Program Updating by Incremental and Answer Subsumption Tabling

Ari Saptawijaya* and Luís Moniz Pereira

Centro de Inteligência Artificial (CENTRIA), Departamento de Informática
Faculdade de Ciências e Tecnologia, Univ. Nova de Lisboa, 2829-516 Caparica, Portugal
ar.saptawijaya@campus.fct.unl.pt, lmp@fct.unl.pt

Abstract. We propose a novel conceptual approach to program updates implementation that exploits two features of tabling in logic programming (in XSB Prolog): incremental and answer subsumption tabling. Our approach, EVOLP/R, is based on the constructs of Evolving Logic Programs (EVOLP), but simplifies it at first by restricting updates to fluents only. Rule updates are nevertheless achieved via the mechanism of rule name fluents, placed in rules' bodies, permitting to turn rules on or off, through assertions or retractions of their corresponding unique name fluents. Incremental tabling of fluents allows to automatically maintain – at engine level – the consistency of program states. Answer subsumption of fluents addresses the frame problem – at engine level – by automatically keeping track of their latest assertion or retraction. The implementation is detailed here to the extent that it may be exported to other logic programming tabling systems.

Keywords: logic program updates, incremental tabling, answer subsumption tabling.

1 Introduction

In this paper we explore the use of state-of-the-art logic programming implementation techniques to exploit their use in addressing a classical non-monotonic reasoning problem, that of logic program updates, with incidence on representing change, i.e. internal or self and external or world changes. Our approach, EVOLP/R, follows the paradigm of Evolving Logic Programs (EVOLP) [1], by adapting its syntax and semantics, but simplifies it at first by restricting updates to fluents only. This restriction nevertheless permits rule updates to take place, as long as we know the rules beforehand, i.e. ones not constructed, learnt, or externally given. To update the program with such known-from-the-start rules, special fluents that serve as names of rules and identify rules uniquely are introduced. Such a rule name fluent is placed in the body of a rule to turn the rule on and off (cf. [2]), this being achieved by asserting or retracting the rule name fluent.

We foster a novel implementation technique to program updates by exploiting Prolog tabling mechanisms, notably two features of XSB Prolog: incremental and answer subsumption tabling. Incremental tabling of fluents allows to automatically maintain the consistency of program states, analogously to assumption based truth-maintenance

* Affiliated with Fakultas Ilmu Komputer at Universitas Indonesia, Depok, Indonesia.

P. Cabalar and T.C. Son (Eds.): LPNMR 2013, LNAI 8148, pp. 479–484, 2013.

system, due to assertion and retraction of fluents. On the other hand, answer subsumption of fluents allows to address the frame problem by automatically keeping track of their latest assertion or retraction, whether obtained as updated facts or concluded by rules. The employment of these tabling features has profound consequences in modeling agents. It permits separating higher-level declarative representation and reasoning, as a mechanism pertinent to agents, from a world's inbuilt reactive laws of operation. The latter, being of no operational concern to the problem representation level, are relegated to engine-level enacted tabling features. EVOLP/R is realized using a program transformation plus a library of system predicates. The transformation adds some extra information, e.g. timestamps, for internal processing. Rule name fluents are system generated and also added in the transform. System predicates are defined to operate on the transform by combining the usage of incremental and answer subsumption tabling.

We describe the constructs of EVOLP/R (Section 2), detail the implementation technique (Section 3), and discuss related work along with concluding remarks (Section 4).

2 The EVOLP/R Language

For convenience, we represent EVOLP/R programs as propositional Horn theories, by simply adapting EVOLP definitions [1]. Let $\mathcal{K}$ be an arbitrary set of propositional variables. We denote $\tilde{\mathcal{K}}$ as the *extension* of $\mathcal{K}$, and is defined as $\tilde{\mathcal{K}} = \{A : A \in \mathcal{K}\} \cup \{\sim A : A \in \mathcal{K}\}$. Atoms $A \in \mathcal{K}$ and $\sim A$ are called *positive fluents* and *negative fluents*, respectively. As in EVOLP, program updates are enacted by having the reserved predicate $assert/1$ in the head of a rule. We define now the EVOLP/R language and program.

