subformula
% appears, if is transformed into the negation of an
% existentially quantified formula and the process
% goes on recursively for the existentially quantified
% formula.

% V is the list of the variables that the formula depends
% on. When write_or is called from sc2pr, V is the empty
% list because the formula is closed. When write_or is called
% from ssa2pr, V contains all the variables that the fluent
```

```
% depends on, including the state and the action. M is the
% correlative number of the clauses that define the top-level
% predicate.

write_or(V,M,or(F1,F2),P,A) :-
  write_and(V,M,F1,P,A),
  M1 is M+1,
  write_or(V,M1,F2,P,A).
write_or(V,M,F,P,A) :-
  write_and(V,M,F,P,A).

% write_and writes one clause of the program

write_and(V,M,F,P,A) :-
  write_file(A,P),
  write_2(V,A),
  write_file(A,':-'),
  nl_file(A),
  write_3(V,M,0,F,P,A),
  write_some([M],0,V,F,P,A).

write_some(N,M,V,and(F1,F2),P,A) :-
  write_some_1(N,M,V,F1,P,A),
  M1 is M+1,
  write_some(N,M1,V,F2,P,A).
write_some(N,M,V,F,P,A) :-
  write_some_1(N,M,V,F,P,A).

write_some_1(N,M,V,some(X,F),P,A) :-
  write_file(A,P),
  write_file(A,'_'),
  write_file(A,M),
  write_1(N,A),
  write_2(V,A),
  write_file(A,':-'),
  nl_file(A),
  write_some_2([M|N],0,[X|V],F,P,A),
  write_some([M|N],0,[X|V],F,P,A).
write_some_1(N,M,V,all(X,F),P,A) :-
  write_file(A,P),
  write_file(A,'_'),
  write_file(A,M),
  write_1(N,A),
  write_2(V,A),
  write_file(A,':-'),
  nl_file(A),
  write_all_2([M|N],0,[X|V],F,P,A),
  write_all([M|N],0,[X|V],F,P,A).
write_some_1(_,_,_,_,_,_).

write_some_2(N,M,V,and(F1,F2),P,A) :-
```

```
  write_some_3(N,M,V,F1,P,A),
  M1 is M+1,
  write_some_4(N,M1,V,F2,P,A),
  write_file(A,'.'),
  nl_file(A).
write_some_2(N,M,V,F,P,A) :-
  write_some_3(N,M,V,F,P,A),
  write_file(A,'.'),
  nl_file(A).

write_some_4(N,M,V,and(F1,F2),P,A) :-
  write_file(A,','),
  nl_file(A),
  write_some_3(N,M,V,F1,P,A),
  M1 is M+1,
  write_some_4(N,M1,V,F2,P,A).
write_some_4(N,M,V,F,P,A) :-
  write_file(A,','),
  nl_file(A),
  write_some_3(N,M,V,F,P,A).

write_some_3(N,M,V,all(_,_),P,A) :-
  write_file(A,'\\+ '),
  write_file(A,P),
  write_file(A,'_'),
  write_file(A,M),
  write_1(N,A),
  write_2(V,A).
write_some_3(N,M,V,some(_,_),P,A) :-
  write_file(A,P),
  write_file(A,'_'),
  write_file(A,M),
  write_1(N,A),
  write_2(V,A).
write_some_3(N,M,_,equal(X,Y),_,A) :-
  write_prolog_eq(equal(X,Y),[M|N],A,pos).
write_some_3(N,M,_,not(equal(X,Y)),_,A) :-
  write_prolog_eq(equal(X,Y),[M|N],A,neg).
write_some_3(_,_,_,not(F),_,A) :-
  write_file(A,'\\+ '),
  change_case_fluent(F,G),
  write_file_q(A,F,G).
write_some_3(_,_,_,F,_,A) :-
  change_case_fluent(F,G),
  write_file_q(A,F,G).

order_some(F1,F2) :-
  separate_some(F1,L1,L2),
  append(L1,L2,L3),
  gen_some(L3,F2).
```

