Inductive Definability and the Situation Calculus

Eugenia Ternovskaia

Department of Computer Science, University of Toronto
eugenia@cs.toronto.edu

Abstract. We explore the situation calculus within the framework of inductive definability. A consequence of this view of the situation calculus is to establish direct connections with different variants of the μ-calculus [Park, 1970; Hitchcock and Park, 1973; Pratt, 1981; Kozen, 1983; Emerson and Clarke, 1980], structural operational semantics of concurrent processes [Plotkin, 1981], and logic programming [Apt, 1990]. First we show that the induction principle on situations [Reiter, 1993] is implied by an inductive definition of the set of situations. Then we consider the frame problem from the point of view of inductive definability and by defining fluents inductively we obtain essentially the same form of successor state axioms as [Reiter, 1991]. Our approach allows extending this result to the case where ramification constraints are present. Finally we demonstrate a method of applying inductive definitions for computing fixed point properties of GOLOG programs.

1 Introduction

Inductive definitions [Aczel, 1977; Moschovakis, 1974] are broadly used in mathematical logic for defining sets. A basis rule defines the simplest element of the set, and an inductive rule or rules build larger elements of the set from smaller ones. Recently inductive definitions have become popular in computer science [Cousot and Cousot, 1992]. Our main motivation for presenting the situation calculus within the general framework of inductive definability arises from the close connection between an inductive definition and a logic program. On one hand, a logic program can be considered as an inductive definition. On the other hand, a positive inductive definition can be encoded as a logic program. Thus, inductive definability provides a basis for logic programming implementations of situation calculus theories.

Situation calculus is a many-sorted language for specifying change. Its foundational axioms are similar to the axioms of Peano arithmetics with the difference that the successor function is parameterized with performed actions. Inductive definability has been extensively used in the situation calculus. Typical examples are the inductive principle on situations, the inductive definition of partial ordering $\leq$ on situations [Reiter, 1993], and the inductive definition of the transition relation $Trans$ [De Giacomo *et al.*, 1997b]. We describe the general concept of inductive definability for the situation calculus and explore this theme in several directions.

First, we show that an inductive definition of the set of situations implies the general induction principle on situations [Reiter, 1993]. Then we provide an additional justification of Reiter's frame problem solution [Reiter, 1991] as we obtain essentially the same form of successor state axioms by defining fluents inductively. Reiter shows that under a natural condition on a situation calculus theory, general effect axioms together with explanation closure axioms, are logically equivalent to successor state axioms. We show that essentially the same successor state axioms can be obtained by an inductive definition of a fluent as a set of situations. We extend this result to the case of indirect causal dependencies. The main idea of our approach to the frame and ramification problems is to treat causal rules specifying direct and indirect effects of actions as rules of inductive definitions.

Then we briefly describe a high-level robot programming language GOLOG [Levesque *et al.*, 1996]. We represent a GOLOG program as a term in the language of the situation calculus. The main constructs include all basic constructs of an imperative programming language: sequence, conditionals, loops, and recursion. However, unlike most conventional programming languages, the primitive operations are actions that in most cases operate in the external world. Moreover, the user can choose primitive actions on an appropriate level of abstraction and specify their preconditions and effects by domain dependent axioms of the situation calculus. The interpreter of GOLOG is a logic programming implementation of the inductive definition of the transition relation $Trans$. This relation specifies transitions performed by a GOLOG program [De Giacomo *et al.*, 1997b].

Finally, we discuss the verification of GOLOG programs. Inductive definability allows us to demonstrate that a strong connection exists between the situation calculus and process algebra (see e.g. [Milner, 1989])[1] on one hand, and between the situation calculus and the propositional μ-calculus [Kozen, 1983] on the other. This connection makes the situation calculus a rigid formal environment for reasoning about GOLOG programs. We can express temporal properties of programs such as safety and liveness (cf. [Emerson, 1990]), as well as different process equivalences such as bisimulation (cf. [Milner, 1989] or [Baeten and Weijland, 1990]). As an example of program verification, a property of a simple GOLOG program is inductively defined and then proved by reducing it to computation of the least fixed point of a logic program.