Definition 1. *Let $\tilde{\mathcal{K}}$ be the extension of a set $\mathcal{K}$ of propositional variables. The* EVOLP/R language $\mathcal{L}$ *is defined inductively as follows:*

1. *All propositional atoms in $\tilde{\mathcal{K}}$ are propositional atoms in $\mathcal{L}$.*
2. *If A is a propositional atom in $\mathcal{L}$, then $assert(A)$ is a propositional atom in $\mathcal{L}$.*
3. *If A is a propositional atom in $\mathcal{L}$, then $\sim assert(A)$ is a propositional atom in $\mathcal{L}$.*
4. *If A_0 is a propositional atom in $\mathcal{L}$ and $A_1, \ldots, A_n$, with $n \geq 0$, are literals in $\mathcal{L}$ (i.e. a propositional atom A, or its default negation $not\ A$), then $A_0 \leftarrow A_1, \ldots, A_n$ is a rule in $\mathcal{L}$.*
5. *Nothing else is a propositional atom in $\mathcal{L}$.*

An EVOLP/R program *over a language $\mathcal{L}$ is a (possibly infinite) set of rules in $\mathcal{L}$.*

We extend the notion of positive and negative fluents in $\tilde{\mathcal{K}}$ to propositional atoms A and $\sim A$ in $\mathcal{L}$, respectively. They are said to be *complement* each other. When it is clear from the context, we refer both of them as fluents. Retraction of fluent A (or $\sim A$), making it false, is achieved by asserting its complement $\sim A$ (or A, respectively). I.e., no reserved predicate for retraction is needed. Non-monotonicity of a fluent can thus be admitted by asserting its complement, so as to let the latter supervene the former. Observe that the syntax permits embedded assertions of literals, e.g. $assert(assert(a))$, $\sim assert(assert(a))$; the latter being the complement of the former.

By Definition 1, EVOLP/R programs are not generalized logic programs (like in EVOLP), but they nevertheless permit negative fluents in the rules' heads. Indeed, one

may view negative fluents as explicit negations, and due to the coherence principle [3], that explicit negation entails default negation, negative fluents obey the principle. Therefore, the two forms of rules' heads, i.e. $assert(not\ A)$ in EVOLP and $assert(\sim A)$ in EVOLP/R, can be treated equivalently. This justification allows the semantics of EVOLP/R to be safely based on that of EVOLP, as long as the paraconsistency of simultaneously having A and $\sim A$ is duly detected and handled, say with integrity constraints or preferences. Note that EVOLP/R restricts updates to fluents only. Nevertheless, rule updates (like in EVOLP) can be achieved, via the mechanism of rule name fluents, placed in rules' bodies, allowing to turn rules on or off, through assertions or retractions of their corresponding unique name fluents.

Like EVOLP, besides the self-evolution of a program, EVOLP/R also allows influence from the outside, either as an observation of fluents that are perceived at some state, or assertion orders of fluents on the evolving program. Different from EVOLP, the outside influence in EVOLP/R, referred as *external updates*, persist by inertia as long as they do not conflict with the more recent values for them. Nevertheless, we may easily define external updates that do not persist by inertia, called *events* in EVOLP, by defining for every atomic event E the rule: $assert(\sim E) \leftarrow E$, i.e. if event E is imposed at some state i, then it is no longer assumed from the next state, i.e. $(i + 1)$, onwards. In other words, E holds at state i only.

3 Implementing EVOLP/R in Tabled Logic Programming

Tabling in logic programming affords reuse of solutions, rather than recomputing them, by maintaining subgoals and their answers (obtained in query evaluation) in a table. In implementing EVOLP/R, we exploit in combination two features of tabling in XSB Prolog [4]: (1) Incremental tabling, which ensures the consistency of answers in tables with all dynamic facts and rules upon which the tables depend, and (2) Answer subsumption, which allows tables to retain only answers that subsume others with respect to some partial order relation. The reader is referred to [5] for the definitions, options, examples and details of both features.