```
separate_some(and(some(X,Y),F),[some(X,Y1)|L1],L2) :-
  order_some(Y,Y1),
  separate_some(F,L1,L2).
separate_some(and(all(X,Y),F),L1,[all(X,Y1)|L2]) :-
  order_all(Y,Y1),
  separate_some(F,L1,L2).
separate_some(and(not(F1),F2),L1,[not(F1)|L2]) :-
  separate_some(F2,L1,L2).

separate_some(and(equal(X,Y),F),L1,[equal(X,Y)|L2]) :-
  \+ is_act(X,Y),
  separate_some(F,L1,L2).
separate_some(and(equal(X,Y),F),[equal(X,Y)|L1],L2) :-
  is_act(X,Y),
  separate_some(F,L1,L2).
separate_some(and(>(X,Y),F),L1,[>(X,Y)|L2]) :-
  separate_some(F,L1,L2).
separate_some(and(<(X,Y),F),L1,[<(X,Y)|L2]) :-
  separate_some(F,L1,L2).
separate_some(and(>=(X,Y),F),L1,[>=(X,Y)|L2]) :-
  separate_some(F,L1,L2).
separate_some(and(=<(X,Y),F),L1,[=<(X,Y)|L2]) :-
  separate_some(F,L1,L2).
separate_some(and(F1,F2),[F1|L1],L2) :-
  separate_some(F2,L1,L2).
separate_some(some(X,Y),[some(X,Y1)],[]) :-
  order_some(Y,Y1).
separate_some(all(X,Y),[],[all(X,Y1)]) :-
  order_all(Y,Y1).
separate_some(not(F1),[],[not(F1)]).
separate_some(equal(X,Y),[],[equal(X,Y)]) :-
  \+ is_act(X,Y).
separate_some(equal(X,Y),[equal(X,Y)],[]) :-
  is_act(X,Y).
separate_some(<(X,Y),[],[<(X,Y)]).
separate_some(>(X,Y),[],[>(X,Y)]).
separate_some(=<(X,Y),[],[=<(X,Y)]).
separate_some(>=(X,Y),[],[>=(X,Y)]).
separate_some(F,[F],[]).

gen_some([H],H).
gen_some([H|[H1|T]],and(H,F)) :-
  gen_some([H1|T],F).

write_all(N,M,V,or(F1,F2),P,A) :-
  write_all_1(N,M,V,F1,P,A),
  M1 is M+1,
  write_all(N,M1,V,F2,P,A).
write_all(N,M,V,F,P,A) :-
  write_all_1(N,M,V,F,P,A).
```

```
write_all_1(N,M,V,some(X,F),P,A) :-
  write_file(A,P),
  write_file(A,'_'),
  write_file(A,M),
  write_1(N,A),
  write_2(V,A),
  write_file(A,':-'),
  nl_file(A),
  write_some_2([M|N],0,[X|V],F,P,A),
  write_some([M|N],0,[X|V],F,P,A).
write_all_1(N,M,V,all(X,F),P,A) :-
  write_file(A,P),
  write_file(A,'_'),
  write_file(A,M),
  write_1(N,A),
  write_2(V,A),
  write_file(A,':-'),
  nl_file(A),
  write_all_2([M|N],0,[X|V],F,P,A),
  write_all([M|N],0,[X|V],F,P,A).
write_all_1(_,_,_,_,_,_).

write_all_2(N,M,V,or(F1,F2),P,A) :-
  write_all_3(N,M,V,F1,P,A),
  M1 is M+1,
  write_all_4(N,M1,V,F2,P,A),
  write_file(A,'.'),
  nl_file(A).
write_all_2(N,M,V,F,P,A) :-
  write_all_3(N,M,V,F,P,A),
  write_file(A,'.'),
  nl_file(A).

write_all_4(N,M,V,or(F1,F2),P,A) :-
  write_file(A,','),
  nl_file(A),
  write_all_3(N,M,V,F1,P,A),
  M1 is M+1,
  write_all_4(N,M1,V,F2,P,A).
write_all_4(N,M,V,F,P,A) :-
  write_file(A,','),
  nl_file(A),
  write_all_3(N,M,V,F,P,A).


