Representing Incomplete Knowledge in Abductive Logic Programming*

Marc Denecker † Danny De Schreye ‡

Department of Computer Science, K.U.Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium.
e-mail : {marcd, dannyd}@cs.kuleuven.ac.be

Abstract

Recently, Gelfond and Lifschitz presented a formal language for representing incomplete knowledge on actions and states, and a sound translation from this language to extended logic programming. We present an alternative translation to abductive logic programming with integrity constraints and prove the soundness and completeness. In addition, we show how an abductive procedure can be used, not only for *explanation*, but also for deduction and proving satisfiability under uncertainty.

From a more general perspective, this work can be viewed as a -successful- experiment in the declarative representation of and automated reasoning on incomplete knowledge using abductive logic programming.

1 Introduction

The key issue in this paper is the representation of incomplete knowledge in a logic programming formalism. In pure Prolog, incomplete knowledge cannot be represented: due to negation as failure, an atom which cannot be proved is assumed to be false. Recently, [17] used extended programs to represent incomplete knowledge in the context of temporal domains. They introduced a new temporal language $\mathcal{A}$ which allows to represent a number of well-known benchmark problems involving incomplete temporal knowledge and they proposed a sound transformation to extended programs, programs with both negation as failure and classical or explicit negation [16].

In the past, another approach has been explored for temporal reasoning, based on event calculus [13], [30], [25], [10]. [10] proposes solutions for the same benchmarks as in [17]. This approach makes use of the formalism of abductive logic programming. One may interpret an abductive program as an *open* logic program in the sense that it contains only definitions for the non-abducible predicates. The completion semantics for abductive logic programs of [3] reflects this well by taking as the completion of an abductive program the set of completed definitions of the non-abducible predicates only. As a consequence, the completion does not impose any restriction on the interpretation of the abducible predicates and incomplete knowledge can be represented via the undefined predicates. Observe that this treatment

*Accepted for publication in the Journal of Logic and Computation

†supported by a postdoctoral mandate from the research fund of the K.U.Leuven

‡supported by the Belgian National Fund for Scientific Research

differs totally from the treatment of a defined predicate with empty definition: this latter predicate is always false.

In general an abductive program will allow too much uncertainty. In most applications, the knowledge engineer has knowledge that implicitly restricts the possible states of the undefined abductive predicates. A declarative way of representing this knowledge is by First-order Logic integrity constraints. In order to illustrate this approach, we present a transformation from $\mathcal{A}$ domain descriptions to open programs with First-order Logic integrity constraints (section 3). Despite the fact that the benchmarks in [10] were solved with abductive event calculus, it turns out to be difficult to transform $\mathcal{A}$ domain descriptions to event calculus programs: as situation calculus, $\mathcal{A}$ is based on the *branching time* philosophy, whereas event calculus is based on a *linear time* philosophy. However, the use of the open programs is in no way restricted to event calculus. The proposed transformation maps an $\mathcal{A}$ domain description to an open situation calculus.

A second high level goal is to illustrate how an abductive procedure can be useful for automated reasoning with abductive programs and integrity constraints. That an abductive procedure can be used for *explanation* of some observation is well-known from [28], [30]. It is less known that an abductive procedure can also be used deduction and for proving consistency of a theory. In section 4, we illustrate how an abductive procedure can be used to prove the consistency of an abductive logic program with first order logic integrity constraints and that an abductive procedure can be used for deduction in an abductive logic program.

In the rest of the paper we will mostly refer to *abductive programs* as *open programs*, and to *abducible predicates* as *undefined predicates*. We prefer these names over the commonly used terminology because they reflect the declarative meaning of the concepts: an undefined predicate has no definition; an open program is open in the sense that it describes the defined predicates in terms of a set of undefined predicates whose interpretation is completely left open. The terminology *abductive program* and *abducible predicate* is somewhat misleading, especially in the context of this paper, whose goal it is to show explicitly how to perform other forms of reasoning than abduction on the *abductive programs*.

The paper is structured as follows. In section 2, we recall the language $\mathcal{A}$ and its semantics. In section 3, the transformation from $\mathcal{A}$ to situation calculus programs is presented and the soundness and completeness are proved. In section 4, the use of abduction for explanation and deduction is illustrated. Section 5 gives a comparison between our transformation and the transformation in [17]. Section 6 compares our work with another recent transformation from $\mathcal{A}$ proposed in [11]. In section 7, we discuss other related work. A short paper on this subject has been published as [8].

2 The temporal language $\mathcal{A}$

The language $\mathcal{A}$ [17] allows to describe relationships between fluents (= time dependent properties of the world) and actions. $\mathcal{A}$ is a propositional language: both fluents and actions are represented by propositional symbols. Two types of expressions occur. A v-proposition describes the value of a fluent after a (possibly empty) sequence of actions. Its syntax is as follows:

$$f \textbf{ after } a_1; \ldots ; a_n$$

Here $a_1; \ldots ; a_n$ is a sequence of action symbols and f is a fluent expression: a positive or negative literal containing a fluent. The expression means that f is true after executing the sequence of actions $a_1; \ldots ; a_n$. When the sequence of actions is empty (n=0), the v-proposition describes the initial situation. Instead of f **after** , one usually writes:

Initially f

An e-proposition describes the effect of actions on the fluents. It has the form:

a **causes** f **if** $p_1, \ldots, p_n$

where $f, p_1, \ldots, p_n$ are fluent expressions and a is an action symbol. The expression means that if $p_1, \ldots, p_n$ are true, then the effect of a on the current situation is that f becomes true. $p_1, \ldots, p_n$ are called preconditions. When n=0, one writes:

a **causes** f.

A domain description is a set of v- and e-propositions.

Example We recall the Yale Turkey Shooting problem (YTS) as formulated in [17]. The fluents are *loaded*, *alive*; the action names are *shoot*, *wait* and *load*. The domain description D_0 contains the following propositions:

Initially *alive*
Initially $\neg$*loaded*
load **causes** *loaded*
shoot **causes** $\neg$*alive* **if** *loaded*
shoot **causes** $\neg$*loaded*

Example The Murder Mystery domain D_1 is a variant of YTS, obtained by substituting

$\neg$*alive* **after** *shoot*; *wait*

for **Initially** $\neg$*loaded* in D_0. This is a prototypical postdiction problem: we want to obtain the conclusion that initially the gun is loaded.

The semantics for $\mathcal{A}$ is defined as follows. A *state* is a set of fluent names and describes a possible state of the world. Given a fluent symbol f and a state σ, f holds in σ if $f \in \sigma$, otherwise $\neg f$ holds in σ. A transition function Φ maps pairs (a, σ) of action symbols a and states σ into the set of states. Φ describes how a situation changes under application of an action a. A structure M is a pair (σ_0, Φ), where σ_0 represents the initial state and Φ the transition function. $M^{a_1;\ldots;a_m}$ denotes the state $\Phi(a_m, \Phi(a_{m-1}, \ldots, \Phi(a_1, \sigma_0)\ldots))$. A v-proposition

f **after** $a_1; \ldots ; a_m$

holds in a structure (σ_0, Φ) iff f holds in $M^{a_1;\ldots;a_m}$.

Definition 2.1 *A structure $M = (\sigma_0, \Phi)$ is a model of a domain expression D if and only if the following rules are satisfied:*

- *Each v-proposition f* **after** *$a_1; \ldots ; a_m$ holds in M.*
- *For any state σ, fluent symbol f and action a, if there exists an e-proposition*

  a **causes** f **if** $p_1, \ldots, p_n$

  such that $p_1, \ldots, p_n$ hold in σ, then f holds in $\Phi(a, \sigma)$. If there is an e-proposition a **causes** *$\neg f$* **if** *$p_1, \ldots, p_n$ such that $p_1, \ldots, p_n$ hold in σ, then $\neg f$ holds in $\Phi(a, \sigma)$. Otherwise, f holds in $\Phi(a, \sigma)$ iff f holds in σ.*

A domain description is called *consistent* if it has a model. We introduce a new notion, *e-consistency*. A domain description is *e-consistent* if the set of e-propositions of D is consistent. There is a simple necessary and sufficient condition for a domain description to be e-consistent.

Lemma 2.1 *A domain description D is e-consistent iff for each pair of rules*

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \text{ and } a \textbf{ causes } \neg f \textbf{ if } p_{n+1}, \ldots, p_m$$

in D, there exists an i and j such that p_i is the complement of p_j.

This condition is satisfied when the complementary literals are found in the bodies of the two rules but also when they appear in the body of one rule, as in

$$shoot \textbf{ causes } alive \textbf{ if } loaded, \neg loaded$$

Such a rule has an inconsistent body. It can never be applied and can never cause an inconsistency.

In [11], an action a for which there exists a pair of rules

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \text{ and } a \textbf{ causes } \neg f \textbf{ if } p'_1, \ldots, p'_m$$

such that no complement of a literal in the first is contained in the second is called self-contradicted. [11] contains a similar proposition as lemma 2.1.

Proof Assume that for some pair of rules:

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \text{ and } a \textbf{ causes } \neg f \textbf{ if } p_{n+1}, \ldots, p_m$$

no complementary literals occur in $\{p_1, \ldots, p_m\}$. This set may contain positive and negative literals. Define the state σ consisting of only the positive literals in $\{p_1, \ldots, p_m\}$. Then obviously each p_i holds in σ. $\Phi(a, \sigma)$ is not consistently defined because both f and $\neg f$ should hold in $\Phi(a, \sigma)$.