2 Inductive Definitions

Let U be a set. A *rule-based inductive definition* is a set $\Re$ of rules of the form $\frac{P}{c}$, where P is the set of premises and c is the conclusion, $P \subseteq U$, $c \in U$. A set $Z \subseteq U$ is $\Re$-closed if each rule in $\Re$ whose premises are in Z also has its conclusion in Z. A set H, *inductively defined by* $\Re$, is given by $H = \bigcap\{Z \mid Z \text{ is } \Re\text{-closed}\}$ or by

[1] This connection follows from recent work [De Giacomo *et al.*, 1997b] where we describe the behavior of non-terminating GOLOG programs in the language of the situation calculus.

$H = \bigcup\{Z \mid Z$ is $\Re$-closed$\}$. The former is called a *positive inductive* definition of H, the latter is called a *negative inductive* or *coinductive* definition of H. An *operator induced by an inductive definition* is a total mapping $\Gamma : Power(U) \mapsto Power(U)$, such that

$$\Gamma(Z) = \{c \in U \mid \exists P \subseteq Z \ : \ \frac{P}{c} \in \Re\}.$$

That is, Γ is a mapping taking sets to sets.

A logic program [Apt, 1990] can be understood as a positive inductive definition [Cousot and Cousot, 1992]. Let P be a logic program containing at least one constant, $\mathcal{H}_P$ be its Herbrand universe and $ground(P)$ be the set of all ground instances over $\mathcal{H}_P$ of clauses in P. The set of inductive rules is

$$\Re_{\mathrm{P}} = \left\{ \frac{B_1, ..., B_n}{A} \middle| A \longleftarrow B_1, ..., B_n \ \in \ ground(P) \right\}.$$

The operator induced by this inductive definition is the immediate consequence operator T_P.

Next we will discuss the logical representation of a particular kind of inductive relations, namely those definable as fixed points of recursive equations.[2] We work within classical second order logic. Below, we demonstrate that a fragment of this language which is equivalent to first order logic plus fixed points is sufficient to solve the main representation problems of the situation calculus. Moreover, our inductive definitions often amount to a first order representation. Every property definable as a fixed point must have, by definition,

- its own *construction principle*, that is a recursive equation for which the property is a fixed point;
- an appropriate *induction principle* (respectively, *coinduction principle*) to guarantee the minimality (respectively, maximality) of the solution of the recursive equation.

To define a set Z, we need to say what its elements are. The construction principle tells us how to obtain these elements recursively.

$$\Phi(Z, \boldsymbol{x}) \equiv Z(\boldsymbol{x}). \tag{1}$$

In this case Φ is called a *constructor* for $Z(\boldsymbol{x})$. Any solution of this recursive equation is called a *fixed point* of $\Phi(Z, \boldsymbol{x})$.

To guarantee that Z is the smallest solution, we apply the induction principle:[3]

$$\forall P, \boldsymbol{x}.\{\forall \boldsymbol{y}.[\Phi(P, \boldsymbol{y}) \supset P(\boldsymbol{y})] \supset [Z(\boldsymbol{x}) \supset P(\boldsymbol{x})]\}, \tag{2}$$

i.e., whatever solution $P(\boldsymbol{y})$ of the recursive specification we take, $Z(\boldsymbol{x})$ is a smaller or equal one. The first implication can be replaced with an equivalence.

[2] In general, an inductive relation can be understood as a relation defined as a fixed point with individual constants (parameters) substituted for some of the variables.

[3] The terms construction and induction are from [Hehner, 1993].

A set $Z(\boldsymbol{x})$ satisfying the construction principle (1) and the induction principle (2) is denoted by $\mu Z(\boldsymbol{x}).\Phi(Z,\boldsymbol{x})$. It is called a *least fixed point* of $\Phi(P,\boldsymbol{y})$.