The EVOLP/R implementation consists of a compiled program transformation plus a library of system predicates. The transformation adds information to program clauses: (1) *Timestamp* includes two extra arguments of fluents, i.e. *holds time* (the time when a fluent is true) and *query time* (the time when it is queried), (2) *Rule name* as a special fluent $\$rule(p/n, id_i)$, which identifies rule of predicate p with arity n by its unique name identity id_i, and is introduced in its body, for checking that the rule still holds.

Transformation. Example 1 illustrates the transformation technique and how the extra information figures in the transform (predicates $\$rule$ and $assert$ are written as $\$r$ and as, respectively). In EVOLP/R, the initial timestamp is set at 1, when a program is inserted. Fluent predicates can be defined as facts (extensional) or by rules (intensional).

Extensional fluent instances, like a, are translated into a rule which inertially constrains its validity from its holds time up to query time Q. In Example 1, a holds at the initial time 1. This validity may become superseded by that of the fluent's complement. For rule regulated intensional fluent instances, like b and $as(\sim a)$, unique rule name fluents are introduced and translated just like for extensional fluents (lines 2, 4, 6).

Line 3 shows the translation of rule $b \leftarrow a$. The extra arguments in its head are holds time H of fluent b and the query time Q. Calls to the goals in the body are translated into calls to the system predicate $holds/3$ (defined later). In the transform of $b \leftarrow a$ (line 3), the first goal in its body verifies whether the unique rule name fluent $\$r(b/0, id_1)$ holds within query time Q, in which case its latest holds time (i.e. the latest time up to Q this rule was turned on) H_r is returned. The next goal verifies whether a holds at Q by returning its latest holds time H_a. The validity of b at Q, with its holds time H ($\leq Q$), is thus obtained from the maximum of H_r and H_a (i.e. H is determined by which inertial fluent in its body holds latest), via $max/2$ system predicate.

Rule $as(\sim a) \leftarrow b$ is transformed into two rules: the transform in line 5 is similar to that of rule $b \leftarrow a$, whereas the one in line 7 is derived as the effect of asserting $\sim a$. I.e., the validity of $\sim a$, being queried at time Q, depends on the latest time when its rule was turned on (H_r in 1^{st} goal in the body) and when $as(\sim a)$ took place (H_{as} in 4^{th} goal in the body). The latter goal is considered at a query time Q_{as}, where $1 \leq Q_{as} \leq Q - 1$ (generated recursively via $gen/2$ system predicate), i.e. existential H_{as} is obtained by querying at a time point Q_{as} within $Q - 1$, just before $\sim a$ is queried (at Q). The holds time H ($\leq Q$) of $\sim a$ is thus determined, via $max/2$, between H_r and $H_{as} + 1$ (rather than H_{as}, because $\sim a$ is actually asserted one time step from the time $as(\sim a)$ holds).

Example 1. Program: $a.$ $b \leftarrow a.$ $as(\sim a) \leftarrow b.$ transforms into:

1. $a(1, Q)$ $\leftarrow 1 \leq Q.$
2. $\$r(b/0, id_1, 1, Q)$ $\leftarrow 1 \leq Q.$
3. $b(H, Q)$ $\leftarrow holds(\$r(b/0, id_1), H_r, Q),\ holds(a, H_a, Q),$ $max([H_r, H_a], H),\ H \leq Q.$
4. $\$r(as(\sim a/0), id_1, 1, Q)$ $\leftarrow 1 \leq Q.$
5. $as(\sim a, H, Q)$ $\leftarrow holds(\$r(as(\sim a/0), id_1), H_r, Q),\ holds(b, H_b, Q),$ $max([H_r, H_b], H),\ H \leq Q.$
6. $\$r(\sim a/0, id_1, 1, Q)$ $\leftarrow 1 \leq Q.$
7. $\sim a(H, Q)$ $\leftarrow holds(\$r(\sim a/0, id_1), H_r, Q),\ Q'$ is $Q - 1,$ $gen(Q_{as}, Q'),\ holds(as(\sim a), H_{as}, Q_{as}),$ H'_a is $H_{as} + 1,\ max([H_r, H'_a], H),\ H \leq Q.$

Since any fluents occurring in the program may be updated, all fluents and their complements should be declared as dynamic and incremental (in order to benefit from incremental tabling), e.g. `:- dynamic a/2,'~a'/2 as incremental`. Their incremental assertions may influence program states, notably the latest time when they are true, which is maintained in conjunction with answer subsumption tabling.

