An agent architecture that combines backward and forward reasoning

R.A. Kowalski, F. Sadri, F.Toni
Imperial College of Science, Technology and Medicine,
Department of Computing
180 Queen's Gate, London SW7 2BZ, UK
{rak,fs,ft}@doc.ic.ac.uk

Abstract. We present a proof procedure that combines backward reasoning with logic programs and forward reasoning with integrity constraints. We illustrate the use of the proof procedure as the inference engine underlying agents. Roughly speaking, the backward reasoning component of the proof procedure is responsible for the deliberative behaviour of agents, whereas the forward reasoning component is responsible for their reactive behaviour.

1 Introduction

Recent years have seen a shift in the treatment of agents in artificial intelligence. In conventional approaches, intelligent agents reason and plan within a closed environment, producing a complete plan before they start executing it. In more recent, and arguably more realistic approaches, intelligent agents reason and plan while reacting to and interacting with the external environment (e.g., see [15]). The agent architecture of [6–8] combines rational, deliberative behaviour and reactivity to the external environment. At the heart of this approach lies a proof procedure, which is executed within an observe-think-act cycle, and is responsible for both the thinking and the reacting components. In this paper we present the proof procedure and discuss the benefits of the combination of forward and backward reasoning that it allows. The proof procedure we present is a version of the procedure of [10], which, in turn, is strongly related to the procedures of [4, 2, 3].

The proof procedures of [4, 2, 3] compute "abductive explanations" for given queries, understood as "observations", with respect to an underlying "abductive logic program". An *abductive logic program* consists of a logic program, a pool of candidate *hypotheses*, i.e. atoms whose predicates (called the *abducible* predicates) are amongst the undefined predicates in the program, and a set of *integrity constraints*, i.e. sentences that must be satisfied by every chosen set of hypotheses in the given pool. *(Abductive) explanations* for queries (or observations) are sets of hypotheses Δ such that the logic program together with Δ "entails" the given queries, and such that the logic program together with Δ "satisfies" the integrity constraints. Various notions of entailment and satisfaction can be adopted (see [5]).

The proof procedure of [10] (henceforth referred to as SLP) is a generalisation of [4]. SLPcan be used to compute explanations for observations in abductive logic programming as well as to optimise queries in deductive databases (*semantic query optimisation*) and to compute answers to queries in *constraint logic programming*. SLP uses logic programs written as sets of iff-definitions (rather than if-clauses as in conventional logic programming) and integrity constraints written as (closed) implications. Iff-definitions are used *backward* to *unfold* atoms in queries replacing the atoms by their definition. Integrity constraints are used *forward* to *propagate* atoms in queries, adding additional (reduced) integrity constraints to the queries. The proof procedure employs additional operations to manipulate equality and to perform logical simplification. Operations are seen as rewrite rule and are used to reduce a given query to a disjunction of answers. Answers consist of conjunctions of *suspended atoms*, i.e. atoms whose definition is not available (hypotheses) or which have multiple definitions. Therefore, suspension is used to control (explosive) non-determinism. For example, SLP assumes iff-definitions for the predicate $>$ such that an atom $X > 10$ is suspended and returned as part of an answer, rather than being replaced by (infinitely many) answers $X = 11$, $X = 12$, $\ldots$.

The remainder of the paper is organised as follows. Section 2 defines some preliminary notions. Section 3 gives the proof procedure. Section 4 revises the agent architecture of [6–8]. Section 5 gives examples of the use of the proof procedure within the chosen agent architecture. Section 6 concludes and discusses future work.

2 Preliminaries

An (if-)**clause** is a sentence of the form

$A \leftarrow L_1, \ldots, L_n,$

with A atom, $L_1, \ldots, L_n$ (positive or negative) literals, and $n \geq 0$. All variables in A, $L_1, \ldots, L_n$ are implicitly universally quantified, with scope the entire clause. If $A = p(t)$, for some vector of terms t, the clause is said to *define* p.

A **logic program** is a set of clauses.

