Specifying Active Rules for Database Maintenance*

Leopoldo Bertossi and Javier Pinto
Departamento de Ciencia de la Computación
Escuela de Ingeniería
Pontificia Universidad Católica de Chile
Casilla 306, Santiago, Chile
{bertossi, jpinto}@ing.puc.cl.

Abstract

In this article, we extend previous work on the development of logical foundations for the specification of the dynamics of databases. In particular, we deal with the problems of derivation of *active rules* that maintain the consistency of the database by triggering repairing actions, and the correct integration of the specification of the rules into the original specification of the database dynamics. In particular, we show that the expected are achieved, and that a part of the specification takes care, at the same object level, of enforcing the execution of the repairing actions when necessary.

1 Introduction

In this article we propose a logic based approach for the automatic derivation of active rules [36] for the maintenance of the integrity of a database [7]. This research follows the tradition of specifying the semantics of a database using mathematical logic [22, 9]. In particular, we deal with a logical framework in which transactions are treated as objects within the logical language, allowing one to reason about the dynamics of change in a database as transactions are executed.

As shown in [32], it is possible to specify the dynamics of a relational database with a special formalism written in the situation calculus (SC) [18], a language of many-sorted predicate logic for representing knowledge and reasoning about actions and change. Apart from providing a natural and well studied semantics, the formalism can be used to solve different reasoning tasks. For instance, reason about the evolution of a database [4],

*This research has been partially financed by FONDECYT (Grants 1990089 and 1980945), and ECOS/CONICYT (Grant C97E05).

reason about the hypothetical evolutions of a database [1], reason about the dynamics of views [2], etc.

In the SC formalism, each relational table is represented by a fluent predicate (a predicate that has one argument that represents the state of the database, i.e., a situation). In order to specify the dynamics of the relations in a database, one derives the so called *successor state axioms* (SSAs); one SSA per base relation or table.

Each SSA describes the exact conditions under which the presence of an arbitrary tuple in the table is true after executing an arbitrary legal primitive transaction[1]. A SSA states necessary and sufficient conditions for a tuple to belong to a relation after a transaction is performed. These conditions refer only to the state[2] in which the transaction is executed and does not make reference to further constraints on the resulting state. Thus, there are no explicit integrity constraints (IC) in the specification.

Given the exhaustive descriptions of the successor states (those obtained after executing primitive transactions) provided by the the SSAs, it is very easy to get into inconsistencies if integrity constraints are introduced in the specification, this is specially true for *ramification constraints*, that is, constraints that force the database to make indirect changes in tables due to changes in other tables of the database.

There are several options to deal with this problem:

1. One can assume that the ICs are somehow embedded in the SSAs. This is the approach in [32]. They should be logical consequences of the DB specification. In [5] a mechanism for the automated proof of these ICs is provided.

2. Some ICs can be considered as *qualification constraints*, that is, they are considered as constraints on the executability of the database actions. In [14] a methodology for translating these constraints into axioms on the executability of actions, or better, on the legality of actions is presented. In this approach, the so-called *Action Precondition Axioms* are generated.

3. For some interesting syntactical classes of ramification ICs, there are mechanisms for compiling them into *Effect Axioms*, from which the SSAs can be recomputed. Then, the explicit ICs disappear and they turn out to be logical consequences of the new specification [23, 20, 29] (see also [4] for implementation issues).

4. It is also possible to think of a database maintenance approach, consisting of adding active rules to a modification of the original specification. These rules enforce the satisfaction of the ICs by triggering appropriate auxiliary actions. Preliminary work on this, in a general framework for knowledge representation of action and change, is shown in [26, 25].

The last alternative is the subject of this paper. There are several issues to be considered. First, a computational mechanism should be provided for deriving active rules and repairing actions from the ICs. Second, the active rules should be consistent with the rest of the

[1]These are the simplest, non-decomposable transactions; they are domain dependent. In the KR literature they are called "actions". We consider the notions *primitive transaction* and *action* as synonyms.

[2]In this paper we do not make any distinction between situations and states.

specification and produce the expected effects. Third, since the active rules will have the usual *Event-Condition-Action* (ECA) form [36, 37], that does not have a direct predicate logic semantics, they should be specifiable in (a suitable extension of) the language of the situation calculus, and integrated smoothly with the rest of the specification. Some work in this direction, on the assumption that general ECA rules are given, is presented in [6].

2 Specification of the Database Dynamics

Characteristic ingredients of a particular language $\mathcal{L}$ of the situation calculus, besides the usual symbols of predicate logic, are: (a) Among others, the sorts *action*, *situation*. (b) Predicate symbols whose last argument is of the sort *situation*. These predicates depend on the state of the world and can be thought of as the tables in a relational database. (c) Operation symbols which applied to individuals produce actions (or primitive transactions), for example, $enroll(\cdot)$ may be an operation, and $enroll(john)$ becomes an action term. (d) A constant, S_0, to denote the initial state of the database. (e) An operation symbol *do* that takes an action and a situation as arguments, producing a new situation, a successor situation resulting from the execution of the action at the given situation.

In these languages there are first order variables for individuals of each sort, so it is possible to quantify over individuals, actions, and situations. They are usually denoted by $\forall \bar{x}$, $\forall a$, $\forall s$, respectively.

