Forgetting Actions in Domain Descriptions

Esra Erdem
Faculty of Engineering and Natural Sciences
Sabancı University
Istanbul, Turkey

Paolo Ferraris
Department of Computer Sciences
University of Texas at Austin
Austin, TX, USA

Abstract

Forgetting irrelevant/problematic actions in a domain description can be useful in solving reasoning problems, such as query answering, planning, conflict resolution, prediction, postdiction, etc.. Motivated by such applications, we study what forgetting is, how forgetting can be done, and for which applications forgetting can be useful and how, in the context of reasoning about actions. We study these questions in the action language $\mathcal{C}$ (a formalism based on causal explanations), and relate it to forgetting in classical logic and logic programming.

Introduction

For performing reasoning tasks (planning, prediction, query answering, etc.) in an action domain, not all actions of that domain might be necessary. By instructing the reasoning system to forget about these unnecessary/irrelevant actions, without changing the causal relations among fluents, we might obtain solutions using less computational time/space; and, for some problems, the quality of solutions might be better when some actions are forgotten.

Consider for instance a simple planning problem in a variation of the blocks world domain, where an agent can not only move blocks but also paint them: initially Block A and B are on the table, both blocks are red; in the goal state Block A is on Block B. A possible plan for such a problem is

$$\langle\{\mathit{Move}(A,B), \mathit{Paint}(A,Red), \mathit{Paint}(B,Yellow)\}\rangle.$$

Since the problem is about the locations of blocks rather than the color of blocks, the action of painting is not necessary to find a plan for the problem above; indeed, if the action of painting is forgotten then we can obtain a better plan like

$$\langle\{\mathit{Move}(A,B)\}\rangle.$$

In larger domains, forgetting irrelevant actions may improve not only the quality of plans but also the computational efficiency. For instance (Agarwal *et al.* 2005) mentions the presence of 100000 irrelevant actions in a 7-step plan for some web-service composition problem formulated as a planning problem. Here is a small example: Suppose that initially Mary is in New York, and she wants to have a plane ticket to Boston. A possible plan for this problem is

$$\begin{array}{l}\langle\{\mathit{RequestTicket}(\mathit{Boston}), \mathit{RequestTicket}(\mathit{Austin}),\\ \ \mathit{RequestTicket}(\mathit{NewYork}), \mathit{RequestTicket}(\mathit{Miami}),\\ \ \mathit{RequestRoom}(\mathit{NewYork}), \mathit{RequestRoom}(\mathit{Boston}),\\ \ \mathit{RequestRoom}(\mathit{Seattle}), \mathit{RequestRoom}(\mathit{Miami}),\\ \ \mathit{RequestRoom}(\mathit{Austin}), \mathit{RequestRoom}(\mathit{Toronto}),\\ \ \mathit{RequestRoom}(\mathit{Chicago}), \mathit{RequestCar}(\mathit{Toronto}),\\ \ \mathit{RequestCar}(\mathit{Miami}), \mathit{RequestCar}(\mathit{Chicago})\}\rangle.\end{array} \quad (1)$$

On the other hand, if we forget all actions but $\mathit{RequestTicket}(\mathit{Boston})$, a more economic plan can be found:

$$\langle\{\mathit{RequestTicket}(\mathit{Boston})\}\rangle. \quad (2)$$

In addition to obtaining better plans, or improving the computational efficiency, forgetting can be useful for finding "good enough" solutions to some problems, when no solution exists: the idea is to forget the "problematic" part of the domain. For instance such problems can occur in multi-agent coordination: when multiple agents have conflicting preferences/constraints over occurrences of actions, solutions to some reasoning problems can be found only if agents make some compromises. One such compromise is to forget about actions causing conflicts, as discussed in (Lang & Marquis 2002; Lang, Liberatore, & Marquis 2003; Wang, Sattar, & Su 2005).

Motivated by such problems above, the concept of forgetting has been studied in classical logic (Lin & Reiter 1994; Lang, Liberatore, & Marquis 2003), and in logic programming (Wang, Sattar, & Su 2005; Eiter & Wang 2006). In this paper, we study forgetting, in particular, forgetting an action in a domain description, in the framework of action languages (Gelfond & Lifschitz 1998).

Action languages are high-level knowledge representation and reasoning formalisms, based on a theory of causality; the meaning of a domain description in such a language can be represented by a transition diagram—a directed graph whose nodes denote states of the world and edges denote the changes caused by occurrence/nonoccurrence of actions. (See Figure 1 for an example.) Here we consider a fragment of the action language $\mathcal{C}$ (Giunchiglia & Lifschitz 1998), which provides a framework for various reasoning problems (e.g., planning, prediction, postdiction) in the presence of concurrency, nondeterminism, ramifications, etc.