We can rewrite the induction principle (2) in the following way

$$\forall \boldsymbol{x}.\{Z(\boldsymbol{x}) \supset \forall P.[\forall \boldsymbol{y}.[\Phi(P,\boldsymbol{y}) \supset P(\boldsymbol{y})] \supset P(\boldsymbol{x})]\}.$$

Notice that implication in the opposite direction follows from the construction principle (1). We obtain

$$\forall \boldsymbol{x}.\{\mu Z(\boldsymbol{x}).\Phi(Z,\boldsymbol{x}) \equiv \forall P.[\forall \boldsymbol{y}.[\Phi(P,\boldsymbol{y}) \supset P(\boldsymbol{y})] \supset P(\boldsymbol{x})\}.$$

The last sentence is often considered as a formal definition of a least fixed point.

To guarantee that $Z(\boldsymbol{x})$ is the greatest solution of (1), we apply the coinduction principle:

$$\forall P,\boldsymbol{x}.\{\forall \boldsymbol{y}.[P(\boldsymbol{y}) \supset \Phi(P,\boldsymbol{y})] \supset [P(\boldsymbol{x}) \supset Z(\boldsymbol{x})]\}. \tag{3}$$

i.e., whatever solution $P(\boldsymbol{y})$ of the recursive specification we take, $Z(\boldsymbol{x})$ is a greater or equal one. The first implication can be replaced with equivalence.

We can rewrite the coinduction principle (3) in the following way

$$\forall \boldsymbol{x}.\{\exists P.[\forall \boldsymbol{y}.[P(\boldsymbol{y}) \supset \Phi(P,\boldsymbol{y})] \land P(\boldsymbol{x})] \supset Z(\boldsymbol{x})\}.$$

An explicit expression for a greatest fixed point can be obtained in a similar way as for a least fixed point:

$$\forall \boldsymbol{x}.\{\nu Z(\boldsymbol{x}).\Phi(Z,\boldsymbol{x}) \equiv \exists P.[\forall \boldsymbol{y}.[P(\boldsymbol{y}) \supset \Phi(P,\boldsymbol{y})] \land P(\boldsymbol{x})]\}.$$

The last sentence can be taken as a definition of a greatest fixed point. Notice that μ and ν can be considered as quantifiers, and

$$\begin{aligned} \mu Z(\boldsymbol{x}).\Phi(Z,\boldsymbol{x}) &\equiv \neg\nu Z(\boldsymbol{x}).\neg\Phi(\neg Z,\boldsymbol{x}),\\ \nu Z(\boldsymbol{x}).\Phi(Z,\boldsymbol{x}) &\equiv \neg\mu Z(\boldsymbol{x}).\neg\Phi(\neg Z,\boldsymbol{x}). \end{aligned}$$

In the following, we always start with a set-theoretic rule-based inductive definition and then demonstrate its logical representation. This method is the essence of our approach. In Sec. 4, the rules of an inductive definition will be considered as causal laws representing direct and indirect effects of actions. The translation between the set of inductive rules and its logical representation constitutes our solution to the frame and ramification problems. Then, in Sec. 6, inductive rules will represent the rules of structural operational semantics [Plotkin, 1981] for process algebra (e.g. [Milner, 1989]). The logical representation of these rules is the inductive definition of relation $Trans$ specifying transitions performed by GOLOG programs [De Giacomo *et al.*, 1997b].

3 Situation Calculus Structures

The language $\mathcal{L}^{sc}$ of the situation calculus is a many-sorted language with equality with sorts for actions ($\mathcal{A}$), objects ($\mathcal{O}$), situations ($\mathcal{S}$), and processes ($\mathcal{P}$). For simplicity, we will assume that there is only one sort of objects. Actions represent all changes in the world and may have objects as their arguments. A situation denotes a sequence of performed actions and represents a world history. We use the following non-logical symbols. The constant S_0 is used to denote the initial situation, the empty history. It is analogous to the number 0 in Peano arithmetic. Binary function symbol $do(a, s)$ denotes the successor situation obtained from situation s by performing action a. Other relations such as *fluents*, i.e., properties of the world, are defined inductively (see, e.g. [Moschovakis, 1974]).