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

In Reiter's formalism, the knowledge contained in items (1) and (2) above is considered the initial database Σ_0. The information given in item (3) is formalized by means of action precondition axioms (*APA*s) of the form $Poss(A(\bar{x}), s) \equiv \pi_A(\bar{x}, s)$, for each action name A, where $\pi_A(\bar{x}, s)$ is a SC formula that is *simple in s*, that is, it contains no state term other than s (e.g., no *do* symbol); no quantifications on states; and no occurrences of the *Poss* predicate [15]. Finally, item (4) is expressed by effect axioms for pairs (primitive transaction, table):

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

$$\forall(\bar{x}, \bar{y}, s)[Poss(A(\bar{y}), s) \wedge \varphi_R^+(\bar{y}, \bar{x}, s) \supset R(\bar{x}, do(A(\bar{y}), s))]. \tag{1}$$

Intuitively, if the named primitive transaction A is possible, and the preconditions on the database, in particular, on the table R, represented by the meta-formula $\varphi_R^+(\bar{y}, \bar{x}, s)$ are true at state s, then the statement R becomes true of $\bar{x}$ at the successor state $do(A(\bar{y}), s)$ obtained after execution of A at state s. Here, $\bar{x}, \bar{y}$ are parameters for the table and action. Notice that in general we have two kinds of conditions: (a) preconditions for action executions, independently from any table they might affect; they are axiomatized by the *Poss* predicate; and (b) preconditions on the database for pairs table/action which make the changes possible (given that the action is already possible). These preconditions are represented by $\varphi_R^+(\bar{y}, \bar{x}, s)$.

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

$$\forall(\bar{x}, \bar{y}, s)[Poss(A(\bar{y}), s) \wedge \varphi_R^-(\bar{y}, \bar{x}, s) \supset \neg R(\bar{x}, do(A(\bar{y}), s))]. \tag{2}$$

This is the case where action A makes table R to become false of $\bar{x}$ in the successor state.

Example 1. Consider an educational database as in [32], with the following ingredients. Tables: 1. $Enrolled(stu, c, s)$, student stu is enrolled in course c in the state s. 2. $Grade(stu, c, g, s)$, the grade of student stu in course c is g in the state s. Primitive Transactions: 1. $register(stu, c)$, register student stu in course c. 2. $change(stu, c, g)$, change the grade of student stu in course c to g. 3. $drop(stu, c)$, eliminate student stu from the course c.

Action Precondition Axioms:

$$\forall(stu, c, s)[Poss(register(stu, c), s) \equiv \neg Enrolled(stu, c, s)].$$
$$\forall(stu, c, g, s)[Poss(change(stu, c, g), s) \equiv \exists g' \, Grade(stu, c, g', s)].$$
$$\forall(stu, c, s)[Poss(drop(stu, c), s) \equiv Enrolled(stu, c, s)].$$

Effect Axioms:

$$\forall(stu, c, s)[Poss(register(stu, c), s) \supset Enrolled(stu, c, do(register(stu, c), s))]$$
$$\forall(stu, c, s)[Poss(drop(stu, c), s) \supset \neg Enrolled(stu, c, do(drop(stu, c), s))]$$
$$\forall(stu, c, g, s)[Poss(change(stu, c, g), s) \supset Grade(stu, c, g, do(change(stu, c, g), s))]$$

$$\forall(stu, c, g, g', s)[Poss(change(stu, c, g'), s) \wedge g \neq g' \supset \neg Grade(stu, c, g, do(change(stu, c, g'), s))]$$

□

A problem with a specification like the one we have so far is that it does not mention the usually many things (entries in tables) that do not change when a specific action is executed. We face the so called *frame problem*, consisting of providing a short, succinct, specification of the properties that persist after actions are performed. Reiter [31] discovered a simple solution to the frame problem as it appears in the situation calculus. It allows to construct a first order specification, that accounts both for effects and non–effects, from a specification that contains descriptions of effects only, as in the example above. We sketch this solution in the rest of this section.

For illustration, assume that we have no negative effects, and two positive effect laws for table R: (1) and

$$\forall(\bar{x}, \bar{z}, s)[Poss(A'(\bar{z}), s) \land \psi_R^+(\bar{z}, \bar{x}, s) \supset R(\bar{x}, do(A'(\bar{z}), s))]. \tag{3}$$

We may combine them into one general positive effect axiom for table R:

$$\forall(a, \bar{x}, s)[Poss(a, s) \land [\exists\bar{y}(a = A(\bar{y}) \land \varphi_R^+(\bar{y}, \bar{x}, s)) \lor \exists\bar{z}(a = A'(\bar{z}) \land \psi_R^+(\bar{z}, \bar{x}, s))] \supset R(\bar{x}, do(a, s))].$$

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

$$\forall(a, \bar{x}, s)[Poss(a, s) \land \gamma_R^+(a, \bar{x}, s) \supset R(\bar{x}, do(a, s))].$$

Analogously, we obtain, for each table R, a general negative effect axiom:

$$\forall(a, \bar{x}, s)[Poss(a, s) \land \gamma_R^-(a, \bar{x}, s) \supset \neg R(\bar{x}, do(a, s))].$$

In this way we have represented, for each table R, in one single axiom all the actions and the corresponding conditions on the database that can make $R(\bar{x})$ to be true at an arbitrary successor state obtained by executing a legal action. In the same way we can describe when $R(\bar{x})$ becomes false.

Example 2. (cont'd) In the educational example we obtain the following general effect axioms for the table *Grade*:

$$\forall(a, stu, c, g, s)[Poss(a, s) \land a = change(stu, c, g) \supset Grade(stu, c, g, do(a, s))]$$
$$\forall(a, stu, c, g, s)[Poss(a, s) \land \exists g'(a = change(stu, c, g') \land g \neq g') \supset \neg Grade(stu, c, g, do(a, s))].$$

$\square$

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

$$\forall(a, s)Poss(a, s) \supset \forall\bar{x}[R(\bar{x}, do(a, s)) \equiv (\gamma_R^+(a, \bar{x}, s) \lor (R(\bar{x}, s) \land \neg\gamma_R^-(a, \bar{x}, s)))]. \tag{4}$$

Here, γ^+ and γ^- are of the form $\bigvee_{some\ A's} \exists\bar{u}(a = A(\bar{u}) \land \varphi(\bar{u}, \bar{x}, s))$, meaning that action A, under condition φ, makes $R(\bar{x}, do(A, s))$ true, in the case of γ^+, and false, in the case of γ^-. Thus, the SSA says that if action a is possible, then R becomes true at the successor state that results from the execution of action a if and only if a is one of the actions causing R to be true (and for which the corresponding preconditions, φ, are true), or R was already true before executing a and this action is not one of the actions that falsify R.