In the following, first we define forgetting an action in a domain description, study its complexity (Proposition 1), and introduce a method to achieve it (Proposition 3). After

Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

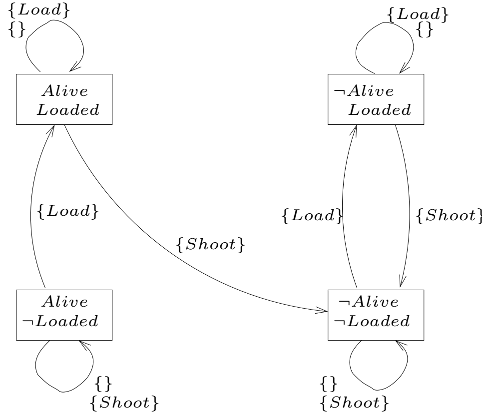

Figure 1: The transition diagram described by (5).

that we present its applications to two domains: planning and resolving conflicts between multiple agents. Finally, we relate our approach, to forgetting in propositional theories (Lin & Reiter 1994) (Proposition 4) and forgetting in logic programming (Wang, Sattar, & Su 2005) (Proposition 5).

Transition Diagrams and Action Descriptions

Our signature consists of propositional symbols of two kinds: *fluent names* and *action names*. A *state* is a set of fluent names. A *transition diagram* is a directed graph whose nodes are states, and whose edges (the *transitions*) are labeled by a set of action names. A transition from state s to state s' with label A is also represented as the tuple $\langle s, A, s'\rangle$. The meaning of $\langle s, A, s'\rangle$ is that the concurrent execution of every action (name) in A (and of no other action) in state s leads to state s'. We agree to denote multiple edges between the same nodes with a single edge having multiple labels. An example of a transition diagram for the shooting domain (Hanks & McDermott 1987) is in Figure 1. The action "wait" of the original description of the shooting domain is represented in the transition diagram by $\{\}$. In the figure, in each node/state, the fluent names that are false are shown also. No edge has label $\{\mathit{Load}, \mathit{Shoot}\}$, i.e., these two actions can't be executed concurrently. Finally, note that effects of actions are deterministic: there are no two edges with the same label leaving the same node.

Action descriptions, described using action languages, are a more compact and natural way of describing transition diagrams. We consider a subset of the action language $\mathcal{C}$ (Giunchiglia & Lifschitz 1998) that consists of two kinds of expressions (called *causal laws*): *static laws* of the form

$$\textbf{caused } L \textbf{ if } G, \qquad (3)$$

where L is a fluent literal (an expression of the form P or $\neg P$, where P is a fluent name) or $\bot$, and G is a propositional combination of fluent names; and *dynamic laws* of the form

$$\textbf{caused } L \textbf{ if } G \textbf{ after } U, \qquad (4)$$

where L and G are as above, and U is a propositional combination of fluent names and action names. An *action description* is a set of causal laws.

Consider, for instance, the action description of the shooting domain, as in (Giunchiglia & Lifschitz 1998):

$$\begin{array}{l} \textbf{caused } \mathit{Loaded} \textbf{ if } \top \textbf{ after } \mathit{Load} \\ \textbf{caused } \neg\mathit{Loaded} \textbf{ if } \top \textbf{ after } \mathit{Shoot} \\ \textbf{caused } \neg\mathit{Alive} \textbf{ if } \top \textbf{ after } \mathit{Loaded} \wedge \mathit{Shoot} \\ \textbf{nonexecutable } \mathit{Load} \wedge \mathit{Shoot} \\ \textbf{inertial } \mathit{Loaded}, \neg\mathit{Loaded}, \mathit{Alive}, \neg\mathit{Alive}. \end{array} \qquad (5)$$

The fourth line is an abbreviation for

$$\textbf{caused } \bot \textbf{ if } \top \textbf{ after } \mathit{Load} \wedge \mathit{Shoot}$$

and the last line stands for four dynamic laws of the form

$$\textbf{caused } X \textbf{ if } X \textbf{ after } X$$

($X \in \{\mathit{Loaded}, \neg\mathit{Loaded}, \mathit{Alive}, \neg\mathit{Alive}\}$). This action description corresponds to the transition diagram in Figure 1.

