Exploiting a Relational DBMS for Reasoning about Actions

Giuseppe De Giacomo[1] and **Fabio Palatta**[2]

Abstract. It is known that Reiter's situation calculus basic action theories are tightly related to relational databases, when complete information on the initial situation is available. In particular the information on the initial situation can be seen as a relational database and action, as specified by the preconditions and successor state axioms can be seen as operations that change the state the the database. In this paper we show how to exploit such a correspondence to build a system for reasoning about actions based on standard relational database technology. In particular, by exploiting a relational DBMS, the system is able to perform both Projection and Progression in very large action theories. A prototype of the system described in the paper has been implemented, and is currently used to do experimentation on the actual capabilities of the approach.

1 INTRODUCTION

Typically, in Cognitive Robotics, we assume that the cognitive agent, the robot, is equipped with a representation of the world and a specification of how its and other agents actions affect the world. There are several choices of the representation and reasoning formalism to adopt for this task. Among them the situation calculus as revised by Reiter and others is emerging as a general tool to express action theories [9, 12].

Often in modeling the world we have to deal with incomplete information of various form. In the situation calculus we typically have incomplete information on the initial situation. However, at least in certain domain, the main issue is not how to cope with incomplete information, but how to deal with hundred thousand facts the describe the current state of the world. In order to do this, we need to be able to scale up action theories, and especially being able to reason with so many facts.

It is known that Reiter's basic action theories are tightly related to relational databases, when complete information on the initial situation is available. In particular the information on the initial situation can be seen as a relational database and action as specified by the preconditions and successor state axioms can be seen as operations that change the state the the database [11, 10, 6].

In this paper we show how to exploit such a correspondence to build a system for reasoning about actions based on standard relational database technology [8, 4]. In particular, by exploiting a relational DBMS, the system is able to perform in very large action theories both Projection, i.e., evaluate a certain (either open or closed) formula in the world resulting from executing a sequence of actions, and Progression, i.e., progress to initial situation to the situation resulting from executing a sequence of actions. We observe that these functionalities, that include the ability of performing projection of closed formulas, allow for using such a system in conjunction with high level robot language interpreters so as to exploit its formula evaluation capabilities for evaluating tests and action preconditions [5, 2].

2 BASIC ACTION THEORIES

Our account of action and change is formulated in the language of the situation calculus [7, 12]. We will not go over the language here except to note the following components. The language is multisorted, in particular we have a sort $Action$ for actions a sort $Situation$ for situations, and a sort $Object$ for all other terms. In fact, we allow to specialize the sort *Object*, creating sorts corresponding to the various type of objects in the domain. A special constant S_0 is used to denote the *initial situation*, namely the one in which no actions have yet occurred. There is a distinguished binary function symbol *do* where $do(a, s)$ denotes the successor situation to s resulting from performing action a. Relations whose truth values vary from situation to situation, are called (relational) *fluents*[3], and are denoted by predicate symbols taking a situation term as their last argument. A special predicate $Poss(a, s)$ is used to state that action a is executable in situation s.

Within this language, we can formulate action theories that describe how the world changes as the result of the available actions. In this paper we focus on Reiter's basic action theories [9, 12]. These theories are formed by:

- Axioms describing the initial situation S_0 which form the initial database D_0.
- Action precondition axioms of the form:

$$Poss(a(\vec{x}), s) \;\equiv\; \phi_a(\vec{x}, s)$$

  one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms of the form:

$$F(\vec{x}, do(\alpha, s)) \;\equiv\; \gamma_F^+(\vec{x}, \alpha, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, \alpha, s)$$

  one for each fluent F, stating under what conditions $F(\vec{x}, do(\alpha, s))$ holds as function of what holds in situation s. For successor state axioms the following consistency requirement must hold:

$$\neg\exists\vec{x}, \alpha, s.[\gamma_F^+(\vec{x}, \alpha, s) \wedge \gamma_F^-(\vec{x}, \alpha, s)]$$

[1] Dipartimento di Informatica e Sistemistica Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy, email: degiacomo@dis.uniroma1.it

[2] Dipartimento di Informatica e Sistemistica Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy, email: tuareg@tiscalinet.it

[3] We do not consider functional fluents here.

which states that is never the case that both γ_F^+ and γ_F^- hold for the same arguments. The successor state axioms take the place of the so-called effect axioms, but also provide a solution to the frame problem [9].