Example 3. (cont'd) In our running example, we obtain the following SSAs for the tables in the database:

$$\forall(a, s)Poss(a, s) \supset \forall(stu, c)[Enrolled(stu, c, do(a, s)) \equiv a = register(stu, c) \lor Enrolled(stu, c, s) \land a \neq drop(stu, c)]$$
$$\forall(a, s)Poss(a, s) \supset \forall(stu, c, g)[Grade(stu, c, g, do(a, s)) \equiv a = change(stu, c, g) \lor Grade(stu, c, g, s) \land \neg\exists g'(a = change(stu, c, g') \land g' \neq g)].$$

$\square$

Notice that, provided there is complete knowledge about the contents of the tables at the initial state, the SSAs completely describe the contents of the tables at every state that can be reached by executing a finite sequence of legal primitive transactions (that is for which the corresponding *Poss* conditions are satisfied). The SSAs have a nice inductive structure that makes some reasoning tasks easy, at least in principle.

In order for the specification to have the right logical consequences, we will assume that the following *Foundational Axioms of the Situation Calculus* (*FA*s) underlie any database specification [14]:

1. Unique Names Axioms for Actions ($UNAA$): $A_i(\bar{x}) \neq A_j(\bar{y})$, for all different action names A_i, A_j; and $\forall(\bar{x}, \bar{y})[A(\bar{x}) = A(\bar{y}) \supset \bar{x} = \bar{y}]$, for every action name A.
2. Unique Names Axioms for States: $S_0 \neq do(a, s)$, $do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2$.
3. For some reasoning tasks we need an Induction Axiom on States:
$$\forall P\ [P(S_0) \wedge \forall s \forall a\ (P(s) \supset P(do(a, s))) \quad \supset \quad \forall s\ P(s)],$$
that has the effect of restricting the domain of situations to the one containing the initial situation and the situations that can be obtained by executing a finite number of actions. In this way, no *non-standard* situations may appear. The axiom is second order, but for some reasoning tasks, like proving integrity constraints, reasoning can be done at the first order level [14, 4].
4. Finally, we will be usually interested in reasoning about states that are accessible from the initial situation by executing a finite sequence of legal actions. This *accessibility relation* on states, $\leq$, can be defined from the induction axiom plus the conditions:
$$\begin{array}{l} \neg s < S_0 \\ s < do(a, s') \equiv Poss(a, s') \wedge s \leq s'. \end{array}$$

Summarizing, a specification Σ, in the SC, of transaction based database updates consists of the sets: $\Sigma_0 \cup APAs \cup SSAs \cup FAs$.

Example 4. (cont'd) A static IC we would like to see satisfied at every accessible state of the database is the functional dependency for table *Grade*[3]. The IC can be expressed by:

$$\forall s(S_0 \leq s \supset \forall stu, c, g_1, g_2\ (Grade(stu, c, g_1, s) \wedge Grade(stu, c, g_2, s) \supset g_1 = g_2)). \qquad (5)$$

According to Reiter [32], this formula should be a logical consequence of a correct specification of the form described above; actually in our example this is the case. Otherwise,

[3] In this paper we consider only static integrity constraints. A methodology for treating dynamic integrity constraints as static integrity constraints is presented in [1].

we should have to embed the IC into a modified specification of the same form as before or we should have to generate active rules for making the IC hold. □

Among others, we find the following advantages in using the SC as a specification language: (1) It has a clear and well understood semantics. (2) Everything already done in the literature with respect to applications of predicate logic to DBs can be done here. In particular, all static and extensional aspects of databases and query languages are included. (3) Dynamic aspects at the same object level can be considered, in particular, it is possible to specify how the database evolves as transactions are executed. (4) It is possible to reason in an automated manner from the specification and to extract algorithms for different computational tasks from it. (5) In particular, it is possible to reason explicitly about DB transactions and their effects. (6) In this form it is possible to extend functionalities of usual commercial DBMSs.

3 Integrity Constraints and Internal Actions

In the following, we distinguish between agent or user actions and *internal actions*[4]. We have already considered the first class; they are user defined primitive transactions, and appear explicitly in a possibly more complex user transaction [5]. Instead, the internal actions are executed by the database management system as a response to the state of the database. They will be executed immediately when they are expected to be executed with the purpose of restoring the integrity of the database.

In the rest of this paper, with the purpose of illustrating our approach, we will consider only ICs of the form

$$\forall s(S_0 \leq s \supset \forall \bar{x}\ (R_1(\bar{x}, s) \wedge R_2(\bar{x}, s)\ \supset\ R(\bar{x}, s)), \tag{6}$$

where R_1, R_2, R are table names, or negations of them; and the variables in each of them in (6) are among the variables in $\bar{x}$. The Rs could also be built-in predicates, like the equality predicate. In particular, as described in a later example, functional dependencies, fall in this class.

A basic assumption here is that an IC like (6) is not just a logical formula that has to be made true in every state of the database, but it also has a *causal intention*, in the sense that every time the antecedent becomes true, necessarily the consequent has to become true as well, whereas from a pure logical point of view, just making the antecedent false would work (see [28, 16, 34] for a discussion about the logical representation of causal rules.).