The semantics of an action description D can be represented by a transition diagram, denoted by $T(D)$, and defined as follows. Let us denote by $I(s)$ the truth assignment that maps only the fluent names that belong to s to true, and by $I(s, A)$ the truth assignment of fluent names and action names that maps only the elements of $s \cup A$ to true. The set $S(D)$ of states in $T(D)$ contains every state s such that $I(s) \models G \supset L$ for every static law (3) in D. Let $D_{\langle s,A,s'\rangle}$ be the set of formulas L of each static law (3) such that $I(s') \models G$, and formulas L of each dynamic law (4) such that $I(s') \models G$ and $I(s, A) \models U$. The set $R(D)$ of transitions in $T(D)$ contains $\langle s, A, s'\rangle$ if $I(s')$ is the only truth assignment that satisfies all formulas in $D_{\langle s,A,s'\rangle}$. (See (Giunchiglia & Lifschitz 1998) for further explanations.)

Forgetting an Action in a Domain Description

We understand forgetting an action name B in a transition diagram as removing B from all edge labels of that transition diagram. For instance, Figure 2 shows the result of forgetting *Load* in the transition system in Figure 1. Based on this understanding, for two action descriptions D and D' and an action name B, we say that D' is *a result of forgetting* B in D if $T(D')$ is the result of forgetting B in $T(D)$. More precisely,

(i) $S(D') = S(D)$;

(ii) $R(D') = \{\langle s, A \setminus \{B\}, s'\rangle | \langle s, A, s'\rangle \in R(D)\}$.

For instance, the following action description is a result of forgetting *Load* in the shooting domain (5):

$$\begin{array}{l} \textbf{caused } \mathit{Loaded} \textbf{ if } \mathit{Loaded} \textbf{ after } \neg\mathit{Shoot} \\ \textbf{caused } \neg\mathit{Loaded} \textbf{ if } \neg\mathit{Loaded} \textbf{ after } \mathit{Shoot} \\ \textbf{caused } \neg\mathit{Alive} \textbf{ if } \top \textbf{ after } \mathit{Loaded} \wedge \mathit{Shoot} \\ \textbf{inertial } \neg\mathit{Loaded}, \mathit{Alive}, \neg\mathit{Alive} \end{array} \qquad (6)$$

whose transition diagram is shown in Figure 2. Using this domain description we can solve reasoning problems according to which *Load* is irrelevant. Consider for instance the following query: from a state where the turkey is alive, is it possible to reach a state where the turkey is still alive after shooting twice and waiting? To answer this query, one can forget about *Load*, and use the reasoning system CCALC (Giunchiglia *et al.* 2004) with (6).

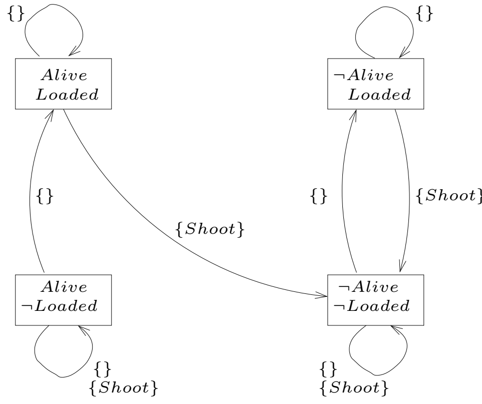

Figure 2: The transition diagram described by (6).

Notice that, according to (6), doing nothing at a state where $\neg$*Loaded* has nondeterministic effects: either *Loaded* becomes true or not. This example shows that forgetting may transform a deterministic domain description into a nondeterministic one. In such cases, depending on the reasoning problem studied, one must ensure that the method used for solving the problem with the original deterministic description may not be suitable after forgetting. For instance, nondeterminism in (6) has not prevented us from answering the query above correctly using CCALC. On the other hand, if we were to solve a planning problem whose solution involves nondeterministic actions, we would need, instead of CCALC, a conformant planner (e.g., CPLAN (Ferraris & Giunchiglia 2000)) to compute a valid plan. (One can use CCALC to compute a "possible" plan, but then needs to monitor its execution to ensure that the goal is achieved.)

It is clear that forgetting preserves equivalence between action descriptions: for action descriptions D_1 and D_2 such that $T(D_1) = T(D_2)$, and an action name B, if D'_1 is a result of forgetting B in D_1 and D'_2 is a result of forgetting B in D_2 then $T(D'_1) = T(D'_2)$.

It is also clear that the order of forgetting two (or more) action names in an action description is irrelevant: the corresponding transition diagrams would be identical.