Vice versa, one easily verifies that when the syntactical condition in the lemma is satisfied, the definition of a model of a domain description gives a consistent description of a transition function Φ. □

Definition 2.2 (Non-inertial) *We say that a fluent f is non-inertial under an action a in a state σ iff there exists an e-proposition*

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \text{ or } a \textbf{ causes } \neg f \textbf{ if } p_1, \ldots, p_n$$

such that $p_1, \ldots, p_n$ hold in σ. Otherwise, it is inertial under action a.

[17] observes that the e-propositions of a domain D completely determine the transition function Φ. Below proposition 2.1 gives a precise and deterministic characterisation of Φ, under the condition that D is e-consistent.

Proposition 2.1 *Let D be e-consistent. For any state σ, action a and positive fluent symbol f: $f \in \Phi(a, \sigma)$ iff*

- *f holds in σ and f is inertial under a in state σ, or*

- *there exists an e-proposition a* **causes** f **if** $p_1, \ldots, p_n$ *such that* $p_1, \ldots, p_n$ *hold in* σ.

The proof is straightforward.

Note that if the condition of e-consistency is not satisfied, then the description of Φ in the proposition does not correspond to the description of Φ in definition 2.1. Indeed, for f, a and σ such that for e-propositions

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \text{ and } a \textbf{ causes } \neg f \textbf{ if } p'_1, \ldots, p'_m$$

the fluents $p_1, \ldots, p_n, p'_1, \ldots, p'_m$ hold in σ, the second item of definition 2.1 requires that both $f \in \Phi(a, \sigma)$ and $f \notin \Phi(a, \sigma)$. This is a contradiction. On the other hand, proposition 2.1 requires that $f \in \Phi(a, \sigma)$ since the first item applies. Because the transformation proposed in next section implements the formulation of proposition 2.1 rather than that of definition 2.1, it will be complete only for e-consistent domains.

A v-proposition Q is *entailed* by a domain description D iff Q holds in each model of D. A domain description is called *complete* if it has a unique model. The YTS domain and Murder Mystery domain are examples of complete domain descriptions. Since they share their e-propositions, their models have an identical transition function Φ which maps tuples $(wait, \sigma)$ on σ, $(load, \sigma)$ on $\sigma \cup \{loaded\}$, and maps $(shoot, \sigma)$ on $\sigma \setminus \{alive, loaded\}$ if $loaded \in \sigma$, otherwise on σ. The model M_0 of D_0 has initial situation $\{alive\}$. The model M_1 of D_1 has initial situation $\{alive, loaded\}$. An *incomplete* domain description is obtained by dropping the v-proposition **Initially** *alive* from the Murder Mystery domain. One additional model is the structure with transition function Φ but with initial situation $\{\}$.

$\mathcal{A}$ provides only restricted expressivity: the language is only propositional, no relationships between fluents can be defined, no indeterminate events are allowed. Nevertheless, $\mathcal{A}$ allows to formalise several interesting domains. This and its clear semantics makes the language interesting for experiments as in [17], [11] and in this chapter.

3 Translation to Open Logic Programs

In this section we present a general translation from an $\mathcal{A}$ domain description D to an open logic program with integrity constraints. An open logic program is a pair of a set of normal clauses, Horn clauses augmented with *negation as failure* and a set of predicates, called undefined or abducible, such that the undefined predicates occur only in the body of the rules. Intuitively, an open logic program allows to represent incomplete information because it leaves open the interpretation of the undefined predicates. To see this, consider the completion semantics of Console, Theseider Dupré and Torasso for abductive logic programs [3], an extension of Clarks completion semantics. According to this completion semantics, the declarative meaning of an open logic program P is given by the theory, denoted $comp_a(P)$, which consists of the axioms of Free Equality (*FEQ*) (also called Clark Equality Theory or Unique names axioms) and the completed definitions of all *defined* predicates, i.e. all predicates of the language which are not undefined. Otherwise said, $comp_a(P)$ is abtained by taking P's Clark completion and dropping for each undefined predicate p/n its completed definition $p(\overline{X}) \leftrightarrow false$.

Note that $comp_a(P)$ contains only equivalences with defined predicates at the left. Intuitively, such an equivalence defines when the predicate at the left is true in terms of the situations described at the right. Since undefined predicates have no such *defining* equivalence, the logic program leaves their interpretation open to a great extend[1]. In general, an abductive logic program allows to much freedom to the interpretation of undefined predicates. Most often, the program P must be augmented with a set $\mathcal{T}$ of First-order Logic (FOL) axioms representing other information which restricts the state of the undefined predicates. The declarative semantics of a couple $(P, \mathcal{T})$ is given by the FOL theory $comp_a(P) + \mathcal{T}$. This defines indirectly a model semantics for $(P, \mathcal{T})$: in the sequel, when we talk about a model of $(P, \mathcal{T})$ we mean a classical model of $comp_a(P) + \mathcal{T}$.

That the resulting formalism is adequate for representing incomplete information is now obvious: for $P = \{\}$, the theory $comp_a(P) + \mathcal{T}$ collapses to $FEQ + \mathcal{T}$, i.e. classical logic with unique names axioms. The expressivity of this formalism for representing incomplete information is widely accepted.

Our transformation from $\mathcal{A}$ to the above formalism produces programs in situation calculus style. Traditionally, two options are available to represent a fluent f in a logic formalism: by a predicate $f(s)$ or by $Holds(f, s)$ where s is a state argument. Then $\neg f$ is translated to $\neg f(s)$ or $\neg Holds(f, s)$. The two approaches are equivalent but the meta-approach has the advantage that the frame axiom can be stated for all fluents at once, whereas in the first approach one frame axiom per fluent predicate is needed. As in [17], we use $Holds/2$. The first order language $\mathcal{L}_D$ contains the predicate symbols $Holds/2$, $Noninertial/3$ and $Initially/1$. Each fluent and action symbol occurs in $\mathcal{L}_D$ as a constant. In addition, there is a constant s_0 to denote the initial state and a functor $Result/2$: $Result(a, s)$ denotes the state obtained by applying action a on state s. In the sequel, we will use $Result[a_1; \ldots; a_n, s]$ as a shorthand notation for $Result(a_n, Result(a_{n-1}, \ldots Result(a_1, s) \ldots))$. For $n = 0$, this denotes s.

$\mathcal{A}$ allows uncertainty on the initial state. Correspondingly, the open program comprises one undefined predicate, $Initially/1$. The translation maps a domain description D to a theory πD consisting of an open logic program P_D and a set of FOL axioms IC_D. P_D is defined as follows:

- Initialisation:
$$Holds(F, s_0) \text{ :- } Initially(F) \tag{1}$$
- Law of Inertia:
$$Holds(F, Result(A, S)) \text{ :- } Holds(F, S), \neg Noninertial(F, A, S) \tag{2}$$
- For each e-proposition a **causes** f **if** $p_1, \ldots, p_m, \neg p'_1, \ldots, \neg p'_n$ with f, p_i and p'_j positive literals:
$$\begin{aligned} Holds(f, Result(a, S)) \text{ :- } & Holds(p_1, S), \ldots, Holds(p_m, S), \\ & \neg Holds(p'_1, S), \ldots, \neg Holds(p'_n, S) \end{aligned} \tag{3}$$

[1] As so often with completion semantics, examples can be formulated which contradict this intuition. For example, the 2-valued completion of the program:

$$p \text{ :- } \neg p, \neg a$$

with undefined predicate a entails that a is true. The problem is caused by the loop over negation of p. For programs which do not contain such loops, it can be shown that the abductive predicates can have any interpretation. In other semantics such as the 3-valued completion semantics for abductive programs [4], the justification semantics for abductive programs [7] and the generalised well-founded semantics for abductive logic programs [27], even for programs with loops over negation, the interpretation of the undefined predicates can be any. Despite these problems with 2-valued completion semantics, we use it here because of its declarative simplicity and its close relationship with First-order Logic.

As in [17], we introduce the convention that when f is a negative literal $\neg f'$, $Holds(f,t)$ is used as a textual denotation for $\neg Holds(f',t)$. This handsome convention allows us to say that a **causes** f **if** $p_1,\ldots,p_n$ is translated to the clause:

$$Holds(f, Result(a,S)) \text{:- } Holds(p_1,S),\ldots,Holds(p_n,S)$$

without considering the sign of the literals p_i. Be aware that a program should never contain literals of the form $Holds(\neg f, S)$, and that when these literals are found in this chapterand the following sections, they always stand for $\neg Holds(f,S)$.

- For each e-proposition a **causes** f **if** $p_1,\ldots,p_n$ with f a positive or negative fluent literal:
$$Noninertial(|f|,a,S) \text{:- } Holds(p_1,S),\ldots,Holds(p_n,S) \tag{4}$$
For a fluent symbol f, $|f|$ and $|\neg f|$ both denote the term f.

The set of FOL axioms IC_D is defined as follows:

- For each v-proposition f **after** $a_1;\ldots;a_n$ (n ≥ 0):
$$Holds(f, Result[a_1;\ldots;a_n, s_0]) \tag{5}$$
with the same syntactic convention on $Holds/2$ as above.