In order to interpret $\mathcal{L}^{sc}$-terms and $\mathcal{L}^{sc}$-formulas, it is necessary to fix an $\mathcal{L}^{sc}$-structure

$$\mathfrak{U} \;=\; (\,\mathcal{U},\;\; do^{\mathfrak{U}},\; S_0^{\mathfrak{U}},\; a_1^{\mathfrak{U}},\ldots,a_n^{\mathfrak{U}},\; c_1^{\mathfrak{U}},\ldots,c_m^{\mathfrak{U}}\;, F_1^{\mathfrak{U}},\ldots,F_n^{\mathfrak{U}},\ldots)$$

consisting of the following:

- the universe $\mathcal{U}$ such that $\mathcal{U} = \mathcal{S}\ \cup\ \mathcal{A}\ \cup\ \mathcal{O}\ \cup\ \mathcal{P}$, and $\mathcal{S}$, $\mathcal{A}$, $\mathcal{O}$, and $\mathcal{P}$ are nonempty[4].
- a binary function $do^{\mathfrak{U}} : \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{S}$.
- a 0-ary function (individual constant) $S_0^{\mathfrak{U}}$ of sort $\mathcal{S}$.
- functions $a_1^{\mathfrak{U}}, \ldots, a_n^{\mathfrak{U}}$, $n > 0$, taking values in $\mathcal{A}$. With each $a_i^{\mathfrak{U}}$, we associate its arity j. Thus, $a_i^{\mathfrak{U}} : \mathcal{O}^j \rightarrow \mathcal{A}$.
- each relation $F_i^{\mathfrak{U}}$ (a fluent) denotes the set of tuples of the form $\langle c_i^{\mathfrak{U}}, S_0^{\mathfrak{U}}\rangle$ which contain no situations other than S_0. With each $F_i^{\mathfrak{U}}$, we associate its arity j. Thus, $F_i^{\mathfrak{U}} \subseteq \mathcal{O}^{j-1} \times S_0^{\mathfrak{U}}$. Fluents are definable on situations other than S_0.
- a set of 0-ary functions (individual constants) $c_1^{\mathfrak{U}}, \ldots, c_m^{\mathfrak{U}}$, $m \geq 0$, ranging over universe of objects $\mathcal{O}$.
- functions taking values in $\mathcal{P}$ represented by the dots at the end. We will describe these functions in Sec. 6.

From now on, we assume that the structure $\mathfrak{U}$ is fixed. Notice that an $\mathcal{L}^{sc}$-interpretation $\mathfrak{U}$ only suffices to assign truth values to some $\mathcal{L}^{sc}$-formulas, namely those that are $\mathcal{L}^{sc}$-sentences. In general, the truth value of a formula depends not only on the interpretation, but also on values assigned to some $\mathcal{L}^{sc}$-variables. We therefore define an $\mathcal{L}^{sc}$-*valuation* σ to be an $\mathcal{L}^{sc}$-structure $\mathfrak{U}$ together with an *assignment* of a value $x^\sigma \in \mathcal{U}$ to each variable x. The *value* t^σ of a term t under valuation σ is defined recursively. The equality relation under σ, $=^\sigma$, is the identity relation, and the *value* ϕ^σ of formula ϕ under σ is defined recursively as usual. Note that the value of a term and the truth value of a formula are not parts of a valuation (cf. [Bell and Machover, 1977]). Let $\phi(\boldsymbol{x})$ be a formula such that all the free variables of ϕ are among $\boldsymbol{x}$. We write $\models_{\mathfrak{U}} \phi(\boldsymbol{x})$ to assert that formula $\phi(\boldsymbol{x})$ is true in structure $\mathfrak{U}$ under any valuation.

[4] If all actions are 0-ary functions and fluents do not contain non-situation arguments, then the universe of objects $\mathcal{O}$ is allowed to be empty, in which case the list of individual constants $c_1^{\mathfrak{U}}, \ldots, c_m^{\mathfrak{U}}$ is empty as well, i.e., $m = 0$.