- Unique names axioms for the primitive actions, plus some foundational, domain independent axioms.

3 SITUATION CALCULUS AND RELATIONAL DATABASES

In this paper we consider basic action theories with the following further constrains:

- We assume that *Object* contains a finite number of objects, that each of them is denoted by a constant and that for these holds the unique name assumption.
- For each fluent F and each tuple of objects $\vec{t}$ the theory either logically implies $F(\vec{t}, S_0)$ or $\neg F(\vec{t}, S_0)$. That is we have complete information on the initial situation. Under this assumption we have that F is characterized, in the initial situation, simply by the set of facts of the form $F(\vec{t}, S_0)$ that are logically implied by the theory, enforcing the closed world assumption.

These two constraints are quite severe and certainly are not suitable for several applications [12]. However if our application does allow us to make such assumptions, then we can use, without loss of generality, formula evaluation instead of logical implication (confusing entailment with truthness) thus getting a quite efficient method to base reasoning on [3]. Moreover under these assumption results in [6] show that progression, i.e., transforming the action theory by updating the initial situation accordingly with the results of performing a sequence of actions, which is a very difficult task in general, becomes straightforward.

These two observations, allows us to exploit standard relational database technology as the base of a system for reasoning about actions. In particular we can use relational database querying techniques to perform formula evaluation, thus being able to exploit sophisticated optimization techniques developed for databases in order to deal with very large amounts of facts. On the other hand when the initial situation needs to be progressed we can exploit the standard update mechanism developed for databases, and take advantage of the transactional support to guaranty consistency even in case of a failure during the progression. The relational database technology promises to be a good vehicle to scaling up reasoning about actions (though of a limited form) to realistic domains that require dealing with a lot of information.

In order to develop such a system, we need to decide how to tackle the following issues: (i) how to represent fluents and their values as database state (keep in mind that we are assuming complete information on the fluents); (ii) how to represent arbitrary (both open and closed) situation calculus formulas[4] as SQL queries; (iii) how to realize an action as database update. We deal with each of these issues below.

3.1 Representing system states as database states

For simplicity, here, we assume that all the constant symbols present in the action theory whose sort is not action or situation belong to a general sort *Object*. Later, we will allow to specialize the sort *Objects*, and this specialization will introduce interesting improvements in the implementation.

We assume that the database does not store explicitly objects of sort situation inside the tables, but the database itself may be considered as a "snap shot" taken of each fluent in the current situation. In other worlds the database keeps track of the *state* of the system, i.e., the truth values of the fluents for all objects.

Specifically, we associate to each fluent $F(\vec{x}, s)$ of arity $k+1$, defined in the action theory, a table F composed by k columns $(f1, ..., fk)$; these tables will be populated by the k-tuple of objects, that make the fluent F true in the situation represented. The database will be initialized with the description of the initial situation, in the obvious way:

$$F = \{\vec{o} \mid \vec{o} \text{ is a } k\text{-tuple of objects and } F(\vec{o}, S_0) = \mathit{True}\}$$

To complete the representation, we define another relational table, called `Objects`, composed by a single column called "`object`", that contains the whole set of objects of sort *Object*. This table will be used to translate logical negation ($\neg$), as we shall see below.

3.2 Formulas translation

Next we show how every (uniform) situation calculus formula, of arbitrary complexity (either open or closed), is translated into a SQL query. Again keep in mind that we are dealing with complete information only.

Let us remind that there are several ways to translate (open/closed) first-order formulas into as SQL query (see, e.g., [1, 13]). Here we propose a simple method that works directly by induction on the structure of the formula.