The problem of checking whether a given description is a result of forgetting is a hard problem:

Proposition 1 *Checking whether a given description D' is a result of forgetting B in D is coNP-complete.*

The proof follows from the fact that checking each condition of the definition of a result of forgetting is coNP-complete.

A Method for Forgetting an Action

One might think that replacing B by $\bot$ is a suitable method for forgetting an action name B in a domain description D; however, this methods doesn't always work (consider forgetting all actions in the query answering example above). Considering the nonmonotonic nature of action languages, we need a more sophisticated method for forgetting. Below we present one.

For simplicity, due to the proposition below, we assume that in every static law (3) in D, L is $\bot$.

Proposition 2 *For an action description D, and a static law (3), $D \cup \{$**caused** L **if** $G\}$ and $D \cup \{$**caused** L **if** G **after** $\top$, **caused** $\bot$ **if** $\neg L \wedge G\}$ describe the same transition diagram.*

This proposition is similar to (Erdoğan & Lifschitz 2006, Proposition 5). Its proof based on the concept of strong equivalence of causal theories (Turner 2004), and uses (Erdoğan & Lifschitz 2006, Lemma 7).

We need the following definitions and notation to describe our method for forgetting. An *n-fluent* is an expression of the form $n(P)$ where P is a fluent name. For a formula F, we denote by $n(F)$ the formula obtained from F by replacing every fluent name P by $n(P)$. Intuitively, $n(P)$ refers to "P at the next state".

For an action name B, let

$$Y_B = \bigwedge_{(\textbf{caused } \bot \textbf{ if } G \textbf{ after } U) \in D, B \text{ occurs in } U} \neg(n(G) \wedge U).$$

Intuitively, Y_B describes the executability conditions for B.

For a fluent literal L, let

$$Z_L = \bigvee_{(\textbf{caused } L \textbf{ if } G \textbf{ after } U) \in D} (n(G) \wedge U).$$

Intuitively, Z_L describes the changes and the conditions that cause L to be true.

Let $\mathbf{F}_B$ be the set of fluent names P such that there exists a rule (4) in D with B that occurs in U, and either $L = P$ or $L = \neg P$.

For a fluent literal L and an action name B, let

$$\gamma_{L,B} = \neg Z_{\overline{L}} \wedge Y_B \wedge \bigwedge_{P \in \mathbf{F}_B \setminus \{L, \overline{L}\}} \mathbf{Z}_P$$

where $\overline{L}$ stands for the literal complementary to L, and

$$\mathbf{Z}_P = (n(P) \wedge \neg Z_{\neg P} \wedge Z_P) \vee (n(\neg P) \wedge \neg Z_P \wedge Z_{\neg P}).$$

Intuitively, $\gamma_{L,B}$ describes the changes and the conditions when L is caused to be true.

For a formula F (possibly with n-fluents), by $F[B/G]$ we denote the formula obtained from F by replacing each occurrence of B by formula G. By $\Sigma(F)$ we denote an arbitrary set of pairs (E^o, E^n) of formulas without n-fluents such that

$$\bigvee_{(E^o, E^n) \in \Sigma(F)} (n(E^n) \wedge E^o)$$

is equivalent to F. Set $\Sigma(F)$ can be computed, for instance, by rewriting F in disjunctive normal form (DNF).

Now we can describe how to obtain from an action description D a result of forgetting an action name B in D:

1. Drop all causal laws

   caused $\bot$ **if** G **after** U

   where B appears in U;

2. Replace each dynamic law

   caused L **if** G **after** U

Table 1: Computation of γ' for $B = Load$ for Example 1

	$[B/\top]$	$[B/\bot]$
Y_B	$\neg Shoot$	$\top$
Z_{Loaded}	$\top$	$n(Loaded) \wedge Loaded$
$Z_{\neg Loaded}$	$Shoot \vee (\neg n(Loaded) \wedge \neg Loaded)$	
$\mathbf{Z}_{Loaded}$	$n(Loaded) \wedge \neg Shoot$	$n(Loaded) \equiv (\neg Shoot \wedge Loaded)$
$\gamma'_{Loaded,B}$	$\neg Shoot$	$\neg Shoot \wedge Loaded$
$\gamma'_{\neg Loaded,B}$	$\bot$	$Shoot \vee \neg Loaded$

such that the fluent name in L belongs to $\mathbf{F}_B$, with the set of causal laws

$$\textbf{caused } L \textbf{ if } G \wedge L \wedge E^n \textbf{ after } U[B/\top] \wedge E^o \quad (7)$$

such that $(E^o, E^n) \in \Sigma(\gamma_{L,B}[B/\top])$, and the set of causal laws

$$\textbf{caused } L \textbf{ if } G \wedge L \wedge E^n \textbf{ after } U[B/\bot] \wedge E^o \quad (8)$$

such that $(E^o, E^n) \in \Sigma(\gamma_{L,B}[B/\bot])$;

3. If $\mathbf{F}_B$ is empty, add the constraints

$$\textbf{caused } \bot \textbf{ if } E^n \textbf{ after } E^o \quad (9)$$

for every $(E^o, E^n) \in \Sigma(\neg(Y_B[B/\top] \vee Y_B[B/\bot]))$.