write_all_3(N,M,V,all(_,_),P,A) :-
  write_file(A,P),
  write_file(A,'_'),
  write_file(A,M),
  write_1(N,A),
  write_2(V,A).
```

```
write_all_3(N,M,V,some(_,_),P,A) :-
  write_file(A,'\\+ '),
  write_file(A,P),
  write_file(A,'_'),
  write_file(A,M),
  write_1(N,A),
  write_2(V,A).
write_all_3(N,M,_,equal(X,Y),_,A) :-
  write_prolog_eq(equal(X,Y),[M|N],A,neg).
write_all_3(N,M,_,not(equal(X,Y)),_,A) :-
  write_prolog_eq(equal(X,Y),[M|N],A,pos).
write_all_3(_,_,_,not(F),_,A) :-
  change_case_fluent(F,G),
  write_file_q(A,F,G).
write_all_3(_,_,_,F,_,A) :-
  write_file(A,'\\+ '),
  change_case_fluent(F,G),
  write_file_q(A,F,G).

order_all(F1,F2) :-
  separate_all(F1,L1,L2),
  append(L1,L2,L3),
  gen_all(L3,F2).

separate_all(or(all(X,Y),F),[all(X,Y1)|L1],L2) :-
  order_all(Y,Y1),
  separate_all(F,L1,L2).
separate_all(or(some(X,Y),F),L1,[some(X,Y1)|L2]) :-
  order_some(Y,Y1),
  separate_all(F,L1,L2).
separate_all(or(not(equal(X,Y)),F2),L1,[not(equal(X,Y))|L2]) :-
  \+ is_act(X,Y),
  separate_all(F2,L1,L2).
separate_all(or(not(equal(X,Y)),F2),[not(equal(X,Y))|L1],L2) :-
  is_act(X,Y),
  separate_all(F2,L1,L2).
separate_all(or(not(>(X,Y)),F2),L1,[not(>(X,Y))|L2]) :-
  separate_all(F2,L1,L2).
separate_all(or(not(<(X,Y)),F2),L1,[not(<(X,Y))|L2]) :-
  separate_all(F2,L1,L2).
separate_all(or(not(>=(X,Y)),F2),L1,[not(>=(X,Y))|L2]) :-
  separate_all(F2,L1,L2).
separate_all(or(not(=<(X,Y)),F2),L1,[not(=<(X,Y))|L2]) :-
  separate_all(F2,L1,L2).
separate_all(or(not(F1),F2),[not(F1)|L1],L2) :-
  separate_all(F2,L1,L2).
separate_all(or(F1,F2),L1,[F1|L2]) :-
  separate_all(F2,L1,L2).
separate_all(all(X,Y),[all(X,Y1)],[]) :-
  order_all(Y,Y1).
separate_all(some(X,Y),[],[some(X,Y1)]) :-
```

```
  order_some(Y,Y1).
separate_all(not(equal(X,Y)),[],[not(equal(X,Y))]) :-
  \+ is_act(X,Y).
separate_all(not(equal(X,Y)),[not(equal(X,Y))],[]) :-
  is_act(X,Y).
separate_all(not(<(X,Y)),[],[not(<(X,Y))]).
separate_all(not(>(X,Y)),[],[not(>(X,Y))]).
separate_all(not(=<(X,Y)),[],[not(=<(X,Y))]).
separate_all(not(>=(X,Y)),[],[not(>=(X,Y))]).
separate_all(not(F1),[not(F1)],[]).
separate_all(F,[],[F]).

gen_all([H],H).
gen_all([H|[H1|T]],or(H,F)) :-
  gen_all([H1|T],F).

% write_1 writes all the numbers in the list separated
% by "_". The list of numbers is the identification
% of the current sub-formula. It represents the position
% of the subformula within the top-level formula considered
% as a tree.

write_1([],_).
write_1([H|T],A) :-
  write_file(A,'_'),
  write_file(A,H),
  write_1(T,A).