Example The domain description D_0 for the YTS problem is transformed to:

$Holds(F,s_0) \text{:- } Initially(F)$
$Holds(F, Result(A,S)) \text{:- } Holds(F,S), \neg Noninertial(F,A,S)$
$Holds(loaded, Result(load,S)) \text{:-}$
$Noninertial(loaded, load, S) \text{:-}$
$Noninertial(loaded, shoot, S) \text{:-}$
$Noninertial(alive, shoot, S) \text{:- } Holds(loaded, S)$

$Holds(alive, s_0)$
$\neg Holds(loaded, s_0)$

The clause $Noninertial(loaded, load, S)$:- may be dropped from this program without effect on the semantics of Holds/2. In general, all $Noninertial/3$ rules for initiating effects of actions may be dropped, without effect on the semantics of Holds/2.

πD_0 strongly resembles the YTS solution in [1] and in [14]. For example, [1] proposes a Prolog program analogous to the one obtained from πD_0 by substituting the program clause:

$Holds(alive, s_0) \text{:-}$

for the program clauses:

$Holds(F,s_0) \text{:- } Initially(F)$
$Noninertial(loaded, load, S) \text{:-}$

and the two FOL axioms of πD_0. Note that the resulting program entails the two FOL axioms. [1] proves that the resulting program is acyclic. The same holds for πD_0, and in fact for all transformed domain descriptions:

Proposition 3.1 *The translation πD of any domain description D is acyclic.*

Proof A slight modification of the level mapping proposed in [1] for the YTS program applies for all domain descriptions. For all ground terms t, let $|t|_{Result}$ denote the number of occurrences of the functor $Result/2$ in t. We define $|.|$ for all ground terms t, a and s as follows:

$$|Initially(t)| = 0$$
$$|Holds(t, s)| = 2 \times |s|_{Result} + 1$$
$$|Noninertial(t, a, s)| = 2 \times |s|_{Result} + 2$$

One easily verifies that $|.|$ is a level mapping. □

Several types of semantics have been defined for open programs: 2-valued completion semantics [3], generalised stable semantics [20], the generalised well-founded semantics [27], 3-valued completion semantics and 3-valued (direct) (partial) justification semantics with FEQ[7]. Due to the fact that πD is acyclic and in each clause of P_D, the variables of the body occur in the head, all these semantics coincide in the (weak) sense that the set of all ground atoms implied by πD under any of the semantics is identical. This extension of results of [1] is proven formally in [4]. Because of these results, we can investigate the soundness and completeness of the transformation under the simplest semantics for open logic programs, the completion semantics of [3].

The translation πD of a domain description contains two defined predicates, $Holds/2$ and $Noninertial/3$. The completed definition of $Holds/2$ is of the form:

$$\forall F, T : Holds(F, T) \leftrightarrow E_1 \vee E_2 \vee \ldots \vee E_m \qquad (6)$$

with:

$$\begin{aligned} E_1 = \quad & T = s_0 \wedge Initially(F) \\ E_2 = \quad & \exists A, S : T = Result(A, S) \wedge \\ & \quad Holds(F, S) \wedge \neg Noninertial(F, A, S) \\ E_i(i > 2) = \quad & \exists S : F = f \wedge T = Result(a, S) \wedge \\ & \quad Holds(p_1, S) \wedge \ldots \wedge Holds(p_n, S) \end{aligned}$$

such that precisely for each e-proposition a **causes** f **if** $p_1, \ldots, p_n$ with f a positive literal, there exists one disjunct E_i $(i > 2)$ in the completed definition. If we map S on state σ, the $Result$ functor on Φ and $Holds(F, S)$ to $F \in \sigma$ then the completed definition of $Holds/2$ is similar to proposition 2.1.

The completed definition of $Noninertial/3$ is of the form:

$$\forall A, F, S : Noninertial(F, A, S) \leftrightarrow E_1 \vee \ldots \vee E_m \qquad (7)$$

with:

$$E_i = \quad A = a \wedge F = |f| \wedge Holds(p_1, S) \wedge \ldots \wedge Holds(p_n, S)$$

such that precisely for each e-proposition a **causes** f **if** $p_1, \ldots, p_n$ (f positive or negative), there exists one corresponding disjunct in the completed definition. This formula is the counterpart of definition 2.2: the formulation is almost identical apart from the fact that $Holds(p_i, S)$ should be replaced by "p_i holds in S".

A transformation such as π can be considered correct if the set of entailed formulas are equivalent. In [17] a translation π is defined to be sound iff for each domain D and v-expression Q, if $\pi D \models \pi Q$ then D entails Q. π is defined to be complete if the reverse holds: if D entails Q then $\pi D \models \pi Q$.

Theorem 3.1 (soundness) *Let D be a domain description. For any v-proposi-tion Q, if $\pi D \models \pi Q$ then D entails Q.*

For the proof of the theorem, we introduce the concept of a state function.

Definition 3.1 *Let M be any interpretation of $\mathcal{L}_D$, x a domain element of M.*

$$state_M(x) = \{f \mid f \textit{ is a positive fluent symbol and } M \models Holds(f, x)^2\}$$

Note that $state_M(x)$ denotes a state as in section 2, and a transition function Φ can be applied on it. Note also that $f \in state_M(x)$ is equivalent with $M \models Holds(f, x)$.

Proof of Theorem 3.1.

Nothing is to be proved when D is inconsistent. So, assume D is consistent. It suffices to show that for each model $M_a = (\Phi, \sigma_0)$ of D and for each v-proposition Q, there exists a model M of πD such that Q holds in (Φ, σ_0) iff $M \models \pi Q$. If then πQ holds in all models of πD, then also in these models corresponding to models of D. Hence, Q holds in all models of D.

We construct a Herbrand model M. $HU(\mathcal{L}_D)$ denotes the Herbrand universe. The basic idea is simple: we define M such that for each state term $s = Result[a_1; \ldots; a_n, s_0]$, $state_M(s) = M^{a_1;\ldots;a_n}$ and such that the atom $Noninertial(a, f, s)$ is true whenever f is non-inertial under a in the state $state_M(s)$. Things are slightly complicated due to the fact the πD is not a sorted program and ill-sorted atoms may occur in the model. For that reason, we extend D to D' by allowing each term $t \in HU(\mathcal{L}_D)$ as a fluent symbol and as an action symbol. Note that non-original fluents t_f (symbols of $HU(\mathcal{L}_D)$ which are not fluent symbols in D) and non-original actions t_a do not occur in the e-propositions. This implies that a non-original fluent t_f of D' always remains as in the initial state; a non-original action t_a has no effect on a state (like the action *wait* in the YTS D_0). (Φ, σ_0) can easily be extended to a model (Φ', σ'_0) of D'. Define $\sigma'_0 = \sigma_0$. Let σ' be any state of D', consisting of original fluents σ and new fluents σ_n. Extend Φ to Φ' in the following way:

$$\Phi'(t_a, \sigma') = \sigma' \quad \text{for } t_a \text{ a non-original action}$$
$$\Phi'(t_a, \sigma') = \Phi(t_a, \sigma) \cup \sigma_n \quad \text{for } t_a \text{ an original action}$$

One easily verifies that (Φ', σ'_0) is a model of D'. Moreover, for any v-proposition Q based on the original language of D, Q holds in (Φ', σ'_0) iff Q holds in (Φ, σ_0).

Next we associate to (Φ', σ'_0) a Herbrand model M of $\pi D = \pi D'$. With any term $t_s \in HU(\mathcal{L}_D)$ we associate a specific state of D'. Below we call a term t_s an empty-state term if $t_s \neq s_0$ and $t_s \neq Result(t_1, t_2)$ for some t_1, t_2. For any term $t_s \in HU(\mathcal{L}_D)$, $state_M(t_s)$ is constructed as follows:

$$\begin{array}{ll}
t_s = s_0 & \Rightarrow state_M(t_s) = \sigma_0 \\
t_s \text{ is an empty-state term} & \Rightarrow state_M(t_s) = \phi \\
t_s = Result[t_1; \ldots; t_n, s_0] & \Rightarrow state_M(t_s) = \Phi'[t_1; \ldots; t_n, \sigma_0] \\
\multicolumn{2}{l}{t_s = Result[t_1; \ldots; t_n, t_0],\ t_0 \text{ an empty-state term}} \\
& \Rightarrow state_M(t_s) = \Phi'[t_1; \ldots; t_n, \phi]
\end{array}$$

M is defined as follows:

$$\begin{array}{l}
\{Holds(t_f, t_s) \mid t_f \in state_M(t_s)\} \cup \\
\{Noninertial(t_f, t_a, t_s) \mid t_f \text{ is non-inertial under } t_a \\
\qquad\qquad\qquad\qquad\qquad \text{in } state_M(t_s)\} \cup \\
\{Initially(t_f) \mid t_f \in \sigma_0\}
\end{array}$$

Clearly for any v-proposition Q using original symbols of D, it holds that Q holds in (Φ, σ_0) iff $M \models \pi Q$. A direct consequence is that M is a model of IC_D. It remains to prove that M is a model of $comp_a(P_D)$.