Let us consider a natural axiomatization of the properties of situations. Unlike natural numbers, situations compose a tree[5]. The set of situations is constructed starting from the constant S_0 by iterative applications of function $do(a, s)$. In place of Peano axioms $0 \neq succ(x)$, $succ(x) = succ(y) \supset x = y$, we have the following unique name assumptions for situations:

$$S_0 \neq do(a, s), \qquad (\mathrm{UNA}_{\mathrm{sit}})$$
$$do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2.$$

These axioms specify desired properties of the set of situations Sit, however they do not describe how this set is constructed. We start from constructing this set semantically with respect to the structure $\mathfrak{U}$ of the situation calculus, and then consider its representation in the language of this structure.

Before proceeding, a notational convention has to be made. In what follows, we will omit the superscript referring to structure $\mathfrak{U}$. Thus, in the inductive rules, $s \in Z$ stands for $s^{\mathfrak{U}} \in Z^{\mathfrak{U}}$, $do(a, s)$ represents $do(a, s)^{\mathfrak{U}}$, and so on.

The set of situations, Sit, is inductively defined as the smallest set Z by the set of rules

$$\Re_{\mathrm{Sit}} = \left\{ \frac{\emptyset}{S_0} \right\} \cup \left\{ \frac{s}{do(a, s)} \middle| \, s \in Z, \; a \in \mathcal{A} \right\}. \qquad (4)$$

The operator induced by this inductive definition is

$$\Gamma_{\mathrm{Sit}}(Z) = \{S_0\} \cup \{do(a, s) | s \in Z, \; a \in \mathcal{A}\}.$$

The formula which defines the operator $\Gamma_{\mathrm{Sit}}(Z)$ is

$$\Phi_{\mathrm{Sit}}(Z, s) \stackrel{\mathrm{def}}{\equiv} s = S_0 \vee (\exists s', a).[s = do(a, s') \wedge Z(s')].$$

We define $Sit(s)$ as a least fixed point of Φ_{Sit}: $Sit(s) \stackrel{\mathrm{def}}{\equiv} \mu Z(s).\Phi_{\mathrm{Sit}}(Z, s)$. The construction principle for the set of situations is $\Phi_{\mathrm{Sit}}(Sit, s) \equiv Sit(s)$. The induction principle is

$$\forall s.\{Sit(s) \supset \forall P.[\forall \tilde{s}.[\Phi_{\mathrm{Sit}}(P, \tilde{s}) \supset P(\tilde{s})] \supset P(s)]\}.$$

Substituting the expression for $\Phi_{\mathrm{Sit}}(Z, s)$ and simplifying:

$$\forall s.\{Sit(s) \supset$$
$$\forall P.[P(S_0) \wedge (\forall s', a).P(s') \supset P(do(a, s'))] \supset P(s)]\}.$$

Since we require all situations in the domain to be in the set Sit, we specify $\forall s.\ Sit(s)$. The induction principle above becomes the standard induction principle on situations [Reiter, 1993]:

$$\forall P.[P(S_0) \wedge (\forall s', a).P(s') \supset P(do(a, s')) \supset \forall s.P(s)]. \qquad (\mathrm{IND}_{\mathrm{sit}})$$

[5] In general, we do not impose any restriction on the cardinality of the domain of actions $\mathcal{A}$. It can be finite, isomorphic to the natural numbers, or even uncountable. An uncountable set of actions occurs, for example, if actions have arguments ranging over real numbers specifying the time of execution. If $\mathcal{A}$ is finite, situations compose a finitely branching tree.

Let $\mathfrak{U}$ be a structure for $\mathcal{L}^{sc}$. We call $\mathfrak{U}$ *tree-like* if $(\mathcal{S}, do^{\mathfrak{U}}, S_0^{\mathfrak{U}})$ is a rooted acyclic structure of $|\mathcal{A}|$ successors.

Theorem 1.

$$\models_{\mathfrak{U}} \ \mathrm{IND_{sit}} \wedge \mathrm{UNA_{sit}} \iff \mathfrak{U} \textit{ is tree-like},$$

i.e., the class of tree-like structures is completely characterized by the induction principle on situations and unique names assumptions for situations.