Example 5. (cont'd) The functional dependency (5) has the form (6). Nevertheless, it is not written in a "causal" form. I.e., the intention behind the axiom is not to state that if two grades g_1 and g_2 are recorded for the same student in a course at a given situation,

[4]In [25] they are called *natural actions*.

[5]Complex database transactions have been treated in [6]. In this paper we restrict ourselves to primitive transactions only.

then both grades are caused to be the same. Actually, it would make no sense to try to enforce this. A more appropriate way to write (5) is

$$\forall s(S_0 \leq s \supset \forall stu, c, g_1, g_2\ (Grade(stu, c, g_1, s) \wedge g_1 \neq g_2\ \supset\ \neg Grade(stu, c, g_2, s), \quad (7)$$

that is, a student having a certain grade is the cause for the same student not having any other different grade. □

For each IC of this form, we introduce an internal action name A_R with as many arguments as non situational arguments appear in R. We introduce a new predicate, *Internal* on actions; then, for A_R, we specify

$$Internal(A_R(\bar{x})), \quad (8)$$
$$Poss(A_R(\bar{x}), s)\ \supset\ R_1(\bar{x}, s) \wedge R_2(\bar{x}, s) \wedge \neg R(\bar{x}, s), \quad (9)$$
$$Poss(A_R(\bar{x}), s) \supset R(\bar{x}, do(A_R(\bar{x}, s)). \quad (10)$$

This says that: (a) the new action is internal; (b) a necessary condition for the internal action to be possible is that the corresponding IC is violated; and (c) if it possible, then, after the internal action is executed, the R, mentioned in the head of (6), becomes true of the tuple $\bar{x}$ at the successor state.

Notice that the right-hand side of (9) should be an evaluable or domain independent formula [35, 11]. In addition, as mentioned in example 5, and according with the causal view of ICs, the literal R there should not be associated to a built-in predicate, because its satisfaction is enforced through formula (10).

As discussed later on, there may be extra necessary conditions at a state s to specify for the execution of A_R at s. Once all these necessary conditions have been collected, they can be placed in a single axiom of the form

$$Poss(A_R(\bar{x}, s))\ \supset \varphi_{A_R}(\bar{x}, s). \quad (11)$$

Later, we appeal to Clark's completion [10] for the possibility predicate for the internal action. Thus, transforming necessary conditions into necessary and sufficient conditions. Thus, replacing $\supset$ by $\equiv$ in (11).

In our example, $\varphi_{A_R}(\bar{x}, s)$ would contain the right-hand side of (9) among other things, but not the right-hand side of (10) that corresponds to a new effect axiom.

The need for extra necessary conditions for the repairing action A_R in (9), is related to specific repair policies to be adopted. In our running example, we might decide that when a new grade is inserted for a student in a course, then the old grade has to be eliminated in favor of the new one. Therefore, adding the new grade will be equivalent to performing an update. This should be the effect of the internal, repairing action. Therefore, the extra necessary condition for this action is that the grade to be eliminated was in the database before the new one was introduced. This sort of historical conditions can be handled in our formalism by means of virtual tables that record the changes in the tables [6] (in our example, the old grade would not be recorded as a change, but the new one would; see example 6 below); or by means of a methodology, presented in [1] and based on [8], for specifying the dynamics of auxiliary, virtual, and history encoding views defined from formulas written in past temporal logic.

Once the internal action is introduced, in order to produce the effects described in (10), the action A_R has to be inserted in the SSA of the corresponding table: if the right-hand side of (6) is a table R, with SSA like (4), then $a = A_R$ must now appear in $\gamma_R^+(a, \bar{x}, s)$ in (4) to make R true. If the right-hand side is $\neg R$, then $a = A_{\neg R}$ must be appear in $\gamma_R^-(a, \bar{x}, s)$, to make R false.

Action A_R is specified to make R true. The fact that R is different from R_1 and R_2, or even when R is the same as, say, R_1, the fact that the arguments to which the literals apply are different, will cause that A_R will not affect the truth of $R_1(\bar{x}, s) \wedge R_2(\bar{x}, s)$; it will persist from s to $do(A_R(\bar{x}), s)$, making the IC true at $do(A_R(\bar{x}), s)$, that is, A_R has a repairing effect. Notice also that the IC is not satisfied at the state s, where the internal action A_R will be forced to occur. In this sense, s will be an "unstable" situation.

Since, later on, we will force internal actions to occur at corresponding unstable situations, we define a *stable situation* as every situation where no internal actions are possible:

$$stable(s) \;\equiv\; \neg\exists a\; (Internal(a) \wedge Poss(a, s)). \tag{12}$$

Intuitively, the stable situations are those where the integrity constraints are to be satisfied. Instead, unstable situations are a part of a sequence of situations leading to a stable situation; in those unstable, intermediate situations, ICs do not need to hold. The transition to a stable situation is obtained by the execution of auxiliary, repairing actions.

The specification in (11) suggests the introduction of the following *active rule*:

$$E_{A_R}(\bar{x});\; \{\varphi_{A_R}(\bar{x})\} \;\Rightarrow\; A_R(\bar{x}); \tag{13}$$

where $E_{A_R}(\bar{x})$ is an *Event* associated to the IC that corresponds to changes produced in the database, for example, the insertion of $\bar{x}$ in the tables appearing in (9). This event causes the rule to be considered. Then, *Condition* $\{\varphi_{A_R}(\bar{x})\}$ is evaluated. It includes the fact that R_1, R_2 became true and $\neg R$ became false. If this *Condition* is satisfied, then the *Action* A_R is executed.