Note that Step 3 applies iff no replacements happen at Step 2. It is also clear that rules (8) can be dropped when B is a conjunctive term of U, and (7) can be dropped when $\neg B$ is a conjunctive term of U.

The method presented above is polynomial, given that $\gamma_{L,B}[B/\top]$ and $\gamma_{L,B}[B/\bot]$ are in DNF. Transformations of formulas into DNF may lead to exponential growth.

Let us denote by $forget(D; B)$ the action description obtained from D and B by the method above. The following proposition states the correctness of the method above.

Proposition 3 *For an action description D and an action name B, $forget(D; B)$) is a result of forgetting B in D.*

One of the most difficult steps in computing $forget(D; B)$ is in computing all the $\gamma_{L,B}[B/\top]$ and $\gamma_{L,B}[B/\bot]$. It is convenient to immediately substitute $\top$ and $\bot$ for B in Y_B and all Z_L to make computation simpler. Furthermore, (8) and (9) both contain L in the "if" part. If P is the fluent literal in L, it is not hard to show that it is safe to replace in both $\gamma_{L,B}[B/\top]$ and $\gamma_{L,B}[B/\bot]$ each occurrence of $n(P)$ with the truth value that makes L true. We will call those formulas $\gamma'_{L,B}[B/\top]$ and $\gamma'_{L,B}[B/\bot]$.

Example 1 Take D to be (5) and $B = Load$. In this case, $\mathbf{F}_B = \{Loaded\}$. The computation of $\gamma'_{L,B}[B/\top]$ and $\gamma'_{L,B}[B/\bot]$ is presented in Table 1. The first step drops the nonexecutability causal law in D. The second step replaces the first two laws of (5) by

$$\textbf{caused } Loaded \textbf{ if } Loaded \textbf{ after } \neg Shoot \quad (10)$$

and

caused $\neg Loaded$ **if** $\neg Loaded$ **after** $Shoot$
inertial $\neg Loaded$

respectively; and **inertial** *Loaded* by

caused $Loaded$ **if** $Loaded$ **after** $Loaded \wedge \neg Shoot$

which is "weaker" than (10) and thus can be dropped. Therefore, $forget(D, B)$ is (6).

Two Applications of Forgetting

Forgetting an action can be applied to various reasoning problems; here we present only two of them.

Forgetting irrelevant actions in planning

An action B is *irrelevant* with respect to a planning problem P and an action description D, if there is a solution to P that does not involve B. (See (Lifschitz & Ren 2004; McIlraith & Fadel 2002; Agarwal *et al.* 2005; Nebel, Dimopoulos, & Koehler 1997) for stronger definitions of irrelevancy.) Irrelevant actions often occur in domains with parallel actions. Consider a simple travelling domain, with actions of buying an airline ticket, renting a car, and reserving a hotel room; consider only seven cities, including New York and Boston. Such a domain can be described as in Figure 3. Consider also the following planning problem in this domain: initially Mary does not have a reserved ticket, room, or car in any city; she wants to have a ticket to Boston. In this travelling domain, all actions except $RequestTicket(Boston)$ are irrelevant with respect to this planning problem. Therefore, before computing a plan for this problem, one can forget all these irrelevant actions by the method above, and obtain a smaller action description, like the one in Figure 4. Indeed, the description in Figure 3 is compiled by CCALC into a propositional theory with 63 atoms and 84 clauses, whereas the description in Figure 4 is compiled by CCALC into a propositional theory with 43 atoms and 24 clauses. CCALC (by calling RELSAT) computes plan (1) with the domain description in Figure 3, whereas it computes a more economic plan, (2), with the domain description in Figure 4. If we increase the number of cities to 40, then CCALC compiles the original description into a theory with 360 atoms and 480 clauses, and finds a plan of length 72, whereas it compiles the description after forgetting into a theory with 241 atoms and 123 clauses, and computes plan (2). In this example, by means of forgetting, not only is a better plan computed (with fewer actions), but also the search space is reduced. The following property allows us to compute a valid plan for a planning problem, using CCALC, after forgetting irrelevant actions in a deterministic action description: A goal state of P is reachable from the initial state of P by executing a sequence $A_1, \ldots, A_n$ of actions, according to $forget(D; B)$, iff there is a valid plan $A'_1, \ldots, A'_n$ for P, according to D, such that $A_i = A'_i \setminus \{B\}$.