Let s be the situation represented by the current database state, and let $F(\vec{t}, s)$ be an atom, where F is a fluent and $\vec{t} = t_1, \ldots, t_n$ be the object arguments of F with $t_1, \ldots, t_k$ free variables, and $t_{k+1}, \ldots, t_n$ objects. Then $F(\vec{t}, s)$ is translated into the following SQL query:

```
select f1 as t1, ...,fk as tk
from F
where fk_+_1=tk_+_1, ...,fn=tn
```

This query returns a table containing all the substitutions for variables $t_1, \ldots, t_k$ that make $F(\vec{t}, s)$ true. In general, to an open formula we associate a table structured as follows: the number of columns in the table correspond with the number of free variables present in the formula, the names indexing the columns are the names of the variables, the tuples contained in the table represent those variable assignments that make the formula true.

Let now $F(\vec{t}, s)$ be closed (i.e., ground). Since it does not contain any free variable, we get a boolean query. SQL does not have a built in mechanism to treat boolean queries, so we create one constant relational table `TRUE`, with a single column, named `true`, containing a unique element: the constant symbol `tt`. Using the table `TRUE`, we can now translate the formula into an SQL query as follows:

```
select true
from F, TRUE
where f1=t1, ..., fn=tn
```

Observe that if $F(\vec{t}, s)$ is false then the above query returns the empty set, otherwise it returns the special symbol `tt`. We use the above technique every time we need to express a boolean query.

Let's now consider negation. Let us assume that, to an open formula $\Phi(\vec{x}, s)$ we have associated a table `Formula`, containing the

[4] To be precise, we focus on *uniform formulas*, i.e., those formulas whose fluents have as situation argument the same situation term, see [12].

set of tuples representing those variables assignments for $\vec{x}$ that make $\Phi(\vec{x}, s)$ true. Then $\neg\Phi(\vec{x}, s)$ has to be translated into a SQL query returning the set of tuples representing those variables assignments that make Φ false. Such a result can be obtained as the difference of the set of tuples in *Objects*^{n} (where n is the arity of $\vec{x}$), which correspond in relational terms to the table obtained by $n-1$ join operations on `Objects` with itself, and the set of tuples contained in table `Formula`.

```
select o1.object as x1, ..., oh.object as xn
from Objects o1, ..., Objects on
  except
select f1 as x1, ..., fh as xn
from Formula
```

The high cost of negation is evident: negation requires $n-1$ joins on a relational column containing a, potentially very high, number of items (every object in the domain).

If Φ is closed, then `Formula` either contains the single special symbol `tt` or is empty. The SQL translation of $\neg\Phi$ is then:

```
select true
from TRUE
  except
select true
from Formula, TRUE
```

Obviously the cost of negation in this case is negligible.

To translate conjunctions and disjunction we can make use of the SQL operators `intersect` and `union` respectively. However we have to deal with the non-homogeneity of the conjuncts (disjuncts), in the sense that they may share only some variables, while the others may be distinct. This can be treated nicely in the case of conjunctions by translating the conjunction with a *join* on the shared variables. In the case of disjunctions the translations is more expensive. Let's consider the open formula $\Phi_1(\vec{x}, \vec{y}, s) \vee \Phi_2(\vec{x}, \vec{z}, s)$ with free variables $\vec{x}, \vec{y}, \vec{z}$. By induction we will have a table `Formula1` corresponding to Φ_1 and a table `Formula2` corresponding to Φ_2. However before making the union of this two tables we need to make them homogeneous adding missing attributes ($\vec{z}$ for `Formula1` and $\vec{y}$ for `Formula2`). Thus we get:

```
select frm1.f1 as x1, ..., frm1.fh as xh,
       frm1.fh_+_1 as y1, ..., frm1.fn as yk,
       o1.object as z1, ..., ol.object as zl
from Formula1 frm1,
     Objects o1, ..., Objects ol
  union
select frm2.f1 as x1, ..., frm2.fh as xh,
       o1.object as y1, ..., ol.object as yk
       frm2.fh_+_1 as z1, ..., frm2.fm as zl,
from Formula2 frm2,
     Objects o1, ..., Objects ok
```