An alternative to rule (13) would be to skip E_{A_R}, and always check *Condition* $\{\varphi_{A_R}(\bar{x})\}$ after any transaction, but including in the *Condition* the information about the changes produced in the database by keeping them in an auxiliary view, so that they can be taken into account for the *Action* to be executed. This approach is illustrated in the next example.

Example 6. (cont'd) Given the functional dependency (7), we introduce the internal action $A_{Grade}(stu, c, g)$. Assume that we already have a view

$$Changes_{Grade}(stu, c, g, s)$$

that records the changes in *Grade*. That is, $Changes_{Grade}(stu, c, g, s)$ means that $Grade(stu, c, g, s) \wedge \neg Grade(stu, c, g, s')$, where s' is the situation that precedes s. Then, the precondition axiom for the new action is

$$\begin{aligned} Poss(A_R(stu, c, g), s) \;\equiv\; & Grade(stu, c, g_1, s) \wedge g_1 \neq g \;\wedge Grade(stu, c, g, s) \\ & \wedge Changes_{Grades}(stu, c, g_1, s). \end{aligned} \tag{14}$$

The new action should have the effect of deleting tuple (stu, c, g) from *Grades*. This is specified with the effect axiom:

$$Poss(A_R(stu, c, g), s) \supset \neg Grade(stu, c, g, do(A_R(stu, c, g), s)) \tag{15}$$

Thus, when the IC is violated (this violation is expressed by the first three conjuncts on the right side of (14)), the action A_R is possible. When the internal action becomes possible, it must be executed. As a result of the execution, the repair is carried out, by eliminating the old grade.

Notice that predicate *Changes* could be pushed to the *Event* part of the rule, as discussed before, because it keeps record of the changes in the database. The dynamics of an auxiliary view like *Changes* could be specified, and in this way, integrating everything we need into the same specification, by means of a corresponding SSA. This can be achieved by means of a general methodology developed in [2] and [1] for deriving SSAs for views and history encoding relations, resp.

4 Specifying Executions

The approach presented in the previous section is still incomplete. Indeed, for this approach to work, we need to address two independent but important problems. First, we need to specify that executions should be enforced, since, up to now, the formalism presented deals with hypothetical executions. Second, we need to deal with the problem of executions of *repairing actions* arising from several ICs violated in the same situation. We deal with these issues below.

Notice that there is nothing in our SC specification that forces the action A_R to be executed. The whole specification is hypothetical in the sense that *if the actions ... were executed, then the effects ... would be observed.*. Thus, from the logical specification of the dynamics of change of a traditional database, it is possible to reason about all its possible legal evolutions. However, these specifications do not consider transactions that *must be executed* given certain environmental conditions (i.e., the database state). In order to include active rules in this style of specifications, it is necessary to extend the situation calculus with *executions*. This is necessary, given that the actions specified by active rules must be *forced to be executed* when the associated *Event* happens and the corresponding *Condition* is satisfied. That is, the future is not open to all possible evolutions, but constrained by the necessary execution of actions mentioned in the rules that fire, given that their related conditions hold.

The notion of execution in SC was first introduced in [27][6]. This problem has subsequently been treated in [21, 26]. Our discussion is based upon [26]. The starting point is the observation that every situation s identifies a unique sequence of actions. That is, situations can be identified with the history of actions that lead to them (starting in S_0): $s = do(a_n, \ldots (do(a_2, do(a_1, S_0)) \ldots)$. We say that the actions $a_1, a_2, \ldots, a_n$ belong to the history of s. The predicate *executed*, that takes an action and a situation as arguments, is introduced in order to specify constraints in valid or *legal* histories. For illustration

[6]It was called *occurrence* there.

purposes, let us assume that we have situations S, S', such that $S < S'$. Further, assume that we specify that $executed(A, S)$. The fact that A has to have been executed in S, from the perspective of S', should entail that action A must appear in the history of S' (unless S' were not legal), immediately after S. To specify such a constraint we use the predicate *legal* for situations. This predicate characterizes the situations that conform to the executions that should arise in their histories; the specification of *legal* is as follows:

$$legal(s_h) \equiv S_0 \leq s_h \wedge \forall a, s\ (s < s_h \wedge executed(a, s) \supset do(a, s) \leq s_h). \tag{16}$$

The notion of legality defined with the *legal* predicate, introduces a more restrictive form of legality for situations, than the notion strictly based upon the *Poss* predicate. A situation is considered *legal*, in this more restrictive sense, if the executions that must arise in its history appear in it, and if all the situations in the history are reached by performing possible transactions starting in S_0. When modeling active databases, we consider that situations are legal when their histories are consistent with the intended semantics of the rule executions.

Whenever the condition of an active rule in consideration is satisfied, the action mentioned in the rule must be executed. Therefore, the specification of an active rule must include the presence of the predicate *executed* associated to its *Action*, actually, an internal action in our context. In a database whose situations are all *legal* will be such that active rules, when triggered, are properly dealt with. Thus, in a situation calculus tree we only consider branches in which actions that must be executed by rule triggerings are actually executed.

Now we can force internal actions to be executed. This is specified as follows:

$$\forall a, s\ Poss(a, s) \wedge Internal(a)\ \supset\ executed(a, s). \tag{17}$$

I.e., if an internal action is possible, it must be executed. In this way the repairing internal actions are executed immediately. Nevertheless, it should not be difficult to specify in our SC formalism delayed executions. This kind of issues, in the context of general active rules (not only for database maintenance), priorities and complex transactions, are considered in [6].