Although the process of first identifying irrelevant actions, next forgetting them, and then planning, is in theory as hard as planning without forgetting, it has in practice advantages as hinted by the examples above: better plans, less

Notation: l, l' range over cities, and f ranges over fluent constants.

caused $Ticket(l)$ **if** $\top$ **after** $RequestTicket(l)$
caused $Car(l)$ **if** $\top$ **after** $RequestCar(l)$
caused $Room(l)$ **if** $\top$ **after** $RequestRoom(l)$
nonexecutable $RequestTicket(l)$ **if** $Ticket(l)$
nonexecutable $RequestRoom(l)$ **if** $Room(l)$
nonexecutable $RequestCar(l)$ **if** $Car(l)$
inertial $f, \neg f$.

Figure 3: A simple travelling domain in $\mathcal{C}$.

Notation: l, l' range over cities, and f ranges over fluent constants.

caused $Ticket(Boston)$ **if** $\top$ **after** $RequestTicket(Boston)$
caused $Ticket(l)$ **if** $Ticket(l)$ **after** $\neg Ticket(l)$ $\quad (l \neq Boston)$
caused $Car(l)$ **if** $Car(l)$ **after** $\neg Car(l)$
caused $Room(l)$ **if** $Room(l)$ **after** $\neg Room(l)$
nonexecutable $RequestTicket(Boston)$ **if** $Ticket(Boston)$
inertial $f, \neg f$.

Figure 4: The travelling domain of Figure 3, after forgetting all actions but *RequestTicket*(*Boston*).

computation time/space.

In the above, we have considered one possible definition of irrelevancy to show one way of applying forgetting in planning. Different definitions of irrelevancy can be used for different applications of forgetting in planning. For instance, assume that Mary wants to go from Austin to Albany. At the moment she has to reserve flight tickets, she may decide to forget about driving, and solve the simpler planning problem of reserving tickets. Here driving and flying are not irrelevant to the original planning problem, in the sense of the definition above, but they are irrelevant to the simpler problem. This example suggests defining irrelevancy considering also how a plan is computed (e.g., in this case, hierarchically). Now consider a planning problem, involving multiple agents, so that each agent is allowed to or capable of performing some actions only. Then Agent 1 may forget the actions that are not of his responsibility, find a partial plan considering only the actions that he is allowed to execute, and communicate the partial plan to Agent 2. Given this partial plan, Agent 2 can compute a plan considering its own actions, and pass the partial plan to next agent, and so on. This example suggests defining irrelevancy of actions considering also how a plan is executed (e.g., in this case, by splitting the "workload" among agents).

As an alternative to forgetting, one can try to minimize the number of actions with cardinality constraints, like in (Büttner & Rintanen 2005). For instance, in the travelling problem above, instead of forgetting, if we constrain the executability of concurrent actions to 2, then CCALC compiles the description into a theory with 63 atoms, 1414 clauses, and finds the plan $\langle\{RequestRoom(NewYork), RequestTicket(Boston)\}\rangle$. If we don't allow concurrence of actions then CALC compiles the description into a theory with 63 atoms, 294 clauses, and finds (2). As we can see from this example, both approaches have advantages over each other: for minimizing the number of actions, one does not need to find out which actions are irrelevant; with forgetting, reasoning systems work with smaller theories.

Resolving conflicts among multiple agents

Suppose that each agent's constraints/preferences over occurrences of actions is described by a set of causal laws. For instance, we can describe by the following causal laws that Lisa does not want to drive to a place that is far, and does not want to fly to a place that is close, that she finds driving enjoyable, and that she wants to fly if she has tickets:

nonexecutable $Drive(l) \wedge In(l') \wedge Far(l', l)$
nonexecutable $Fly(l) \wedge In(l') \wedge \neg Far(l', l)$
caused $EnjoyableRide(l)$ **if** $\top$ **after** $Drive(l)$
caused $\bot$ **if** $\top$ **after** $Ticket(l) \wedge \neg Fly(l)$.