If both Φ_1 and Φ_2 are closed, then we get a much simplified SQL query:

```
select true
from Formula1 frm1, TRUE
  union
select true
from Formula2 frm2, TRUE
```

While if just one of them is closed say Φ_2 we get:

```
select x1,... ,xn
from Formula1
  union
select o1.object as x1, ..., on.object as xn
from Formula2, Objects o1, ... , Objects on
```

Existentially quantified formulas $\exists x_1.\Phi(\vec{x}, s)$ are translated by projecting out x_i from the table `Formula` associated with Φ:

```
select x1,...,xi-1,xi_+_1, ...,xh
from Formula
```

Universally quantified formulas $\forall x_1.\Phi(\vec{x}, s)$ are simply treated as an abbreviation for $\neg\exists x_1.\neg\Phi(\vec{x}, s)$.

It is important to stress that the translation process is not performed by applying recursively the translation rules above in a top-down fashion. It works exploring the formula starting from its atomic elements and proceeding bottom-up to obtain the final SQL query. So the fluents present in the formula are translated first, and the respective relational tables are created on the database. Tables associated with AND, OR, NOT, and with existential quantifiers are created only after their operands or bodies have been translated. In addition, the data structure used to store the formula in memory is a DAG (Direct Acyclic Graphs), in this way we avoid nodes duplication in the graphs and significantly improve efficiency. Indeed complex formulas, especially those obtained by regression, contain very often recurring subformulas, the elimination of these redundant computations improve the efficiency of the system by reducing the number of relational tables to elaborate and reusing the same table each time an identical subformula recurs. In practice, if an atomic subformula has more occurrences in the formula, it will be inserted just one time (the first one) in the graph, likewise if one of the operands of an operator or the argument of a quantifier, is already present in the DAG, it will be linked to the new operator or quantifier node without any duplication, and so on.

It should also be mentioned that the system performs some pre-elaboration of the formula in order to apply some simple form of simplifications. In particular, it is important to limit the number of negations and of non-homogeneous disjunctions.

Finally, we remind the reader that for simplicity in the discussion above we have assumed that terms that were not actions or situations were of a single sort *Object*. In fact, it is much better to partition the sort *Object* into several subsorts according to the different kind of objects we have in the domain. This, on the one hand, gives a better coherence between the representation and real world. On the other hand, it improves computational efficiency. Indeed, having introduced such subsorts of *Object*, we may restrict a fluent to take arguments of predefined types, in this way the formula complementation will be no longer executed respecting powers of *Object*, but in relation with Cartesian Products of predefined sets of items, classified by type. Each set is a subset of *Object*, so the product will return a smaller table, the improvement achieved may be very sensible. The implemented system allows to partitions the sort *Object* into subsorts according to what is required by the domain of interest.

3.3 Performing actions and changing the database state

Normally we will be interested in querying the current database state, possibly exploiting regression, in order to make some projection on the future.

From time to time we will want to actually progress the database changing the database state according to the action performed. In order to do this we need to clarify how we exploit the successor state axioms as a specification of the updates to perform.

Let the successor state axiom for the fluent F be the following:

$$F(\vec{x}, do(\alpha, s)) \;\equiv\; \gamma_F^+(\vec{x}, \alpha, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, \alpha, s)$$