Before continuing with this proof, we want to stress that the complexity of the construction above is in no way an indication that the proposed transformation π is on itself unnecessarily complex or lacks elegance. The increased technicality is only due to the fact that πD can be considered as an untyped meta-program. It is well-known (see e.g. [19], [24]) that such programs give rise to technical problems with respect to Herbrand semantics. Alternatives would have been to define πD as a typed logic program (as in [11]), or to make its clauses range restricted, using additional range predicates. Both solutions would have reduced the complexity of the proof, but increased the complexity of π itself. This motivates our choice.

That the completed definition of $Noninertial/3$ is satisfied follows straightforwardly: since the expression "p_i holds in $state_M(t_s)$" is equivalent with $M \models Holds(p_i, t_s)$, the completed definition of $Noninertial/3$ is a direct representation of definition 2.2.

Finally, we check the completed definition of $Holds/2$. Essentially what must be done is to check all its ground instances $Holds(t_f, t_s) \Leftrightarrow \ldots$. This requires a simple case-analysis depending on the type of t_s. We consider three cases. Take $t_s = s_0$. The completed definition collapses to:

$$Holds(t_f, s_0) \leftrightarrow Initially(t_f)$$

which is clearly satisfied in M.

Take t_s an empty-state term. The completed definition collapses to:

$$Holds(t_f, t_s) \leftrightarrow false$$

which is also satisfied in M.

Finally, take $t_s = Result(t_a, t)$. The completed definition collapses to:

$$Holds(t_f, Result(t_a, t)) \leftrightarrow E_1 \vee \ldots E_n$$

with:

$$\begin{array}{ll} E_1 = & Holds(t_f, t) \wedge \neg Noninertial(t_f, t_a, t) \\ E_i (i > 1) = & Holds(p_1, t) \wedge \ldots \wedge Holds(p_n, t) \end{array}$$

such that precisely for each e-proposition

$$t_a \textbf{ causes } t_f \textbf{ if } p_1, \ldots, p_n$$

in D there exists one corresponding disjunct E_i $(i > 1)$ in the formula. Substituting:

$$"t_f \in state_M(Result(t_a, t))"$$

for:

$$Holds(t_f, Result(t_a, t))$$

and substituting:

$$"p_i \text{ holds in } state_M(t)"$$

for:

$$Holds(p_i, t)$$

preserves the truth value of the expression wrt to M. Now observe that by the definition of M, we have

$$t_f \in state_M(Result(t_a, t)) \text{ iff } t_f \in \Phi'(t_a, state_M(t))$$

Hence substituting:

$$t_f \in \Phi'(t_a, state_M(t))$$

for:

$$t_f \in state_M(Result(t_a, t))$$

at the left in the equivalence, preserves again the truth value of the expression. Clearly, we obtain an equivalence as in proposition 2.1. This equivalence is satisfied because Φ' is the transition function of a model of D' and D' is e-consistent.

□

Theorem 3.2 (completeness) *Let D be e-consistent. For each v-proposition Q, if D entails Q then $\pi D \models \pi Q$.*

Proof Since D is e-consistent, there exists a unique transition function Φ which satisfies the e-propositions of D. As for the soundness, it suffices to prove that for each model M of πD, there exists a model $M_a = (\sigma_0, \Phi)$ of D such that for each v-proposition Q, $M \models \pi Q$ iff Q holds in M_a. Notice that this immediately implies that all v-propositions of D hold in M_a since M is a model of πQ for each v-proposition Q of D.

M maps each term $Result[a_1; \ldots; a_n, s_0])$ to a domain element which we denote as:

$$\tilde{M}(Result[a_1; \ldots; a_n, s_0])$$

M_a is defined in the following way: Φ is given; σ_0 is defined as $state_M(\tilde{M}(s_0))$.

We should prove that for each sequence of actions $a_1, \ldots, a_n$:

$$M_a^{a_1;\ldots;a_n} = state_M(\tilde{M}(Result[a_1; \ldots; a_n, s_0]))$$

The proof is by induction on n. For $n = 0$, this is trivial. So assume that the theorem holds for $n-1$, $n > 0$. We have the following identity:

$$\begin{aligned} M_a^{a_1;\ldots;a_n} &= \Phi(a_n, M_a^{a_1;\ldots;a_{n-1}}) \\ &= \Phi(a_n, state_M(\tilde{M}(Result[a_1; \ldots; a_{n-1}, s_0]))) \end{aligned}$$

The second identity follows from the induction hypothesis. Let x be the domain element $\tilde{M}(Result[a_1; \ldots; a_{n-1}, s_0])$. It suffices to show that:

$$\begin{aligned} &\Phi(a_n, state_M(\tilde{M}(Result[a_1; \ldots; a_{n-1}, s_0]))) = \\ &\qquad state_M((\tilde{M}(Result[a_1; \ldots; a_n, s_0])) \end{aligned}$$

or equivalently:

$$\Phi(a_n, state_M(x)) = state_M(\tilde{M}(Result(a_n, x)))$$

By proposition 2.1, we find that $f \in \Phi(a_n, state_M(x))$ iff

- f holds in $state_M(x)$ and f is inertial under a_n in $state_M(x)$, or

- there exists an e-proposition a_n **causes** f **if** $p_1, \ldots, p_m$ such that the fluents p_1, $\ldots$, p_m hold in $state_M(x)$.

Because M is a model of $Noninertial/3$, the first disjunct corresponds to

$$M \models Holds(f, x) \wedge \neg Noninertial(f, a_n, x)$$

The second disjunct corresponds to the fact that

$$M \models Holds(p_1, x) \wedge \ldots \wedge Holds(p_m, x)$$

for some e-proposition a_n **causes** f **if** $p_1, \ldots, p_m$. Because M is a model of the completed definition of $Holds/2$, we obtain that

$$f \in \Phi(a_n, state_M(x)) \text{ iff } M \models Holds(f, Result(a_n, x))$$

or equivalently

$$f \in state_M(\tilde{M}(Result(a_n, x)))$$

This gives the desired identity.

□

The following example shows that the condition of e-consistency is necessary: π is not complete in general.

Example Consider the following domain description D_2, which uses the fluent *alive* and the action *shoot*.

shoot **causes** *alive*
shoot **causes** $\neg$*alive*

Obviously, D_2 is inconsistent: no transition function Φ can exist which satisfies the two e-propositions. Therefore, each v-proposition is entailed by D_2. πD_2 is given by:

$Holds(F, s_0)$:- $Initially(F)$
$Holds(F, Result(A, S))$:- $Holds(F, S), \neg Noninertial(F, A, S)$
$Holds(alive, Result(shoot, S))$:-
$Noninertial(alive, shoot, S)$:-

This program is consistent. Below, $Result[shoot; \ldots; shoot, s_0]$ is denoted by $shoot^n$. A Herbrand model of πD_2 is given by the set:

$$\{Holds(alive, shoot^n), Noninertial(alive, shoot, shoot^n) \mid n > 0\}$$

In this model, the e-proposition *shoot* **causes** *alive* overrules the contradicting rule *shoot* **causes** $\neg$*alive*. π is not complete since D_2 entails all v-propositions, while πD_2 does not.

When D is inconsistent but e-consistent, then πD is inconsistent too. When D is not e-consistent, then π is incomplete iff πD is consistent. Even in such a case, it is often possible to restore the equivalence between D and πD by extending π as follows. For each e-proposition a **causes** f **if** $p_1, \ldots, p_n$ with f a positive literal, we add the rule:

$$Initiates(a, f, S) \text{ :- } Holds(p_1, S), \ldots, Holds(p_n, S)$$

For each e-proposition a **causes** $\neg f$ **if** $p_1, \ldots, p_n$ with f a positive literal, we add the rule:

$$Terminates(a, f, S) \text{:-} Holds(p_1, S), \ldots, Holds(p_n, S)$$

In addition, we add the integrity constraint:

$$\forall A, F, S : \neg Initiates(A, F, S) \vee \neg Terminates(A, F, S)$$

Example In πD_2 we have two additional rules:

$$Initiates(shoot, alive, S) \text{:-}$$

$$Terminates(shoot, alive, S) \text{:-}$$

It is trivial that the resulting program violates the integrity constraint.

In some interesting situations, this solution does not work. Consider the following example:

Example The domain D_3 is about flipping a (light) switch. There is one action: $switch$ and two fluents, on and off.

$switch$ **causes** on **if** off
$switch$ **causes** off **if** on
$switch$ **causes** $\neg on$ **if** on
$switch$ **causes** $\neg off$ **if** off
Initially on
Initially $\neg off$

D_3 is not consistent, because $\Phi(switch, \{on, off\})$ is not defined consistently. However, starting from the initial situation in which on is true and off is false and applying $switch$ consecutively flips the state of on and off in such a way that on and off are never true in the same state. Hence, from the initial state, the problematic state $\{on, off\}$ can never be reached. For this reason, πD_3 is consistent, even with $Terminating/3$ and $Initiating/3$. Applying the model construction of theorem 3.1, we obtain:

$Initially(on)$	
$Holds(on, switch^n)$	for each even n
$Holds(off, switch^n)$	for each odd n
$Noninertial(on, switch, switch^n)$	for each n
$Noninertial(off, switch, switch^n)$	for each n
$Initiates(switch, on, switch^n)$	for each odd n
$Terminates(switch, on, switch^n)$	for each even n
$Initiates(switch, off, switch^n)$	for each even n
$Terminates(switch, off, switch^n)$	for each odd n

For this example, the semantics of D_3 and πD_3 differ. Which semantics is to be preferred? This is a matter of taste, but intuitively we find the domain description D_3 a sensible theory, and the model a sensible model of the theory. By considering D_3 inconsistent, the semantics of $\mathcal{A}$ is to our taste too severe[3].