The active rule (13) is not written in the SC object language of the specification, and in that sense its semantics is not integrated with the rest of the first order semantics. Nevertheless, it can be eliminated now from the specification in favor of axioms (8), (9), (10), and (17). Recall that, in addition to the introduction of these new axioms, the SSA (4) for table R has to be modified by introducing $a = A_R$ in the formula $\gamma_R^+(a, \bar{x}, s)$, since new positive effects have been specified for table R. This possible recomputation of the SSAs is a very simple task, only one action for IC with its condition has to be plugged into a SSA. If the active rules for database maintenance are given in advance, then the SSAs for the tables can be computed incorporating the corresponding actions from the very beginning.

Now, it can be proved that the following formula is a logical consequence of the new specification:

$$\forall s (S_0 \leq s \wedge stable(s)\ \supset\ \forall \bar{x}\ (R_1(\bar{x}, s) \wedge R_2(\bar{x}, s)\ \supset\ R(\bar{x}, s)). \tag{18}$$

It says that the IC is satisfied at all stable situations of the database.

There is, however, a problem with the above axiomatization in the context of our specification. The specification of executions in [26] is given in a situation calculus with concurrent (simultaneous) actions. In our specification, primitive actions are executed non concurrently. This may be a problem if two separate IC repairing actions are possible in the same situation. In fact, assume that two separate ICs are violated in a given situation S. Assume further that there are two internal repairing actions A_1 and A_2, that are defined for each of these two ICs. Since both ICs are violated in S, then we need both A_1 and A_2 to be executable in S.

The situation calculus that we have been using is *non-concurrent*, in the sense that given a situation s, any successor situation is obtained by the execution of a single primitive action. one way out of this problem is to ensure that the views that record changes to the databases (*Changes* in the running example) are updated only after *non-internal* actions are executed. In the example, we can non-deterministically execute a_1, and reevaluate the applicability of a_2 once the first repairing action has been executed. In this case, it would be possible to have a_1 repair both ICs without having to execute a_2. It would also be possible to have situations where one repair introduces other violations to ICs, forcing yet other repairs. Chaining of repairs and further details related to this problem are still to be worked out. In order for this approach to work, we need to drop axiom (17) in favor of:

$$\forall a, s \; Poss(a, s) \wedge Internal(a) \;\supset\; pexecuted(a, s), \tag{19}$$

$$(\forall s)[(\exists a) pexecuted(a, s) \supset (\exists b) executed(b, s) \wedge pexecuted(b, s)]. \tag{20}$$

Here, we introduce the predicate *pexecuted* to represent the notion of *possibly executed*. Thus, if some action is possibly executed in a situation s, then some possibly executed action *must* be executed in s. Notice that this has a non-deterministic flavor. Thus, if several internal actions are possible, then the specification is satisfied if either of them is executed.

5 A Causal Approach to Integrity Constraints

In this paper we have not considered in detail the problem of determining all possible repairs of a database. From a logical point of view, there are many possible minimal repairs for an inconsistent database [13, 3]. In principle, we could choose any of them and specify corresponding maintenance rules for enforcing that particular kind of repair. This could be accommodated in our formalism. Nevertheless, we might have some preference for some repairs instead of others. For example, we may want to keep the changes produced by a sequence of primitive transactions even in the case they take the database to a state that that does not satisfy the ICs. In this case, we would generate new, additional changes that restore the consistency of the database, pruning out some of the logically possible repairs (like the ones that undo some of the new primitive transactions). This kind of repairs are possible only if the updated database is consistent with the ICs [33] [7].

[7] The updated database DB'' may not satisfy the IC, i.e $DB' \not\models IC$, but DB' seen as a first order theory ([30]) together with IC may be a consistent theory.

In our approach, there is implicit a notion of causality behind the ICs (see example 5). There are cases where the ICs have an implicit causal contents, and making them explicit may help us restrict ourselves, as specifiers, to some preferred forms of database repairs, like in our running example. Introducing explicit causality relations into the ICs can be seen as form of user intervention [7], that turns out to be a way of predetermining preferred forms of database repairs.

It is possible to make explicit the causal relation behind a given integrity by means of a new causality predicate, as introduced by Lin in [17]. This avoids considering a causal relation as a classical implicative relation.

Lin's approach is also based upon the situation calculus, albeit in a different dialect. The main difference is that the fluents are not predicates but functions. For instance, in order to express that a property p is true of an object x in a situation s, we write $p(x, s)$. In the dialect used by Lin, this same statement is written as $Holds(p(x), s)$. The advantage of treating fluents at the object level, is that one can use properties as arguments in a first order setting. In particular, Lin's approach to causality is based upon the introduction of a special predicate *Caused*, which takes a fluent, a truth value[8], and a situation as arguments. Thus, one can write $\mathit{Caused}(p(x), \mathit{True}, s)$ with the intent of stating that fluent p has been caused to be *True* of x in situation s. In our framework, we will use the alternative syntax $\mathit{Caused}(p(x, s), \mathit{True})$ and will eliminate the *Caused* predicate, as discussed below.

In Lin's approach, the *Caused* predicate is treated in a special manner. First, there is an assumption, formalized using circumscription [19], that *Caused* has minimal extent. That is, *Caused* is assumed to be false, unless it must be *True*. Furthermore, if a fluent is caused to be true (false) in a situation, then it must be true (false) in that situation. If there is no cause for fluent to take a truth value, then the fluent does not change.

It turns out that it is possible, in many interesting cases, to translate Lin's handling of the *Caused* predicate, into a specification in the style proposed in this article (making use of *Internal* actions) [24]. We illustrate this approach, by interpreting *Caused* as syntactic sugar, and by providing a translation of a *causal* formula to our language.