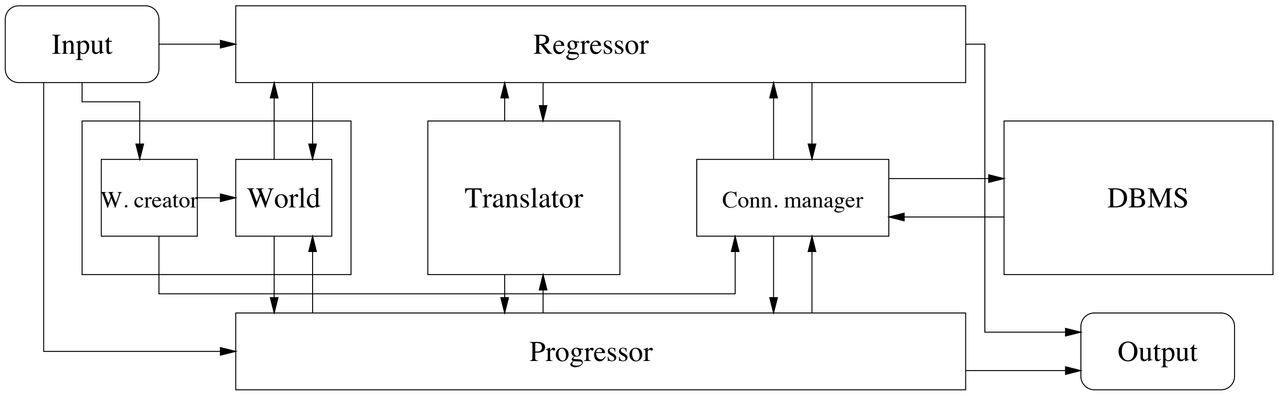

Figure 1. System Architecture

From it we can extract the *effect axioms* for each (ground) action a as follows:

$$\gamma_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s))$$
$$\gamma_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s))$$

In order to change the database accordingly, assuming `GammaFminus_a`, and `GammaFplus_a` to be the tables corresponding to $\gamma_F^-(\vec{x}, a, s)$ and $\gamma_F^+(\vec{x}, a, s)$ respectively, we need to perform the following SQL commands as a unique transaction:

```
delete from F where exists
(
  select x1, ..., xn
  from GammaFminus_a
  where x1 = f1 and ... and xn = fn
)

insert into F
(
  select x1 as f1, ... xn as fn
  from GammaFplus_a
)
```

That is, a tuple is contained in the relational table F after the the update associated with action a, if and only if, either the update has inserted it in the table, or the tuple was already in the table and the update has not deleted it. Observe that this corresponds exactly to what specified by the successor state axiom instantiated to the action a. Observe also that since the following consistency requirement holds:

$$\neg\exists\vec{x}, s.[\gamma_F^+(\vec{x}, a, s) \wedge \gamma_F^-(\vec{x}, a, s)]$$

the tables `GammaFplus_a` and `GammaFminus_a` are disjoint. Hence exchanging the order of the two statements would not change the result.

4 SYSTEM ARCHITECTURE

The system developed is based on standard relational database technologies. It is divided into two main component: a relational DBMS, which supports SQL (in particular the standard required is SQL-92 entry level, supported by virtually all commercial DBMS), and an application written in Java that communicates with the DBMS by means of JDBC (the standard Java DataBase Connectivity package).

The architecture of the system is depicted in Figure 3.2. It is composed of five main modules described in the paragraphs below.

Action Theory Module The Action Theory Module is formed by two sub-modules: the World Creator and the World. The World Creator, given the formal specification in the situation calculus of the action theory of interest, generates a representation in terms of Java classes of the action theory. The description of the initial situation is stored in the database, while the rest of the action theory is stored in the module called World.

Translator The Translator works as an interpreter between the situation calculus and SQL, implementing the translation discussed in Section 3.

Connection Manager The Connection Manager works as an interface between the application and DBMS, passing queries, updates and result sets, from one to the other.

Regressor This module takes in input a sequence (possibly empty) of actions and a (uniform) situation calculus formula, typically open. Its purpose is to evaluate the set of substitutions that make the formula true, in the situation resulting from the initial situation by performing the sequence of actions specified. Observe that the regressor does not update the database, the desired result is obtained by regressing the formula step by step along the action sequence, using the successor state axioms suitably stored in the World submodule. For each regression step the formula is translated in a new one, logically equivalent, parameterized in function of the previous state, until it arrives in the initial situation S_0. When the process stops the resulting formula, which is stored as a DAG to avoid repetitions of identical subformulas, is translated in SQL and evaluated on the DBMS.