An **iff-definition** is a sentence of the form

$A \leftrightarrow D_1 \vee \ldots \vee D_k$

with A atom, $D_1, \ldots, D_k$ conjunctions of (positive or negative) literals, and $k \geq 0$. All variables in A are implicitly universally quantified, with scope the entire iff-definition. All variables in $D_1, \ldots, D_k$ but not in A are implicitly existentially quantified, with scope the disjunct in which they occur. A is called the *head* and $D_1, \ldots, D_k$ the *body* of the iff-definition. If $A = p(t)$, for some vector of terms t, $A \leftrightarrow D_1 \vee \ldots \vee D_k$ is said to *define* p.

An **(iff-)theory** is a set of iff-definitions. The **completion**, $Comp(P)$ [1] of a logic program P consists of Clark's equality theory (CET) [1], together with an iff-theory consisting of exactly one iff-definition for every predicate different from $=$. Any such iff-definition is of the form

$p(X) \leftrightarrow [X = t_1 \wedge D_1] \vee \ldots \vee [X = t_k \wedge D_k]$
with X a vector of variables, each t_i a vector of terms, and $p(t_i) \leftarrow D_i$, for $i = 1, \ldots, n$, all clauses defining p in P, or of the form
$p(X) \leftrightarrow false$
with X a vector of variables and $false$ a special predicate, if p is not defined in P.

In an iff-theory (one not necessarily resulting from the completion of a logic program), predicates may be defined by no iff-definition, by exactly one iff-definition, or by multiple iff-definitions. For example, the predicate $=$ may be defined by the (possibly infinite) set of iff-definitions $t = t \leftrightarrow true$, for all variable-free terms t, or by the single iff-definition $X = X \leftrightarrow true$ ($true$ is a special predicate). Undefined predicated correspond to the abducible predicates in abductive logic programs. Undefined predicates are distinguished from predicates defined by iff-definitions with $false$ in the body.

The proof procedure defined in the next section uses iff-theories not necessarily resulting from the completion of a logic program. In order to avoid inconsistencies, we assume that any given iff-theory satisfies the restriction that the heads of no two different iff-definitions unify.

Integrity constraints are implications of the form
$L_1, \ldots, L_n \rightarrow A$
with $L_1, \ldots, L_n$ (positive or negative) literals, $n \geq 0$, and A atom, possibly $false$. All variables in A, $L_1, \ldots, L_n$ are implicitly universally quantified, with scope the entire implication. A is called the *conclusion* and $L_1, \ldots, L_n$ are called the *conditions* of the integrity constraint.

Integrity constraints can be used to express "properties" of theories. For example, the integrity constraints
$X < Y \wedge Y < Z \rightarrow X < Z$
$X < Y \wedge Y < X \rightarrow false$
express the transitive and non-symmetric properties of any correct iff-theory defining the predicate $<$. Given the sentence $3 < Y \wedge Y < 1$, the first integrity constraints can be used to add $3 < 1$, and then the second, together with the information $1 < 3$, derivable from any correct iff-theory for $<$, can be used to reduce the original sentence to $false$.

An **initial query** is a conjunction of literals, with all variables free. The proof procedure that we will describe in the next section is a rewriting procedure, consisting of a number of inference rules, each of which replaces a formula by one which is equivalent to it in a given iff-theory, for all variable-free instances of the free variables. We will assume that the given iff-theory contains the (possibly infinitely many) iff-definitions $t = t \leftrightarrow true$, for all variable-free terms t, and some iff-definition for arithmetic predicates such as $<$ and $plus$.

3 The proof procedure

The basic inference rules are:
• **Unfolding**: given an atom A and an iff-definition

$H \leftrightarrow D_1 \vee \ldots \vee D_n$
such that A is an instance of H via a substitution θ (i.e. $A = H\theta$)
A is replaced by $(D_1 \vee \ldots \vee D_n)\theta$.