A final example illustrates why v-propositions are added as integrity constraints and not as program clauses.

[3] Notice that the inconsistency of D_3 can easily be repaired by dropping the fluent off and replacing it everywhere by $\neg on$. It is unclear to us whether such a solution exists in general when the semantics of D and πD differ.

Example Take the domain D_4:

> a **causes** $\neg f$
> f **after** a

Obviously this domain is inconsistent. πD_4 is also inconsistent: indeed the completed definition of $Holds/2$ subsumes

$$Holds(f, Result(a, s_0)) \leftrightarrow false$$

That contradicts with the integrity constraint $Holds(f, Result(a, s_0))$.

On the other hand, adding $Holds(f, Result(a, s_0))$ as a rule has the effect of adding the disjunct $F = f \wedge T = Result(a, s_0)$ to the completed definition of $Holds$. The resulting theory is consistent and has the model:

$$\{Holds(f, Result(a, s_0)), Noninertial(f, a, t_s) | t_s \in HU(\mathcal{L}_{D_4})\}$$

4 Reasoning on open logic programs

Traditionally, open programs have been associated with abduction as procedural paradigm. We show how other important procedural paradigms such as deduction and proving satisfiability are feasible and can be emulated by a suitable abduction procedure.

In [6] we proposed SLDNFA, an abductive procedure for normal abductive programs. The soundness of SLDNFA has been proven. Two completeness results were proven: if an SLDNFA execution terminates with failure on a query $\leftarrow Q$, then $comp_a(P_D) \models \forall(\neg Q)$; hence no abductive solutions exist. If an SLDNFA execution terminates and generates abductive solutions $\Delta_1, \ldots, \Delta_m$, then for each other abductive solution Δ, there exists a Δ_i which is *more general* than Δ in the sense that a skolem substitution θ exists such that $\theta(\Delta_i) \subseteq \Delta$. If Δ_i contains no skolem constants, this simply means that $\Delta_i \subseteq \Delta$[4].

SLDNFA is not developed for dealing with FOL integrity constraints, but there is a general technique to transform an incomplete program P and a theory of FOL integrity constraints IC into an equivalent incomplete program P'. This transformation technique is a trivial extension of the transformation proposed in [23]. In a first step, P is extended with:

$$false \text{ :- } \neg F$$

for each integrity constraint $F \in IC$. The result is a general incomplete program. In the second step, it is transformed to a normal incomplete program P' using the technique in [23]. The transformation is correct in the sense that $P + IC$ is equivalent with $P' + \{\neg false\}$ according to completion semantics. The remaining integrity constraint $\neg false$ can be added as an extra literal to the query to be solved by the abductive solver. This result shows that FOL (with FEQ) and incomplete logic programming have the same expressivity in the strongest possible sense, namely on the level of logical equivalence. The proof of this result is a trivial extension of the proof in [23].

[4]Any other abductive procedure which satisfies the same completeness results can be used. At the time of writing, SLDNFA seems to have advantages over other published procedures in two senses: it suffers less from floundering since it does not flounder on abducible atoms and more powerful completeness results have been proved for it.

For a domain description D, the transformation of the integrity constraints of πD to an incomplete program is trivial. A ground atom $Holds(F, Result[A_1; \ldots; A_n, S_0])$ is transformed to:

$$false \text{ :- } \neg Holds(F, Result[A_1; \ldots; A_n, S_0])$$

A ground negative literal $\neg Holds(F, Result[A_1; \ldots; A_n, S_0])$ is transformed to:

$$false \text{ :- } Holds(F, [A_1, ; \ldots ; A_n, S_0])$$

Applying this technique on the FOL axioms of the Murder Mystery domain D_1, we obtain an open program P', in which the following rules:

$false \text{ :- } \neg Holds(alive, s_0)$
$false \text{ :- } Holds(alive, [shoot; wait, s_0])$

are substituted for the FOL axioms of πD_1.

An abductive procedure generates explanations for a given observation on the problem domain. Here we can take $\neg false$ as an observation. SLDNFA solves the query $\leftarrow \neg false$ and returns the solution:

$$\Delta_1 = \{Initially(loaded), Initially(alive)\}$$

Not only this gives an explanation for $\neg false$, but proves also that πD_1 is consistent. The precise reasoning goes as follows: the resulting program $P' + \Delta$ is an acyclic program and therefore is consistent wrt completion semantics [1]. $P' + \Delta$ comprises P_{D_1} and entails both $\neg false$ and $\neg false \leftrightarrow IC_{D_1}$. Hence a model of $P' + \Delta$ is a model of πD.

An abductive procedure can also be used for deduction. For example, we want to prove that $\pi D_0 \models Initially(loaded)$ or equivalently that the theory $\pi D + \neg Initially(loaded)$ is inconsistent. To prove that, we add the extra rule :

$$false \text{ :- } Initially(loaded)$$

Now SLDNFA fails finitely on the query $\leftarrow \neg false$. From the first completeness result of SLDNFA, it follows that $\pi D_0 + \neg Initially(loaded)$ is inconsistent. Notice that a completeness result for abduction is used here as a soundness result for deduction.

An abductive procedure allows reasoning under uncertainty. By dropping **Initially** *alive* from the Murder Mystery domain D_1, an incomplete domain description D_1' is obtained. Using $\pi D_1'$, SLDNFA answers the goal $\leftarrow \neg false$ by returning the answer $\Delta_2 = \{\}$. The original solution Δ_1 is still a solution but is not generated. This does not conflict with the completeness result of SLDNFA because $\Delta_2 \subset \Delta_1$.

That we have uncertainty in this domain description becomes obvious when we want to know whether **Initially** *alive* is *possible* according to D_1'. This is done by posing the query $\leftarrow \neg false, Initially(alive)$. SLDNFA proves that **Initially** *alive* is possible by returning Δ_1.

Deduction under uncertainty is possible. Observe that D_1' entails:

$$\textbf{Initially } \neg alive \vee \textbf{Initially } loaded$$

SLDNFA can prove this. This is done by transforming the negation of the disjunction to:

$$false \text{ :- } Initially(loaded)$$

$$false \text{ :- } \neg Initially((alive)$$

After adding these rules to πD_1, SLDNFA fails finitely on $\leftarrow \neg false$. This proves the disjunction.

The above experiments show in the first place that though open/ abductive logic programs are traditionally associated with abduction as procedural paradigm, other procedural paradigms such as deduction and consistency proving are of interest. This illustrates our argument that an *abductive program* is better called an *open program*. In the second place, the experiments show that a suitable abductive procedure can be used to emulate these paradigms.

5 The Gelfond & Lifschitz approach

We recall the transformation proposed in [17], from $\mathcal{A}$ domain descriptions to extended programs. For any domain description D, $\pi_{GL}D$ is defined as the extended logic program containing the following extended clauses:

- Four inertia rules:

$$Holds(F, Result(A,S)) \leftarrow Holds(F,S), not\ Noninertial(F,A,S) \quad (1')$$
$$\neg Holds(F, Result(A,S)) \leftarrow \neg Holds(F,S), not\ Noninertial(F,A,S) \quad (2')$$
$$Holds(F,S) \leftarrow Holds(F, Result(A,S)), not\ Noninertial(F,A,S) \quad (3')$$
$$\neg Holds(F,S) \leftarrow \neg Holds(F, Result(A,S)), not\ Noninertial(F,A,S) \quad (4')$$

- Each v-proposition f **after** $a_1; \ldots; a_n$, is transformed into:

$$Holds(f, Result[a_1; \ldots; a_n, s_0]) \quad (5')$$

  Recall that $Holds(\neg f, \ldots)$ denotes $\neg Holds(f, \ldots)$.

- Each e-proposition a **causes** f **if** $p_1, \ldots, p_n$ is translated into 2n+2 rules. Below, $\overline{Holds(f,S)}$ denotes the complement of $Holds(f,S)$ with respect to $\neg$.

$$Holds(f, Result(a,S)) \leftarrow Holds(p_1,S), \ldots, Holds(p_n,S) \quad (6')$$
$$Noninertial(|f|, a, S) \leftarrow not\ \overline{Holds(p_1,S)}, \ldots, not\ \overline{Holds(p_n,S)} \quad (7')$$

  For each i, $1 \leq i \leq n$:

$$Holds(p_i,S) \leftarrow \overline{Holds(f,S)}, Holds(f, Result(a,S)) \quad (8')$$
$$\overline{Holds(p_i,S)} \leftarrow \overline{Holds(f, Result(a,S))}, Holds(p_1,S), \ldots, Holds(p_{i-1},S), Holds(p_{i+1},S), \ldots, Holds(p_n,S) \quad (9')$$

[17] gives the intuition behind the translation and gives a soundness theorem for all domain descriptions D provided D does not contain *similar* e-propositions, i.e. e-propositions which only differ by the preconditions. A comparison of $\pi_{GL}D$ with πD is of interest. Observe that if the two negations *not* and $\neg$ in extended programs are mapped both on "$\neg$" in open programs, we find clauses or formulas in πD corresponding to (1′), (5′), (6′) (if f is a positive literal) and (7′), while (2′), (3′), (4′), (8′) and (9′) lack in πD.