The IC (7) of our example, can be expressed in *causal* terms as follows:

$$\mathit{Changes}_{\mathit{Grades}}(stu, c, g, s) \supset (\forall g')[g \neq g' \supset \mathit{Caused}(\mathit{Grade}(stu, c, g', s), \mathit{False})]. \qquad (21)$$

Keeping in mind that $\mathit{Changes}_{\mathit{Grades}}$ records the addition of a grade for a student in a course, the formula above should be interpreted as *if the grade* g *has been provided for student* stu *in course* c*, then there is a cause for the student not to have any other grade.*

To eliminate Lin's causality predicate, taking the specification back to the formalism based on table names, actions and situations only, we pursue the following idea. We admit the existence of unstable situations in which the causal rules (ICs) can be violated. In this unstable situations, some internal actions become possible, which, as before, repair the ICs. The approach introduces a new action function per causal rule. Let A_I denote the

[8] In Lin's framework, a special sort for truth values is introduced. The sort is fixed, with two elements denoted with the constants *True* and *False* respectively.

new action function for rule (21). The rule is replaced by the following axioms:

$$Internal(A_I(stu, c, g)). \tag{22}$$

$$\begin{aligned} Poss(A_I(stu, c, g), s) \equiv\ & Changes_{Grades}(stu, c, g, s) \wedge \\ & \neg(\forall g')[g \neq g' \supset \neg Grade(stu, c, g', s)] \end{aligned} \tag{23}$$

$$Poss(A_I(stu, c, g), s) \supset (\forall g')[g \neq g' \supset \neg Grade(stu, c, g', s)] \tag{24}$$

Notice that the elimination of the causal relation follows a mechanical procedure which can be applied to a set of *stratified causal rules*, as defined by Lin. The stratification simply ensures that there are no circular causalities. In this setting, it can be proven that both approaches, using explicit *causal* rules, and the translation lead to the same results.

It is illustrative to compare axioms (22)-(24) with the axioms obtained for the integrity constraint (7) in the approach described in section 3, which results in an axiomatization (see example 5) that yields equivalent results. Therefore, from a methodological point of view, one can use a logical language extended with a *causality* relation that would enhance the expressive capabilities of the language. This extra expressive power gives the modeler a more natural way to express preferences regarding database repairs. Furthermore, the semantics for the new causal relations can be understood in terms of a more conventional logical language.

6 Determining Events for the Maintenance Rules

So far, we have not said much about the events in the active rules (see last part of section 3). In [7], Ceri and Widom handle this problem in detail, although without saying much about deriving *Actions* for the maintenance rules. They provide a mechanism that, from the syntactic form of an IC, derives the transition predicate that determines the *Event* part of the active rule that will maintain that constraint. This predicate is defined in terms of the primitive transactions that might lead to a violation of the IC, and they present a methodology for determining those transactions. The primitive transactions they consider are insertions, deletions and updates in tables.

In our case, we have user defined primitive transactions that may affect several tables simultaneously. In addition, we know exactly, by the presence of the SSAs, how each base table evolves as legal actions are executed, and which actions may affect them. Now, it is possible to associate a view to an IC, namely the view that stores the tuples that violate the IC (hopefully this view remains empty), this is what we have in the RHS of condition (9). Since this view is a derived predicate and not one of the base tables in the database, we may not have a SSA for it. Nevertheless, as shown in [2], it is always possible to derive, in automated way, a SSA for a view. Then, we may easily compute a SSA for the violation predicate (or view) associated to an IC.

Having a SSA of the form (4) for the violation predicate, we can easily detect, actually which are the primitive transactions that can make it change, in particular, become non empty when it was empty before the transaction, and then showing a violation of the corresponding IC (this can be detected from the γ^+ part of the SSA). In this way, we are in position to obtain a mechanism for determining events as in [7], but for more general primitive transactions. In [2] other applications of SSAs for views storing violating tuples of ICs are presented.

7 Conclusions

In this paper we have considered the problem of specifying policies for database maintenance in a framework given by the specification of the dynamics of a relational database. The specification is given in a language that includes both primitive actions and database states at the same level as the objects in the database domain.

Repairing actions are introduced for the integrity constraints that are expected to be satisfied. They are integrated into the original specification, by providing their effects and preconditions. Then simple active rules are created for repairing the ICs. Since these active rules do not have, as stated, a predicate logic semantics, there are alternatively specified in the same formalism as the database dynamics.

The ICs are expected to be logical consequences of the modified specification, which must be true at every legal state of the database. Nevertheless, IC violations may give rise to a transition of the database along a sequence of executions of repairing actions, during which the ICs are not necessarily satisfied. Since we may not exclude those intermediate states from the database dynamics, we distinguish in our formalism between stable and unstable states. It is only at stable states where the ICs have to be satisfied, and this can be proved from the new specification. Instead, the unstable states are related to executions of the repairing actions.

Since the original specification has a hypothetical nature, in the sense of describing what the database would be like if the actions were executed, and then not forcing any action executions, the formalism was extended by a notion of executed action. In this way we can deal with the imperative nature of active rules, thus forcing executions of repairing actions.

We have considered the derivation of repairing actions for a simple case of actions (primitive actions) and ICs. Reparations policies for more complex cases of ICs based on sequences of atomic transactions [12] could be integrated in our specification formalism. For doing this, a treatment of more complex active rules would be necessary. In [6], we developed in an extended formalism for specifying a database dynamics, the whole framework needed for specifying this kind of active rules, including complex user transactions and the *Actions* in the rules, priorities among rules, database transitions and rollbacks.

By using the derived specification of the dynamic of views storing the tuples that violate an IC, it is possible to determine the right events for the maintenance rules. For restoring the consistency new, internal, primitive actions are introduced. Which repairing actions will be introduced and with which effects may depend on the causal contents that the user attributes to the ICs.