• **Propagation**: given an atom $p(t)$ and an integrity constraint
$L_1 \wedge \ldots p(s) \ldots \wedge L_n \rightarrow C$,
a new integrity constraint $L_1 \wedge \ldots t = s \ldots \wedge L_n \rightarrow C$ is added.

• **Logical simplification**:
$[B \vee C] \wedge E$ is replaced by $[B \wedge E] \vee [C \wedge E]$ (**splitting**)
$not\, A \wedge B \rightarrow C$ is replaced by $B \rightarrow C \vee A$ (**negation elimination**)
$B \wedge false$ is replaced by $false$
$B \vee false$ is replaced by B, and so on.

• **Equality rewriting**: applies equality rewrite rules (see [4]) simulating the unification algorithm [11].

Unfolding is backward reasoning with iff-definitions, whereas propagation is forward reasoning with integrity constraints. Note that unfolding does not instantiate the unfolded atom A, and therefore differs from backward reasoning in ordinary logic programming. Note also that propagation is a form of resolution. The proof procedure described here is a version of SLP in the sense that SLP adopts full resolution as propagation.

Given an iff-theory T, a set of integrity constraints I and an initial query Q, a **derivation** for Q is a sequence of formulae, $F_1 = Q \wedge I, \ldots, F_n$, such that each F_{i+1} in the sequence is obtained from the previous formula F_i by application of one of the inference rules: unfolding (applied to atoms in bodies of integrity constraints and to atoms that are conjuncts in F_i), propagation (applied to atoms that are conjuncts in F_i), equality rewriting (for equalities) or logical simplification. [1] We assume that negative literals *not* A occurring as conjuncts in any F_i in a derivation are rewritten as integrity constraints $A \rightarrow false$. Note that integrity constraints occurring in a derivation might be integrity constraints in I as well as implications derived from I or from the rewriting of a negative literal by possibly several applications of the inference rules.

Let $F_1 = Q \wedge I, \ldots, F_n = D \vee Rest$ be a derivation such that $D \neq \texttt{false}$, no inference step can be applied to D which has not already been applied earlier in the derivation and all atoms in D are **suspended**, i.e. no atoms in D can be further unfolded. Then, the set of all atoms and *denials* (integrity constraints with $false$ as conclusion) in D is a **computed answer** to Q.

Note that atoms may be suspended because there is no iff-definition for the predicate of the atoms (corresponding to failure in ordinary logic programming and to the assumption of hypotheses in abductive logic programming) or because there are multiple iff-definitions for the predicate of the atoms (corresponding to suspension in concurrent logic languages [14, 13]). Note also that atoms of undefined predicates are always suspended.

[1] To simplify the treatment of quantifiers, both iff-definitions in T and integrity constraints in I are assumed to be range-restricted, i.e. all variables in the head/conclusion must appear in at least one atom in the body/conditions, and this atom must not be an equality between two variables.

Example 1. Let T be the iff-theory

$has(X,Y) \leftrightarrow buys(X,Y) \vee steals(X,Y)$

$honest(x) \leftrightarrow x = tom$

and I consist of the integrity constraint

$honest(x) \wedge steals(x,y) \rightarrow false.$

The predicates $buys$ and $steals$ have no iff-definitions. A possible derivation for $Q{=}has(tom, computer)$ is

$F_1 = Q \wedge I$

$F_2 = Q \wedge [x = tom \wedge steals(x,y) \rightarrow false]$ (by unfolding)

$F_3 = Q \wedge [steals(tom,y) \rightarrow false]$ (by equality rewriting)

$F_4 = [buys(tom, computer) \vee steals(tom, computer)] \wedge [steals(tom,y) \rightarrow false]$ (by unfolding)

$F_5 = [buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]] \vee [steals(tom, computer) \wedge [steals(tom,y) \rightarrow false]]$ (by splitting)

$F_6 = [buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]] \vee [steals(tom, computer) \wedge [y = computer \rightarrow false]]$ (by propagation)

$F_7 = [buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]] \vee [steals(tom, computer) \wedge false]]$ (by equality rewriting)

$F_8 = [buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]] \vee false$
(by logical simplification)

$F_9 = [buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]]$
(by logical simplification)

from which the answer $buys(tom, computer) \wedge [steals(tom,y) \rightarrow false]$ can be extracted.