Regressor in validation mode We may use Regressor just to test if an action sequence is legal, i.e, it does not violate the conditions imposed by the *Poss* predicate. This is achieved by giving as input to the regressor only the sentence of actions. In this case, the program will run in *validation mode* verifying wither the sequence of action is executable. For example, given a sequence of action a, b, c, it will regress the formula $Poss(a, S_0) \wedge Poss(b, do(a, S_0)) \wedge Poss(c, do(b, do(a, S_0)))$[5] making use of the precondition axioms of

[5] Observe that this regression requires some care since the formula is not uniform, it essentially requires three regressions, one for $Poss(a, S_0)$, one for $Poss(b, do(a, S_0))$ and one for $Poss(c, do(b, do(a, S_0)))$. The system

each action in the sequence. Then the formula obtained (which is to be evaluated in S_0) is translated into a boolean SQL query and evaluated against the database.

Progressor The progressor receives as input a sequence of actions, and perform a transaction in order to progress the database according to sequence of actions required. It works as follows, for each action a to be executed in the situation s it evaluates $Poss(a, s)$ translating into a boolean SQL query and evaluated against the database. If the action is legal, the progressor, interacting with World, elaborates the corresponding effect axioms, and translate them into suitable `insert` and `delete` SQL statements that are executed on the database. Then the system starts working with the next action and so on, until it reaches the end of the sequence, then it *commit*s completing with success the database progression. If one of the actions in the sequence can not be executed in the required situation, the whole sequence is aborted and all the database updates are *rolled back* so the database is left in the original state. Observe how the transactional support of the DBMS is exploited to avoid leaving the database in an undesired state in case the progression fails. Actually the transactional support can be exploited further. In those case where regression of a formula is too expensive (in general regression can generate exponential formulas even using DAGs), one can use progression to evaluate the formula with the proviso that after the evaluation the transaction corresponding to the progression gives an explicit *roll back* command to restore the initial situation. In fact, this techniques has shown to be quite effective in practice.

5 CONCLUSION

In this paper we have described how reasoning about actions, under complete information, can be potentially scaled up exploiting standard relational technology. We have described a system that realizes these ideas. A prototype of such a system has been implemented, and is currently used to do experimentation on the actual capabilities of the approach.

REFERENCES

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

[2] Giuseppe De Giacomo, Yves Lesperance, and Hector J. Levesque, 'CONGOLOG, a concurrent programming language based on the situation calculus', (2000). To appear.

[3] Giuseppe De Giacomo and Hector J. Levesque, 'Two approaches to efficient open-world reasoning'. to appear, 2000.

[4] R. A. ElMasri and S. B. Navathe, *Fundamentals of Database Systems*, Benjamin and Cummings Publ. Co., year =.

[5] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R.B. Scherl, 'GOLOG: A logic programming language for dynamic domains', *Journal of Logic Programming*, **31**, 59–84, (1997).

[6] F. Lin and R. Reiter, 'How to progress a database', *Artificial Intelligence*, **92**, 131–167, (1997).

[7] J. McCarthy and P. Hayes, 'Some philosophical problems from the standpoint of artificial intelligence', *Machine Intelligence*, **4**, 463–502, (1969).

[8] R. Ramakrishnan, *Database Management Systems*, WCB/McGraw-Hill, 1998.

[9] R. Reiter, 'The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression', in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, 359–380, Academic Press, (1991).

[10] R. Reiter, 'Formalizing database evolution in the situation calculus', in *Proceedeings of the International Conference on Fifth Genearation Computer Systems*, pp. 600–609, (1992).

[11] R. Reiter, 'The projection problem in the situation calculus: a soundness and completeness result, with an application to database updates', in *Proc. First Int. Conf. on AI Planning Systems*, pp. 198–203, (1992).

[12] R. Reiter, *Knowledge in Action: Logical Foundation for Describing and Implementing Dynamical Systems*, Kluwer, 2000. In preparation.

[13] J. D. Ullman, *Principles of Database and Knowledge Base Systems*, volume 1, Computer Science Press, 1988.

tries to perform such regression together in order to avoid generating tables of the same formula twice. In fact, the idea or reusing tables between formulas can be pushed further introducing some form of cashing mechanism for tables, that keeps in memory some of the tables computed according to some specified policy.