We say that a finite set Q of causal laws *conflicts* with respect to a planning problem P, if the planning problem obtained from P by adding Q to the the domain description of P has no solutions. With multiple agents, we suppose that each agent's constraints, denoted by a set of causal laws, do not conflict with respect to the domain description and the planning problem, but their union might. For instance, suppose that John and Lisa want to join Mary, travelling from Seattle to Boston. Lisa has the constraints above. Mary does not want to drive or fly unless they have a reserved room, she does not enjoy driving, and she thinks to get somewhere else one should have a ticket; and John wants to drive, and does not want to request other services while requesting a ticket. Suppose that Q_{Lisa}, Q_{Mary} and Q_{John} denote three sets of causal laws describing these constraints respectively, and that P is the following planning problem: Initially, everyone is in Seattle, and no one has a booked ticket, car, or room; the goal is to get to Boston. Suppose that we are given a travelling domain description D that extends the one in Figure 3 with causal laws describing effects of driving and flying. Although each Q_{Lisa}, Q_{Mary}, and Q_{John} does not conflict with D and P, their union does: Mary wants to fly to Seattle since it is far, but John wants to drive. Therefore, a travel plan cannot be found.

When such conflicts occur, one way to find a solution to the given planning problem is (as suggested by (Eiter & Wang 2006, Definition 4)) to forget in each Q_i actions that cause a conflict. In the example, one such action is driving; if John, Lisa, Mary decide to weaken their constraints by forgetting about driving, a plan for P can be computed by CCALC with $D \cup forget(D; Drive) \cup forget(Q_{Lisa}; Drive) \cup forget(Q_{Mary}; Drive) \cup forget(Q_{John}; Drive)$. However, such a plan may involve other irrelevant actions, e.g., *RequestCar*(*Miami*). In fact, CCALC finds a plan of length 3 involving 14 actions. If all the irrelevant actions are forgotten also, then CCALC finds the following plan: $\langle\{RequestRoom(Boston)\}, \{RequestTicket(Boston)\}, \{Fly(Boston)\}\rangle$. This example shows the use of forgetting in conflict resolution, by means of finding a set of actions that cause conflicts. Instead of identifying such conflict sets, one might consider priorities of preferences/constraints of multiple agents over occurrences of actions, and decide to forget some of them with lower priorities, as in (Lang, Liberatore, & Marquis 2003). Alternatively, the domain description can be updated iteratively by the method of (Eiter *et al.* 2005).

Forgetting in Other Formalisms

The result of forgetting an atom p in a propositional formula F (denoted $forget_P(F; p)$) is defined in (Lin & Reiter 1994) as $F[p/\bot] \vee F[p/\top]$. An action description D can be transformed into a propositional formula $comp(D)$ by the process of literal completion, as described in (Giunchiglia & Lifschitz 1998) . These results allow us to relate forgetting in action languages to forgetting in propositional logic:

Proposition 4 *For an action description D and an action name B, $forget_P(comp(D); B) \equiv comp(forget(D; B))$.*

A result of forgetting a literal L in a logic program P (denoted $\mathit{forget}_{lp}(P; L)$) is defined in (Wang, Sattar, & Su 2005) as a program whose answer sets are the minimal sets in $\{X \setminus \{L\} \mid X \text{ is an answer set for } P\}$. On the other hand, an action description D can be translated into a logic program $\mathit{lp}(D)$ as described in (Lifschitz & Turner 1999). The forgetting in action descriptions can be formulated in terms of forgetting in logic programs as follows:

Proposition 5 *For an action description D and an action name B, $\mathit{forget}_{lp}(\mathit{forget}_{lp}(\mathit{lp}(D); B); \neg B)$ and $\mathit{lp}(\mathit{forget}(D; B))$ have the same answer sets.*

Note that since forgetting in logic programming is defined for literals, both B and $\neg B$ are forgotten on the left hand-side. The order of forgetting them is actually irrelevant. It is also interesting to notice that forgetting both B and $\neg B$ in a program P whose answer sets are complete (such as $\mathit{lp}(D)$) brings forth a logic program whose answer sets are the result of removing these literals from all the answer sets for P. This doesn't hold for arbitrary P though. For instance, the answer sets for $P = \{p \leftarrow \mathit{not}\ q, q \leftarrow \mathit{not}\ p\}$ are $\{p\}$ and $\{q\}$, and we would expect the answer sets for $\mathit{forget}_{lp}(P; p)$ to be $\emptyset$ and $\{q\}$; however, the only answer set is $\emptyset$.