System Predicates. We first introduce predicate $fluent/3$, i.e. given query time Qt, $fluent(F, Ht, Qt)$ looks for (dynamic) definitions of fluent F, and returns the one with the latest holds time Ht. It makes good combined use of tabling features: (1) Since $fluent/3$ aims at returning only the latest holds time of F, $fluent/3$ can be tabled using answer subsumption on its second argument; and (2) Predicate $fluent/3$ depends on dynamic fluent definitions of F, and this dependency indicates that $fluent/3$ can be tabled incrementally, to avoid abolishing the table each time a Prolog assertion is made and then recomputing from scratch. Consequently, predicate $fluent/3$ is declared as `:- table fluent(_,po('>'/2),_) as incremental`. It is defined as:

$$fluent(F, Ht, Qt) \leftarrow extend(F, [Ht, Qt], F'),\ call(F').$$

where $extend(F, Args, F')$ extends the arguments of fluent F with those in list $Args$ to obtain F'. Since $fluent/3$ enjoys incremental and answer subsumption tabling, it cannot also be dynamic [5]; the latter being delegated to F'.

Example 1 describes how predicate $holds(F, Ht, Qt)$ should be interpreted, i.e. it verifies whether fluent F is true in a given query time Qt, in which case its *latest* holds time Ht is returned. It suggests that $holds/3$ can be defined using $fluent/3$, which provides such latest holds time. But additionally, $holds/3$ has to make sure its fluent complement $\sim F$ does not hold after Ht, in which case F will fail to hold. I.e.,

$$holds(F, Ht, Qt) \leftarrow compl(F, F'), fluent(F, Ht, Qt), fluent(F', Ht', Qt),\\ (Ht \neq 0 \rightarrow Ht \geq Ht'\ ;\ fail).$$

where $compl(F, F')$ obtains the fluent complement F' from F. The last goal in the body, i.e. $(Ht \neq 0 \rightarrow Ht \geq Ht'\ ;\ fail)$, specifies the condition for F to successfully hold. Observe that this condition requires every fluent and its complement to be defined at time 0 (zero), i.e. they are set to true in that special (vacuum) moment in time. This aims to prevent $holds/3$ to fail prematurely in calls to $fluent/3$, which may happen when a fluent or its complement is not defined yet. The condition reads quite straightforward, where only positive timestamps are countenanced, i.e. $Ht \neq 0$ (as they reflect actual time after 0 when a fluent is true): F holds lastly at Ht with respect to query time Qt only if Ht is at least the same as the latest holds time Ht' of $\sim F$. Note that the condition also implicitly covers the case when $\sim F$ is never asserted (i.e. $Ht' = 0$). It also allows paraconsistency (in case $Ht = Ht'$), to be dealt by the user as desired.

Example 2. Recall Example 1, which is loaded initially at time 1. It is easy to verify that query $holds(a, H, 1)$ succeeds with $H = 1$, whereas $holds(a, H, 2)$ fails, but $holds(\sim a, H, 2)$ succeeds with $H = 2$; the latter two persist by inertia. Suppose at time 3, an external update $\{a, \sim\$r(b/0, id_1)\}$ is given. Now, $holds(a, H, 3)$ no longer fails, but succeeds with $H = 3$, because $fluent(a, H, 3)$ succeeds, now with $H = 3$ (instead of with $H = 1$), thanks to incremental tabling (triggered by the external update a) and answer subsumption, whereas $fluent(\sim a, H', 3)$ succeeds with $H' = 2$, and $H \geq H'$. Moreover, due to the external update $\sim\$r(b/0, id_1)$, rule $b \leftarrow a$ is turned off at time 3; consequently $holds(b, H, 3)$ fails (so do $holds(as(\sim a), H, 3)$ and $holds(\sim a, H, 4)$). Thus, a continues to hold at time 4, i.e $holds(a, H, 4)$ succeeds with $H = 3$, onwards.