A striking fact is that $\pi_{GL}D$ contains four inertia rules instead of one in πD. (1′) and (2′) are *forward persistence rules* for respectively positive and negative fluents. (3′) and (4′)

are *backward persistence rules* for again positive and negative fluents. Clearly (2′), (3′) and (4′) are natural rules, which are expected to hold in any correct formalisation. Therefore, they must be subsumed by πD, otherwise π could never be sound and complete. As a matter of fact, it is straightforward to prove that for each of the extended rules in $\pi_{GL} D$, the corresponding clause is subsumed by $comp_a(P_D)$, where P_D is the logic program part of πD. For example, notice that from the classical logic point of view the rules (1′) and (4′) are equivalent and so are the rules (2′) and (3′). This immediately gives that $comp_a(P_D)$ subsumes (4′). Clauses corresponding to (2′) and (3′) can be derived from the completed definition (6.6) of $Holds/2$ in P_D. Substitute $Result(A, S)$ for T. After simplification one obtains:

$$\forall F, A, S : Holds(F, Result(A, S)) \leftrightarrow E_1 \vee \ldots \vee E_n$$

where E_1 is of the form:

$$Holds(F, S) \wedge \neg Noninertial(F, A, S)$$

and for each e-proposition a **causes** f **if** $p_1, \ldots, p_n$ with f is a positive literal, there is an E_i of the form:

$$F = f \wedge A = a \wedge Holds(p_1, S) \wedge \ldots \wedge Holds(p_n, S)$$

Now, it is easy to see that $comp_a(P_D)$ satisfies:

$$\forall F, A, S : F = f \wedge A = a \wedge Holds(p_1, S) \wedge \ldots \wedge Holds(p_n, S) \Rightarrow Noninertial(F, A, S)$$

By dropping $\neg Noninertial(F, A, S)$ from the first disjunct and substituting $Noninertial(F, A, S)$ for the other disjuncts, we find:

$$\forall F, A, S : Holds(F, Result(A, S)) \Rightarrow Holds(F, S) \vee Noninertial(F, A, S)$$

Simple rewriting gives formulas corresponding to (2′) and (3′).

A shortcoming of π_{GL} is its incompleteness. [17] gives the following example D_5:

a **causes** f **if** f
f **after** a

Clearly D_5 entails **Initially** f. However, $Initially(f)$ is not entailed by $\pi_{GL} D_5$. On the other hand, notice that D_5 is e-consistent. Therefore, πD_5 implies $Initially(F)$ and SLDNFA can prove that.

Another problem of π_{GL} shows up when $\mathcal{A}$ is extended to allow predicates. Consider the following rule:

$$Pick(X, Obj) \textbf{ causes } thief(X) \textbf{ if } owner(Y, Obj), X \neq Y$$

which says that X becomes a thief if he picks an object Obj which he does not own. The translation to open programs does not require any modification. π produces:

$Holds(thief(X), Result(Pick(X, Obj), S))$:- $Holds(owner(Y, Obj), S), X \neq Y$
$Noninertial(thief(X), Pick(X, Obj), S)$:- $Holds(owner(Y, Obj), S), X \neq Y$

For π_{GL}, there are problems with the rules of type (8′):

$$
\begin{aligned}
Holds(owner(Y,Obj),S) &\leftarrow \neg Holds(thief(X),S),\\
&\qquad Holds(thief(X),Result(Pick(X,Obj),S))\\
X \neq Y &\leftarrow \neg Holds(thief(X),S),\\
&\qquad Holds(thief(X),Result(Pick(X,Obj),S))
\end{aligned}
$$

These rules say that when X becomes thief by picking something in situation S, then each Y is owner at situation S and no Y is equal to X. This is a contradiction. The problem is that Y should not be universally but existentially quantified. The following formulas are subsumed by πD but are not extended clauses:

$$
\begin{aligned}
\forall Obj,S,X:\exists Y: Holds(owner(Y,Obj),S) &\leftarrow \neg Holds(thief(X),S),\\
&\qquad Holds(thief(X),Result(Pick(X,Obj),S))\\
\forall Obj,S,X:\exists Y: X \neq Y &\leftarrow \neg Holds(thief(X),S),\\
&\qquad Holds(thief(X),Result(Pick(X,Obj),S))
\end{aligned}
$$

The translation π to open programs is superior to the translation π_{GL} to extended programs. π_{GL} creates a higher number of rules, is incomplete, suffers from problems with similar e-propositions and is not directly extendible to the predicate case. The open program approach seems more understandable because only one negation occurs, is sound even with similar e-propositions, is complete for all reasonable domain descriptions and applies without modification for the predicate case. Proofs are easy compared with the proofs in [17].

6 Dung's approach

Independent from the work presented here, another approach, quite similar to ours in a number of aspects, has been developed for translating $\mathcal{A}$ domain descriptions to a logic program formalism. This work, described in [11], maps $\mathcal{A}$ to the logic program formalism and defines a special purpose semantics, similar to the completion semantics.

On the syntactical level, the most important difference with our approach is the symmetrical treatment of fluent symbols f and their negation $\neg f$. The translation $\pi_{Du}D$ contains our frame axiom (6.2), and contains for each e-proposition a **causes** f **if** $p_1,\ldots,p_n$ (f positive or negative) the following rules:

$$
\begin{aligned}
&Holds(f,Result(a,S)) \text{:- } Holds(p_1,S),\ldots,Holds(p_n,S)\\
&Noninertial(f,a,S) \text{:- } Holds(p_1,S),\ldots,Holds(p_n,S)\\
&Noninertial(f^*,a,S) \text{:- } Holds(p_1,S),\ldots,Holds(p_n,S)
\end{aligned}
$$

Here f^* denotes the complement of f. Contrary to our approach, here fluent literals like $\neg f$ appear within $Holds/2$. Each v-proposition f **after** $a_1;\ldots;a_n$ is transformed to the denial:

$$\leftarrow Holds(f^*,Result(a_1;\ldots;a_n,s_0))$$

In addition, for each fluent symbol f the following constraints are added:

$$
\begin{aligned}
&\leftarrow Holds(f,S),Holds(\neg f,S)\\
&Holds(\neg f,s_0) \leftrightarrow \neg Holds(f,s_0)
\end{aligned}
$$

These additional constraints are necessary due to the symmetrical treatment of a fluent f and its negation $\neg f$. In contrast, our transformation maps $\mathcal{A}$ domains to a more classical situation calculus, which treats the positive and negative literals asymmetrically. Considering only

Dungs solution, one might wonder whether the symmetrical treatment is crucial for solving the frame problem in logic programming or whether it has advantages. Our transformation shows that a symmetrical treatment is not necessary and leads to substantially more rules.

The semantics of $\pi_{Du}D$ is defined via a domain dependent variant of the completion semantics. It contains FEQ and the normal completed definition of $Noninertial/3$, but a specialised version of the completed definition of $Holds/2$. $\pi_{Du}D$ does not contain rules with $Holds(F, s_0)$ in the head. As a consequence the standard completion would imply that

$$\forall F : \neg Holds(F, s_0)$$

[11] avoids this by the following alternative:

$\forall F, A, T : Holds(F, Result(A, S)) \leftrightarrow E_1 \vee \ldots \vee E_n$

with:

$$\begin{array}{ll} E_1 = & Holds(F, S) \wedge \neg Noninertial(F, A, S) \\ E_i(i > 1) = & F = f \wedge A = a \wedge Holds(p_1, S) \wedge \ldots \wedge Holds(p_n, S) \end{array}$$

such that precisely for each e-proposition a **causes** f **if** $p_1, \ldots, p_n$ (f positive or negative), there exists one disjunct E_i ($i > 1$) in the completed definition. This formula says nothing about $Holds(F, s_0)$, and therefore, we get a similar semantics as in our approach, but without $Initially/1$.

Dung extends $\mathcal{A}$ to the predicate case and gives an application for integrity checking of a database update. For this purposes, a predicate domain description D is developed to represent the effects of primitive and compound update operations. An unfolding partial evaluation procedure with constructive negation can then be applied on $\pi_{Du}(D)$ to check the consistency of one or more integrity constraints in the database after the update. The use of a similar unfolding procedure has been proposed earlier in the context of abductive logic programming by [3]. As Dung here, Console et al. use a general unfolding procedure to reduce a given query by unfolding all defined literals until a first order logic formula is obtained, containing only abductive literals. The resulting formula is equivalent with the original query and the equivalence with in the left the query and in the right the computed formula is called the explanation formula. Though such a procedure can be used to solve abductive problems, it is much weaker than an abductive procedure like SLDNFA because naive unfolding cannot guarantee the consistency of the righ hand of the explanation formula. In [5, 9], we show that SLDNFA can be interpreted also as a kind of an unfolding procedure, but SLDNFA is equiped with a mechanism for checking the consistency of each of the disjuncts at the right hands of the generated explanation formula. In our opinion, this is absolutely necessary because even for simple problems, an explanation formula can be so complex that it is impossible for humans to grasp its meaning and to check its consistency.