References

[1] M. Arenas and L. Bertossi. Hypothetical Temporal Queries in Databases. In A. Borgida, V. Chaudhuri, and M. Staudt, editors, *Proceedings "ACM SIGMOD/PODS 5th International Workshop on Knowledge Representation meets Databases (KRDB'98): Innovative Application Programming and Query Interfaces*, pages 4.1 – 4.8, 1998. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10/.

[2] M. Arenas and L. Bertossi. The Dynamics of Database Views. In Burkhard Freitag, Hendrik Decker, Michael Kifer, and Andrei Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*, pages 197–226, Berlin, 1998. Springer-Verlag.

[3] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *Proc. ACM Symposium on Principles of Database Systems*, pages 68–79, Philadelphia, 1999.

[4] L. Bertossi, M. Arenas, and C. Ferretti. SCDBR: An Automated Reasoner for Specifications of Database Updates. *Journal of Intelligent Information Systems*, 10(3):253–280, 1998.

[5] Leopoldo Bertossi, Javier Pinto, Pablo Saez, Deepak Kapur, and Mahadevan Subramaniam. Automating Proofs of Integrity Constraints in Situation Calculus. In Zdzislaw Pawlak and Zbigniew W. Ras, editors, *Proc. of the 9th International Symposium on Methodologies for Intelligent Systems*, number 1079 in Lectures Notes in Artificial Intelligence, pages 212–222. Springer-Verlag, 1996. ISBN 3-540-61286-6.

[6] Leopoldo Bertossi, Javier Pinto, and Ricardo Valdivia. Specifying Database Transactions and Active Rules in the Situation Calculus. To appear, 1999.

[7] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. 16th VLDB Conference*, pages 566–577, Brisbane, Australia, 1990.

[8] J. Chomicki. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Transactions on Database Systems*, 20(2):149–186, June 1995.

[9] J. Chomicki and G. Saake, editors. *Logics for Databases and Information Systems.* Kluwer Academic Publishers, Boston, 1998.

[10] K.L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[11] A. Van Gelder and R. Topor. Safety and correct translation of relational calculus formulas. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 313–327, 1987.

[12] M. Gertz. *Diagnosis and Repair of Constraint Violations in Database Systems.* PhD thesis, Universität Hannover, 1996.

[13] M. Gertz. An extensible framework for repairing constraint violations. In S. Conrad, M-J. Klein, and K-D. Schewe, editors, *Working Notes "6th International Workshop on Foundations of Models and Languages for Data and Objects"*, 1996.

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

[15] F. Lin and R. Reiter. How to Progress a Database. *Artificial Intelligence*, 92(1–2):131–167, 1997.

[16] Fangzhen Lin. Embracing Causality in Specifying the Indirect Effects of Actions. In *Proc. International Joint Conference on Artificial Intelligence, Montreal*, pages 1985–1991. Morgan Kaufmann, 1995.

[17] Fangzhen Lin. Embracing Causality in Specifying the Indirect Effects of Actions. In *Proc. International Joint Conference on Artificial Intelligence, Montreal*, pages 1985–1991. Morgan Kaufmann, 1995.

[18] 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.

[19] John McCarthy. Circumscription a form of Non-Monotonic Reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[20] Sheila A. McIlraith. Representing actions and state constraints in model-based diagnosis. In *Proceedings of the National Conference on Artificial Intelligence (AAAI–97)*, pages 43–49, 1997.

[21] Rob Miller and Murray Shanahan. Narratives in the Situation Calculus. *The Journal of Logic and Computation*, 4(5):513–530, 1994.

[22] Jack Minker. Logic and Databases. Past, Present and Future. *AI Magazine*, pages 21–47, 1997.

[23] Javier Pinto. *Temporal Reasoning in the Situational Calculus.* PhD thesis, Department of Computer Science, University of Toronto, 1994.

[24] Javier Pinto. Causality, indirect effects and triggers (preliminary report). In *Seventh International Workshop on Non-monotonic Reasoning*, 1998.

[25] Javier Pinto. Concurrent actions and interacting effects. In A.G. Cohn, L.K. Schubert, and S.C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 292–303, San Francisco, CA, 1998. Morgan Kaufmann Publishers.

[26] Javier Pinto. Occurrences and Narratives as Constraints in the Branching Structure of the Situation Calculus. *Journal of Logic and Computation*, 8:777–808, 1998.

[27] Javier Pinto and Raymond Reiter. Adding a Time Line to the Situation Calculus. In *The Second Symposium on Logical Formalizations of Commonsense Reasoning*, pages 172–177, January 1993.

[28] Javier A. Pinto. Causality in Theories of Action. In Rob Miller and Murray Shanahan, editors, *Common-Sense '98*, pages 349–364. Queen Mary and Westfield College, University of London, January 1998. Also found at `URL = http://www.ida.liu.se/ext/etai/nj/fcs-98/349/paper.ps`.

[29] Javier A. Pinto. Compiling Ramification Constraints into Effect Axioms. *Computational Intelligence*, 13(3), 1999.

[30] R. Reiter. Towards a logical reconstruction of relational databases theory. In J. Mylopoulos and J. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, pages 191–233. Springer-Verlag, 1984.

[31] 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.

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

[33] K-D. Schewe and B. Thalheim. Limitations of the rule triggering systems for integrity maintenance in the context of transition specifications. *Acta Cybernetica*, 13:277–304, 1998.

[34] Michael Thielscher. Ramification and causality. *Artificial Intelligence*, 89:317–364, 1997.

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

[36] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, Inc., 1996.

[37] C. Zaniolo, S. Ceri, Ch. Faloutsos, R. T. Snodgrass, V.S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.