% write_2 writes all the variables in the list, which are
% the variables that the current subgoal depends on.
% write_6 and write_5 are subgoals of write_2.

write_2([],_).
write_2([H|T],A) :-
  write_file(A,'('),
  write_5(H,A),
  write_6(T,A),
  write_file(A,')').

write_6([],_).
write_6([H|T],A) :-
  write_file(A,','),
  write_5(H,A),
  write_6(T,A).

write_5(X,A) :-
  change_case(X,Y),
  write_file(A,Y).

% write_3 writes the body of the clause. It is a loop
% that writes literals.
```

```
write_3(V,M,N,and(F1,F2),P,A) :-
  write_7(V,M,N,F1,P,A),
  N1 is N+1,
  write_8(V,M,N1,F2,P,A),
  write_file(A,'.'),
  nl_file(A).
write_3(V,M,N,F,P,A) :-
  write_7(V,M,N,F,P,A),
  write_file(A,'.'),
  nl_file(A).

% write_8 is a subgoal of write_3.

write_8(V,M,N,and(F1,F2),P,A) :-
  write_file(A,','),
  nl_file(A),
  write_7(V,M,N,F1,P,A),
  N1 is N+1,
  write_8(V,M,N1,F2,P,A).
write_8(V,M,N,F,P,A) :-
  write_file(A,','),
  nl_file(A),
  write_7(V,M,N,F,P,A).

% write_7 is another subgoal of write_3. It writes
% one literal. When the literal is a "normal" literal
% write_7 writes is as expected, depending on whether
% it is negated, it is an equality or a normal predicate.
% When the "literal" is a quantified subformula,
% write_7 replaces it by a new symbol, that will be
% <prefix><id>(<vars>). The prefix is the fifth parameter,
% P, and is the prefix that all subgoals have. <id>
% is the list of numbers that identifies the current
% subformula, and is its position in the top-goal formula.
% It is the second and third parameters, M and N.
% <vars> is the first parameter, V, the list of variables
% that the predicate depends on. This new predicate,
% <prefix><id>(<vars>), will be defined later, by
% predicate write_some.

write_7(V,M,N,some(_,_),P,A) :-
  write_file(A,P),
  write_file(A,'_'),
  write_file(A,N),
  write_file(A,'_'),
  write_file(A,M),
  write_2(V,A).
write_7(V,M,N,all(_,_),P,A) :-
  write_file(A,'\\+ '),
  write_file(A,P),
```

```
  write_file(A,'_'),
  write_file(A,N),
  write_file(A,'_'),
  write_file(A,M),
  write_2(V,A).
write_7(_,_,_,not(equal(T1,T2)),_,A) :-
  change_case_general(T1,T3),
  change_case_general(T2,T4),
  write_file_q(A,T1,T3),
  write_file(A,' \\== '),
  write_file_q(A,T2,T4).
write_7(_,_,_,equal(T1,T2),_,A) :-
  change_case_general(T1,T3),
  change_case_general(T2,T4),
  write_file_q(A,T1,T3),
  write_file(A,' = '),
  write_file_q(A,T2,T4).
write_7(_,_,_,not(F),_,A) :-
  write_file(A,'\\+ '),
  change_case_fluent(F,F1),
  write_file_q(A,F,F1).
write_7(_,_,_,F,_,A) :-
  change_case_fluent(F,F1),
  write_file_q(A,F,F1).

% This is the third top-level subgoal. It is the
% last change that is done on the formula. Its
% purpose is to order the clauses, including
% the clauses that appear on the quantified
% subformulas, putting first the positive literals,
% including the literals that represent quantified
% sub formulas and last the negative literals
% and/or equalities. The objective is to have
% a bigger chance for the top-level goal not
% to produce run-time errors due to non-instantiated
% variables.

order(or(F1,F2),or(F3,F4)) :-
  order_some(F1,F3),
  order(F2,F4).
order(F,F2) :-
  order_some(F,F2).


is_act(X,_):-
atomic(X),
variable(X),
type_variable(X,action).

is_act(_,Y):-
atomic(Y),
```

```
variable(Y),
type_variable(Y,action).
```