An interesting statement in [11] is related to the following example D_6 (a syntactical simplification of the example in theorem (8) in [11]):

Initially f
a **causes** $\neg f$ **if** g

Dung observes correctly that the Gelfond and Lifschitz transformation $\pi_{GL}D_6$ has two answer sets:

$$Z_1 = \{Holds(f, s_0)\} \cup \{Noninertial(f, a, a^n) \mid n \in \mathbb{N}\}$$
$$Z_2 = \{Holds(f, a^n), \neg Holds(g, a^n) \mid n \in \mathbb{N}\}$$

In Z_1, $Holds(f, a^n)$ $(n > 0)$ and $Holds(g, a^n)$ $(n \geq 0)$ are unknown since neither the atom nor its negation appears in Z_1. Z_2 corresponds to a two-valued model, obtained by having f initially true and g false, a situation which is preserved when applying a. Then Dung argues that *"it is obvious that only the first solution captures the intended semantics of D_6, for if we don't know anything about g, it is impossible to say anything about the outcome of a"*. Remarkable now is that Z_2 corresponds to a model of Dung's $\pi_{Du}D_6$.

Here we touch a fundamental issue in knowledge representation: how should incomplete knowledge be formalised in a logic? What Dung suggests in this statement, and what is probably the original intention of Gelfond and Lifschitz, is that a model of a theory should reflect what one knows about the world: a model should be the set of atoms which are known to be true. Then indeed, Z_2 is incorrect, since we cannot know for sure that g is not true initially and hence f is possibly terminated by applying a.

This is not the classical logic view on incomplete knowledge. In the classical logic view, a model is a mathematical abstraction of a possible state of the world, not the set of ground atoms which are known by or provable from the theory. To have incomplete knowledge on g in the initial state, is reflected by the fact that there are models in which g is initially true and others in which g is initially false. In a model in which g is initially true, f is necessarily terminated after applying a. When g is initially false, as in Z_2, f remains true. It turns out that the two transformations π and π_{Du} behave correct under this view. Note that D_6 has two models, one with initial situation $\{f\}$ and another with initial situation $\{f, g\}$. These models correspond to models of πD_6 and, as Dung perhaps has not noticed, with models of $\pi_{Du}D_6$. Only for $\pi_{GL}D_6$, the second model is weakened to the answer set Z_1.

Of the two views, the classical view is definitely the richest one. Indeed, consider the following formulas:

$$\neg Holds(g, s_0) \rightarrow [Holds(f, S_0) \rightarrow Holds(f, Result(a, S_0))]$$

$$\neg Holds(g, S_0) \rightarrow [\neg Holds(f, S_0) \rightarrow \neg Holds(f, Result(a, S_0))]$$

$$Holds(g, S_0) \rightarrow \neg Holds(f, Result(a, S_0))$$

$$Holds(f, Result(a, S_0)) \vee Holds(g, Result(a, S_0))$$

Surely one will agree that they are intuitively right. As a matter of fact, they are true in all models of πD_6 and $\pi_{Du}D_6$ and hence, they are implied by these theories. SLDNfA can prove each of them in πD_6. On the other hand, they are not true in the answer set Z_1, and hence $\pi_{GL}D_6$ does not imply them. The problem with the incompleteness of π_{GL} is not due to models like Z_2 but due to models like Z_1.

Though Dung does not investigate this, his technique for representing uncertainty can be generalised to other applications in a rather straithforward way. The resulting formalism would be a generalisation of the abductive logic program formalism, in which it is possible to give definitions at the level of atoms instead of predicates. Such a generalised abductive program might be seen as a tuple of a set of defined atoms $D = \{p_1(\bar{t}_1), \ldots, p_n(\bar{t}_n)\}$ and a set of normal clauses such that if two atoms $p(\bar{t}), p(\bar{s})$ occur in D, then they do not unify and if $p(\bar{s})$:- $L_1, \ldots, L_n$ belongs to the program, then $p(\bar{s})$ should be an instance of an atom $p(\bar{t}) \in D$. Such a clause can be viewed as a defining clause for $p(\bar{t})$.

The semantics of such a generalised open program could then be given by FEQ and for each defined atom $p_i(\bar{t}_i)$ a completed definition $p_i(\bar{t}_i) \leftrightarrow \ldots$ constructed using its defining clauses.

However, this generalisation does not produce extra expressivity, as the formalism can be easily and elegantly translated to the open logic program formalism. Given such a generalised logic program P, we can translate P to an open logic program P' obtained by adding for each predicate p/n one new undefined predicate $undefined\_p/n$ and taking the clauses of P and adding for each predicate p/n with defined atoms $p(\bar{t}_1), \ldots, p(\bar{t}_n)$ one new clause:

$$p(\overline{X}) :\text{-} \neg(\exists \overline{Y}_1 . \overline{X} = \bar{t}_1), \ldots, \neg(\exists \overline{Y}_n . \overline{X} = \bar{t}_n), other - p(\overline{X})$$

Here $\overline{Y}_i$ is the set of variables $var(\bar{t}_i)$ and $\overline{X} = \bar{t}_i$ denotes the conjunction of equality atoms $X_j = t^i_j$. The completion of P' can easily be proven to be a conservative extension of P.

Though our approach was developed independently of Dung's approach, it clearly shows similarities it. Both generate a form of situation calculus using the logic program syntax, and the semantics of the resulting programs is given by first order logic theories, constructed using different forms of completion of sets of clauses.

Despite the similarities, we believe that our work contributes in several important aspects. Recall our two main goals of this experiment with $\mathcal{A}$: first, to show the role of open/abductive logic programs as a general purpose logic for representing uncertainty and second, to show the role of a suitable abductive procedure (as SLDNFA) for solving different computational problems, including deduction under uncertainty, abduction or the generation of explaining hypotheses and consistency proving. To both goals, Dung's paper does not really contribute.

With respect to the first goal, Dung proposes a novel formalism. He does not relate this formalism to open/abductive logic programming. As argued in section 3, we could prove the correctness of the transformation π not only wrt completion semantics but wrt 3-valued completion semantics, generalised stable model semantics, generalised well-founded semantics, justification semantics, etc. This aspect is necessarily lacking in Dung's paper, since he doesn't consider abductive logic programming in the first place.

With respect to the second goal, we have shown how SLDNFA can be used to solve three important types of computational problems:

- deduction (under incertainty)
- abduction
- consistency proving

Dung does not consider these classical forms of reasoning but focusses on the use of an unfolding partial evaluation procedure for integrity checking of database updates. The question whether such an unfolding procedure is of practical use for general problem solving in temporal domains is not dealt with in Dung's paper.

7 Discussion

In [12], a First Order Logic solution to the frame-problem was proposed. [29] uses the same type of theory to formalise database evolution. This type of theory is a form of situation calculus which shows a strong similarity with the completion of a program πD. $Result/2$ is replaced by $do/2$. Instead of using the meta predicate $Holds/2$, [29] adds one additional argument to each fluent predicate; i.e. an atom $Holds(p(x), t)$ is contracted to the atom $p(x, t)$. As a consequence the law of inertia has to be stated for each fluent. An example taken from [29] is given below:

$$\begin{aligned}\forall St, C, A, S : Poss(A, S) \rightarrow (enrolled(St, C, do(A, S)) \leftrightarrow \\ A = register(St, C) \vee \\ enrolled(St, C, S) \wedge A \neq drop(A, C))\end{aligned}$$

The rule says that when action A may be executed in situation S ($Poss(A,S)$), then student St is enrolled in course C at time $do(A,S)$ iff A is an action of registering St in C or, St was enrolled at S and A is not an action of dropping St from the course C.

As said above, this type of theory shows a strong similarity with the completion of the open programs produced by π and with the completions of the logic programs found in [14] and [1]. If we forget about $Poss(A,S)$, and introduce *Result* and *Holds*, we find:

$$\begin{aligned}\forall St, C, A, S : Holds(enrolled(St,C), Result(A,S)) \leftrightarrow \\ A = register(St,C) \vee \\ Holds(enrolled(St,C),S) \wedge A \neq drop(St,C)\end{aligned}$$

Similar formulas are subsumed by πD. The first disjunct corresponds to a rule initiating $enrolled(St,C)$ by $enregister(St,C)$. The second disjunct corresponds to the law of inertia, with $\neg Noninertial(enrolled(St,C),A,S)$ replaced by its definition: $drop(St,C)$ is the only action which terminates $enrolled(St,C)$[5].

It is remarkable and frustrating that the above monotonic solution in classical logic has not been discovered much earlier. Some people have been experimenting with situation calculus in logic programming ever since the first experiments in Kowalski's *Logic for problem solving* [21] (first edition in 76). That Prolog with negation as finite failure can be interpreted as a sound theorem prover wrt to the completion of a Prolog program is known since Clark's work [2] in 78. When [18] introduced the YTS problem to show the failure of several nonmonotonic solutions to the frame problem, it was soon realised by some in the logic programming community that the Prolog solution behaved perfectly correct. However, if Prolog can prove that *waiting* does not unload the gun and that the turkey is *dead* after *loading*, *waiting* and *shooting*, then from Clark's soundness theorem it follows immediately that the completion of the Prolog program entails these conclusions also, and hence that the completion of the Prolog program provides a (monotonic) solution to the frame problem. Generalising this example, it is a relatively simple step to find that the principle of completion of implications gives an elegant and general solution to the frame problem. Yet, it seems that for many years, nobody, neither in the logic programming community nor in the A.I. community has come to this obvious conclusion or was interested in it.