4 Concluding Remarks

We have proposed EVOLP/R as a simplified EVOLP, by restricting updates to fluents only, for the moment. Rule updates can nevertheless be enacted by introducing a unique rule name fluent to each rule, placed in its body, functioning as a switch to turn the rule on and off. We also showed how incremental tabling is useful to facilitate fluent updates incrementally in dynamic environments and evolving systems (in line with the goals of introducing incremental tabling [6]), and in conjunction with answer subsumption, to avoid recursing through the frame axiom but instead allow direct access to the latest time when a fluent is true.

As a distinct but somewhat similar and complementary approach, we should mention the recent Logic-based Production System with abduction [7], and its successive installments [8], aiming at defining a new encompassing logic-based framework for computing, for knowledge representation and reasoning. It relies on the fundamental role of state transition systems in computing, and involving fluent updates by destructive assignment. It is implemented in LPA Prolog [9], but no details are given about it. In future, we intend to learn from their results and evolve EVOLP/R towards enabling their higher level constructs and compare implementations. Their approach differs from ours in that it defines a new language and an operational semantics, rather than taking an existing one, and implements it on a commercial Prolog system with no underlying tabling mechanisms.

It is our purpose to combine EVOLP/R with tabled abduction [10], so as to jointly afford abduction and updating in one integrated XSB system by exploiting its tabling features, and to apply the integrated system to abductive moral reasoning (cf. [11, 12]), with updating and argumentation, as a sequel to our ongoing approach to this type of non-monotonic reasoning.

Acknowledgements. We thank David S. Warren for elucidating features of tabling. AS acknowledges the support of FCT/MEC Portugal, grant SFRH/BD/72795/2010.

References

1. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 50–61. Springer, Heidelberg (2002)
2. Poole, D.L.: A logical framework for default reasoning. Artificial Intelligence 36(1), 27–47 (1988)
3. Alferes, J.J., Pereira, L.M.: Reasoning with Logic Programming. LNCS (LNAI), vol. 1111. Springer, Heidelberg (1996)
4. Swift, T., Warren, D.S.: XSB: Extending Prolog with tabled logic programming. Theory and Practice of Logic Programming 12(1-2), 157–187 (2012)
5. Swift, T., Warren, D.S., Sagonas, K., Freire, J., Rao, P., Cui, B., Johnson, E., de Castro, L., Marques, R.F., Saha, D., Dawson, S., Kifer, M.: The XSB System Version 3.3.x Volume 1: Programmer's Manual (2012)
6. Saha, D.: Incremental Evaluation of Tabled Logic Programs. PhD thesis, SUNY Stony Brook (2006)
7. Kowalski, R., Sadri, F.: Abductive logic programming agents with destructive databases. Annals of Mathematics and Artificial Intelligence 62(1), 129–158 (2011)
8. Kowalski, R., Sadri, F.: Towards a logic-based unifying framework for computing (2013), `http://www.doc.ic.ac.uk/~rak/papers/TUF.pdf`
9. Logic Programming Associates Ltd.: LPA prolog, `http://www.lpa.co.uk/`
10. Saptawijaya, A., Pereira, L.M.: Tabled abduction in logic programs. Accepted as Technical Communication at ICLP 2013 (2013), `http://centria.di.fct.unl.pt/~lmp/publications/online-papers/tabdual_lp.pdf`
11. Pereira, L.M., Saptawijaya, A.: Modelling Morality with Prospective Logic. In: Anderson, M., Anderson, S.L. (eds.) Machine Ethics, pp. 398–421. Cambridge U. P. (2011)
12. Han, T.A., Saptawijaya, A., Pereira, L.M.: Moral reasoning under uncertainty. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 212–227. Springer, Heidelberg (2012)