Example 2. Let T be the iff-theory $p \leftrightarrow p$, I be empty and Q be p. The only possible derivation is

$F_1 = Q \wedge I = p$

$F_2 = p.$

No answer can be extracted from F_2 since p is not suspended.

4 The agent architecture

Every agent can be thought of as an iff-theory together with a set of integrity constraints and an initial query. Atoms in answers are *actions* to be executed as well as *observations* to be performed. Both updates and queries to the agent are treated uniformly as observations. Selected actions can be thought of as outputs into the environment and observations as inputs from the environment. From every agent's viewpoint, the environment contains all the other agents.

The basic engine of an agent is the proof procedure given above, executed via the following cycle:

To cycle at time T,

(i) observe any input at time T,
(ii) record any such input,
(iii) resume the proof procedure by propagating the inputs,
(iv) continue applying the proof procedure,
using for steps (iii) and (iv) a total of R units of time,
(v) select an atomic action which can be executed at time $T + R + 2$,
(vi) execute the selected action at time $T + R + 2$ and record the result,
(vii) cycle at time $T + R + 3$.

The cycle starts at time T by observing and recording any inputs from the environment (steps (i) and (ii)). Steps (i) and (ii) are assumed to take one unit of time each. Then, the proof procedure is applied for R units of time (steps (iii) and (iv)), both assimilating the input and working on any remaining goals. The amount of resources R available in steps (iii) and (iv) is bounded by some predefined amount n. By decreasing n the agent is more *reactive*, by increasing n the agent is more *rational*. Propagation is applied first (step (iii)), in order to allow for an appropriate reaction to the inputs. Finally, an action is selected and executed, taking care to record the result (steps (v) and (vi)). Steps (v) and (vi) conjoined are assumed to take one unit of time.

Selected actions will be amongst the suspended atoms in an answer extracted from a disjunct in a derivation. The disjunct represents an *intention*, i.e. a (possibly partial) plan executed in stages. We can assume that actions from the same disjunct (intention) are selected at different iterations of the cycle. Failure of a selected plan is obtained via logical simplification, after having propagated $false$ into the selected disjunct.

5 Examples

The event calculus [9] is a logic programming formalism for temporal reasoning. A simplified version of the event calculus can be expressed via the iff-definitions:

$$holds\_at(P, T_2) \leftrightarrow happens(E, T_1) \land T_1 < T_2 \land \\ initiates(E, P) \land not\, broken(T_1, P, T_2)$$

$$broken(T_1, P, T_2) \leftrightarrow happens(E, T) \land terminates(E, P) \land T_1 < T < T_2.$$

The first iff-definition states that a property P holds at some time T_2 if and only if it is initiated by an event E occurring at some earlier time T_1 and is not broken (i.e. P persists) from T_1 to T_2. The second iff-definition states that a property P is broken (i.e. does not persist) from a time T_1 to a later time T_2 if an event E that terminates P happens at a time T between T_1 and T_2.

There is no iff-definition for the predicate *happens*. Atoms of this predicate will always be suspended and be generated as part of answers (plans). Atoms of this predicate can also be given as inputsi, or be part of initial queries.

The following integrity constraint

$happens(E, T) \wedge precondition(E, T, P) \wedge not\, holds\_at(P, T) \rightarrow false$

can be used to express that an event E cannot happen at a time T if a precondition P of E does not hold at time T.

The predicates *precondition*, *initiates* and *terminates* are defined by application-specific iff-definitions, e.g.

$$\begin{aligned} precondition(carry\_umbrella, T, P) \leftrightarrow\ & P = own\_umbrella \vee \\ & P = borrowed\_umbrella \end{aligned}$$

$initiates(rain, raining) \leftrightarrow true$

$terminates(sun, raining) \leftrightarrow true.$

Additional integrity constraints might be given to represent reactive behaviour of intelligent agents, e.g.