In the past, another approach has been explored for temporal reasoning, based on event calculus [22]. [13] and [30] have simplified event calculus and have extended it with abduction for the purpose of planning. [30] extended event calculus to deal with necessary preconditions of actions. [25] implemented a planning system based on this formalism. Other work has been done to extend event calculus with continuous actions [31] and time granularity [15], [26]. Recently [10] applied abductive event calculus to solve a number of benchmark problems in temporal reasoning, such as the Murder Mystery, the Stolen Car problem, the Walking Turkey Shooting problem and the Russian Turkey Shooting problem. The latter problem contains an indeterminate action. Situation and event calculus seem two non-equivalent ways of representing time and action. $\mathcal{A}$ domain descriptions cannot (easily) be translated to event calculus, because $\mathcal{A}$ assumes a *branching time* philosophy, as in the situation calculus, whereas event calculus assumes a linear time philosophy. A deep analysis of situation versus event calculus is beyond the scope of the chapter.

The $\mathcal{A}$ language is designed to represent one form of incomplete temporal knowledge: on the initial situation. A totally different form of incomplete knowledge appears with indeterminate actions. Our transformation from $\mathcal{A}$ to open programs can easily be adapted in order to deal with such actions, which shows again the expressivity of the open program

[5] Here the version of $Noninertial/3$ is needed which contains only rules for terminating effects of actions.

formalism for representing incertainty. It turns out that the technique used in [10] to represent indeterminate actions in the context of event calculus can easily be translated to situation calculus. The Russian Turkey Shooting problem is a variant of the Yale Turkey Shooting problem in which one additional action *spinning* of spinning the gun's chamber occurs. The effect is that the gun is possibly unloaded. Below we allow e-propositions of the form a **possibly causes** f **if** $p_1, \ldots, p_n$. The problem is formalised as follows:

Initially *alive*
Initially *loaded*
load **causes** *loaded*
shoot **causes** $\neg$*alive* **if** *loaded*
shoot **causes** $\neg$*loaded*
spinning **possibly causes** $\neg$*loaded*

The semantics of $\mathcal{A}$ should be adapted. While in $\mathcal{A}$, a successor state is completely determined by the action and the previous state, this is not the case with indeterminate actions. Therefore, in the extended version the transition function should be replaced by a transition relation. While it is beyond the scope of this paper to work out this semantics in detail, it is easy to show how the transformation could be adapted to model this kind of indeterminate actions. The indeterminism can be captured by introducing an undefined *Good_luck*/2 predicate:

$$Noninertial(loaded, spinning, S) \text{:- } Good\_luck(spinning, S)$$

The above clause has the effect that the rule of inertia is disabled for *loaded* iff good luck occurs at the spinning action in state S. In general, for each clause

$$a \textbf{ possibly causes } f \textbf{ if } p_1, \ldots, p_n$$

the following rule must be introduced:

$$Noninertial(|f|, a, S) \text{:- } Holds(p_1, S), \ldots, Holds(p_n, S), Good\_luck(a, S)$$

For a positive f, in addition the following rule is added:

$$Holds(f, Result(a, S)) \text{:- } Holds(p_1, S), \ldots, Holds(p_m, S), Good\_luck(a, S)$$

8 Summary

We presented a sound and complete transformation π from $\mathcal{A}$ domains to open logic programs with FOL axioms. We have illustrated the use of SLDNFA for abductive and deductive reasoning and satisfiability proving under uncertainty. The transformation of Gelfond and Lifschitz is more complex, is not complete, is only sound for domains without e-similar actions and cannot be extended to the predicate case (at least not without imposing other syntactic constraints). Moreover, no reasoning procedure is currently described for the resulting programs. Dung's approach is in some aspects similar to ours and provides a reasoning procedure, but is still more complex than ours, has the disadvantage of relying on a special purpose logic and does not show the application of the reasoning procedure for classical forms of reasoning such as deduction, abduction and consistency proving.

We have investigated also a number of typical temporal reasoning issues. Although in πD only forward persistence axioms are contained, the completion of πD subsumes backward persistence axioms. We have also shown how to extend $\mathcal{A}$ with indeterminate actions.

From a more general perspective, this work can be viewed as a -successful- experiment in the declarative representation of and diverse forms of automated reasoning on incomplete knowledge using open logic programming and an abductive procedure.

References

[1] K.R. Apt and M. Bezem. Acyclic programs. In *Proc. of the International Conference on Logic Programming*, pages 579–597. MIT press, 1990.

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

[3] L. Console, D. Theseider Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[4] M. Denecker. *Knowledge Representation and Reasoning in Incomplete Logic Programming.* PhD thesis, Department of Computer Science, K.U.Leuven, 1993.

[5] M. Denecker and D. De Schreye. On the duality of abduction and model generation. In *Proc. of the International Conference on Fifth Generation Computer Systems*, 1992.

[6] M. Denecker and D. De Schreye. SLDNFA; an abductive procedure for normal abductive programs. In K.R. Apt, editor, *Proc. of the International Joint Conference and Symposium on Logic Programming*, pages 686–700, 1992.

[7] M. Denecker and D. De Schreye. Justification semantics: a unifying framework for the semantics of logic programs. In *Proc. of the Logic Programming and Nonmonotonic Reasoning Workshop*, pages 365–379, 1993.

[8] M. Denecker and D. De Schreye. Representing incomplete knowledge in abductive logic programming. In *Proc. of the International Symposium on Logic Programming*, pages 147–163, 1993.

[9] M. Denecker and D. De Schreye. On the Duality of Abduction and Model Generation in a Framework for Model Generation with Equality. *Journal of Theoretical Computer Science*, 122(1&2), 1994.

[10] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *Proc. of the European Conference on Artificial Intelligence*, 1992.

[11] P.M. Dung. Representing Actions in Logic Programming and its Applications in Database Updates. In *Proc. of the International Conference on Logic Programming*, 1993.

[12] C. Elkan. Reasoning about Action in First-Order Logic. In *Proc. of the CSCSI-92*, 1992.

[13] K. Eshghi. Abductive planning with Event Calculus. In R.A. Kowalski and K.A. Bowen, editors, *Proc. of the International Conference on Logic Programming*, 1988.

[14] C. Evans. Negation as failure as an approach to the Hanks and McDermott problem. In *Proc. of the second International Symposium on Artificial Intelligence*, 1989.

[15] C. Evans. The Macro-Event Calculus: Representing Temporal Granularity. In *Proc. of PRICAI, Tokyo*, 1990.

[16] M. Gelfond and V. Lifschitz. Logic Programs with Classical Negation. In D.H.D. Warren and P. Szeredi, editors, *Proc. of the 7th International Conference on Logic Programming 90*, page 579. MIT press, 1990.

[17] M. Gelfond and V. Lifschitz. Describing Action and Change by Logic Programs. In *Proc. of the 9th Int. Joint Conf. and Symp. on Logic Programming*, 1992.

[18] S. Hanks and D. McDermott. Default reasoning, nonmonotonic logic, and the frame problem. In *Proc. of the National Conference on Artificial Intelligence, Philadelphia*, pages 328–333, 1986.

[19] P. M. Hill and J. W. Lloyd. Analysis of meta-programs. In H. D. Abramson and M. H. Rogers, editors, *Proceedings of Meta88*, pages 23–51. MIT Press, 1989.

[20] A.C. Kakas and P. Mancarella. Generalised stable models: a semantics for abduction. In *Proc. of the European Conference on Artificial Intelligence*, 1990.

[21] R.A. Kowalski. *Logic for problem solving.* Elsevier Science Publisher, 1976.

[22] R.A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(4):319–340, 1986.

[23] J.W. Lloyd and R.W. Topor. Making prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[24] B. Martens and D. De Schreye. A perfect Herbrand semantics for untyped vanilla meta-programming. In K.R. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 511–525, Washington, November 1992. MIT Press.

[25] L.R. Missiaen. *Localized abductive planning with the event calculus.* PhD thesis, Department of Computer Science, K.U.Leuven, 1991.

[26] A. Montanari, E. Maïm, E. Ciapessoni, and E. Ratto. Dealing with Time Granularity in the Event Calculus. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 702–712, 1992.

[27] L.M. Pereira, J.N. Aparicio, and J.J. Alferes. Hypothetical Reasoning with Well Founded Semantics. In B. Mayoh, editor, *Proc. of the 3th Scandinavian Conference on AI.* IOS Press, 1991.

[28] C.S. Pierce. *Philosophical Writings of Pierce.* Dover Publications, New York, 1955.

[29] R. Reiter. Formalizing Database Evolution in the Situation Calculus. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 600–609, 1992.

[30] M. Shanahan. Prediction is deduction but explanation is abduction. In *Proc. of the IJCAI89*, page 1055, 1989.

[31] M. Shanahan. Representing continuous change in the event calculus. In *Proc. of the European Conference on Artificial Intelligence*, page 598, 1990.