Forgetting in propositional formulas consists of simple, local transformations of the same formula. The reason why the methods of forgetting in logic programs and in action languages are not as simple can be explained in view of the following two facts. First of all, logic programs and action languages are syntactically more "rigid": for instance, there is no concept of a disjunction of two action descriptions. Second, both logic programs and the underlying logic of action languages are nonmonotonic logics.

The method of forgetting a literal L in a logic program P, described in (Wang, Sattar, & Su 2005), computes $\mathit{forget}_{lp}(P; L)$ from the answer sets for P, and not by syntactical transformations of P. Consequently, the two logic programs may look very different from each other, and the computation of $\mathit{forget}_{lp}(P; L)$ involves every rule of P. On the other hand, our method of forgetting in action languages is purely syntactical, and it generally doesn't involve the whole action description: the only laws that are considered and modified in the process of forgetting an action B are those that contain B and those that contain, in the "if" part, a fluent (possibly negated) that is directly influenced by B by some causal law. This may be very important in large domain descriptions.

Conclusion

We have defined forgetting an action in a domain description, in the framework of the action language $\mathcal{C}$, introduced a method to achieve it, illustrated two possible applications of it (i.e., planning, and resolving conflicts among multiple agents), and compared our notion of forgetting to other notions in related formalisms (i.e., classical logic, and logic programming). One question left as a future work is forgetting a fluent in an action description; note that this may involve collapsing some nodes of the transition diagram. This paper has been about "what forgetting is", "how forgetting can be done", and "for which applications forgetting can be useful", in the context of reasoning about actions; a more detailed study of "what to forget" (e.g., irrelevant actions, conflict sets) is also left as a future work.

Acknowledgments Thanks to Vladimir Lifschitz, and anonymous reviewers for useful comments.

References

Agarwal, V.; Chafle, G.; Dasgupta, K.; Karnik, N.; Kumar, A.; Mittal, S.; and Srivastava, B. 2005. Synthy: A system for end to end composition of web services. *Journal of Web Semantics* 3(4).

Büttner, M., and Rintanen, J. 2005. Improving parallel planning with constraints on the number of operators. In *Proc. of ICAPS*, 292–299.

Eiter, T., and Wang, K. 2006. Forgetting and conflict resolving in disjunctive logic programming. In *Proc. of AAAI*.

Eiter, T.; Erdem, E.; Fink, M.; and Senko, J. 2005. Updating action domain descriptions. In *Proc. of IJCAI*.

Erdoğan, S., and Lifschitz, V. 2006. Actions as special cases. In *Proc. of KR*.

Ferraris, P., and Giunchiglia, E. 2000. Planning as satisfiability in nondeterministic domains. In *Proc. of AAAI*.

Gelfond, M., and Lifschitz, V. 1998. Action languages. *Electronic Transactions on AI* 3:195–210.

Giunchiglia, E., and Lifschitz, V. 1998. An action language based on causal explanation: Preliminary report. In *Proc. of AAAI*, 623–630.

Giunchiglia, E.; Lee, J.; Lifschitz, V.; McCain, N.; and Turner, H. 2004. Nonmonotonic causal theories. *AIJ* 153:49–104.

Hanks, S., and McDermott, D. 1987. Nonmonotonic logic and temporal projection. *AIJ* 33(3):379–412.

Lang, J., and Marquis, P. 2002. Resolving inconsistensies by variable forgetting. In *Proc. of KR*, 239–250.

Lang, J.; Liberatore, P.; and Marquis, P. 2003. Propositional independence: Formula-variable independence and forgetting. *JAIR* 18:391–443.

Lifschitz, V., and Ren, W. 2004. Irrelevant actions in plan generation (extended abstract). In *Proc. of the Ninth Ibero-American Workshops on AI*, 71–78.

Lifschitz, V., and Turner, H. 1999. Representing transition systems by logic programs. In *Proc. of LPNMR*, 92–106.

Lin, F., and Reiter, R. 1994. Forget it! In *Working Notes of the AAAI Fall Symposium on Relevance*.

McIlraith, S., and Fadel, R. 2002. Planning with complex actions. In *Proc. of NMR*, 356–364.

Nebel, B.; Dimopoulos, Y.; and Koehler, J. 1997. Ignoring irrelevant facts and operators in plan generation. In *Proc. of ECP*, 338–350.

Turner, H. 2004. Strong equivalence for causal theories. In *Proc. of LPNMR*, 289–301.

Wang, K.; Sattar, A.; and Su, K. 2005. A theory of forgetting in logic programming. In *Proc. of AAAI*, 682–687.