$happens(rain, T) \rightarrow happens(carry\_umbrella, T + 1)$

and to prevent concurrent execution of actions (events):

$happens(E, T) \wedge happens(E', T) \rightarrow E = E'.$

An answer computed by the proof procedure can be interpreted as sets of actions to be performed to achieve given goals (queries) and/or to react to given events. For example, given the query

$happens(rain, 5)$

the proof procedure computes the answer $happens(carry\_umbrella, 6)$.

Within cycle, the query is an input at time 5, and the computed answer is a selected action at time 6, matching the time within the action.

6 Conclusions

We have presented a version of the proof procedure of [10]. The procedure combines backward reasoning with iff-definitions and forward reasoning with integrity constraints. We have illustrated the application of the proof procedure within the agent architecture of [6–8]. Here, backward reasoning with iff-definitions corresponds to reduction of goals to sub-goals and forward reasoning with integrity constraints corresponds to reactivity to inputs. Agents' plans of action are generated via a combination of the two reasoning modes. The execution of (possibly partial) plans is achieved via a particular execution of the proof procedure, where processing of inputs from the external environment in interleaved with application of the procedure.

Acknowledgements

This reasearch was supported by the UK EPSRC project "Logic-based multi-agent systems".

References

1. K.L. Clark, Negation as failure. *Logic and Data Bases*, Gallaire and Minker eds., Plenum, New York(1978) 293–322
2. Console, L., Theseider Dupré, D., Torasso, P. On the relationship between abduction and deduction. *Journal of Logic and Computation* 2(5) (1991) 661–690
3. M. Denecker, D. De Schreye, SLDNFA: an abductive procedure for normal abductive programs. Proceedings of *International Conference and Symposium on Logic Programming* (1992), 686–700
4. T.H. Fung, R.A. Kowalski, The iff procedure for abductive logic programming. *Journal of Logic Programming* 33(2):151–165 (1997)
5. A.C. Kakas, R.A. Kowalski, F. Toni, The role of abduction in logic programming. *Handbook of logic in Artificial Intelligence and Logic Programming* 5, 235–324, D. Gabbay, C. Hogger, and A. Robinson eds., Oxford University Press (1998)
6. R.A. Kowalski, Using meta-logic to reconcile reactive with rational agents. In *Meta-logic and Logic Programmimg*, K.Apt and F. Turini (eds.), MIT Press (1995), 227–242
7. R.A. Kowalski, F. Sadri, Towards a unified agent architecture that combines rationality with reactivity. In *Proc. International Workshop on Logic in Databases*, San Miniato, Italy, Springer-Verlag LNCS 1154 (1996)
8. R.A. Kowalski, F. Sadri, An agent architecture that unifies rationality with reactivity. Imperial College Technical Report (1997)
9. Kowalski, R.A. and Sergot, M.: "A logic-based calculus of events", *New Generation Computing* **4**, 1986, 67–95.
10. R.A. Kowalski, F. Toni, G. Wetzel, Executing suspended logic programs. To appear in *Fundamentae Informaticae* (1998)
11. A. Martelli, U. Montanari, An efficient unification algorithm, *ACM Trans. on Prog. Lang. and Systems* 4(2), 258–282, 1982
12. F. Sadri and F. Toni, Active behaviour in deductive databases, Imperial College Technical Report (1996)
13. Saraswat, V. A. and Rinard, M.: "Concurrent Constraint Programming", *Proc. of the 17$^{\text{th}}$ ACM Symp. on the Principles of Programming Languages*, 1990, 232–245.
14. Shapiro, E.: "The Family of Concurrent Logic Programming Languages", *ACM Computing Surveys* **21 (3)**, 1989, 413–510.
15. Y. Shoham, Agent-oriented programming. *Artificial Intelligence* 60(1) (1993), 51–92