Proof. From the construction of the set *Sit* it follows that every tree-like structure satisfies ($\mathrm{IND_{sit}}$) and ($\mathrm{UNA_{sit}}$). The other direction is slightly more difficult. Suppose Υ is a structure for the language of the situation calculus that satisfies the induction principle on situations ($\mathrm{IND_{sit}}$) and the unique name assumptions for situations ($\mathrm{UNA_{sit}}$), and that *is not* a tree-like structure. Then either Υ contains a loop including S_0, but this violates the first axiom of ($\mathrm{UNA_{sit}}$), or Υ contains merging, but this violates the second axiom of ($\mathrm{UNA_{sit}}$), or it contains situations not reachable from S_0 by one or more applications of *do*, but this contradicts the induction principle on situations ($\mathrm{IND_{sit}}$). Thus every structure that satisfies the induction principle on situations ($\mathrm{IND_{sit}}$) and the unique name assumptions for situations ($\mathrm{UNA_{sit}}$) is a tree-like structure. □

An interesting observation is that a particular axiomatization of the properties of situations does not affect further results. We may, for example, allow merging or cyclic time. However, in some applications, such as planning, unique name assumptions for situations are necessary. These assumptions allow us to interpret situations as histories or records of performed actions. We could prove, for example, that two situations are equal if and only if they are obtained by the same sequence of actions.

4 Frame Problem

The *frame problem* is the question of how to specify invariants of the domain, i.e., those fluents unaffected by the performance of an action. In this section we provide a solution to the frame problem from the point of view of inductive definability, thereby obtaining successor state axioms of the form

$$\begin{aligned}(\forall \boldsymbol{x}, a, s).\ & F(\boldsymbol{x}, do(a,s)) \equiv \\ & [\gamma_F^+(\boldsymbol{x}, a, s) \vee F(\boldsymbol{x}, s) \wedge \neg\gamma_F^-(\boldsymbol{x}, a, s)].\end{aligned} \tag{5}$$

Formula $\gamma_F^+(\boldsymbol{x}, a, s)$ (respectively, $\gamma_F^-(\boldsymbol{x}, a, s)$) denotes a first order formula specifying the conditions under which fluent F is true (respectively, false) in the successor situation [Reiter, 1991]. The only free variables of these formulas are those among $\boldsymbol{x}$, a, s. An example of a successor axiom is

$$\begin{aligned}(\forall sw, a, s).\ & [on(sw, do(a,s)) \equiv \\ & a = toggle(sw) \wedge \neg on(sw, s) \vee on(sw, s) \wedge a \neq toggle(sw)].\end{aligned}$$

This axiom says that a switch is on in situation $do(a,s)$ if and only if this situation was obtained by performing action $toggle(sw)$ in situation s where this switch was off, or the switch was already on and an action other than $toggle(sw)$ was performed.

We have seen the connection between logic programming and inductive definitions. The solution to the frame problem using inductive definitions provides a formal justification for encoding situation calculus theories as logic programs.

In the framework of the situation calculus, each fluent $F(x_1, ..., x_n, s)$, $n \geq 0$, (or, equivalently, $F(\boldsymbol{x}, s)$) represents an $(n+1)$-ary relation on the elements of the domain, i.e., a set of $(n+1)$-ary tuples. From the point of view of causality, a fluent can be caused to hold, or caused not to hold in a situation. Following this view, for each fluent, we will construct two sets of tuples, $\mathcal{F}^+$ and $\mathcal{F}^-$, according to a set of inductive rules. In the language of the situation calculus, the set $\mathcal{F}^+$ (respectively, $\mathcal{F}^-$) is represented by fluent F (respectively, by $\neg F$). Formula γ_F^+ (respectively, γ_F^-) denotes the condition under which a contribution to the set $\mathcal{F}^+$ (respectively, $\mathcal{F}^-$) has to be made. Each set $\mathcal{F}^+$ is inductively defined as a set of tuples containing situations other than S_0.

$$\Re_{\mathcal{F}^+} = \left\{ \frac{\emptyset}{\langle \boldsymbol{x}, do(a,s)\rangle} \middle| \models_{\mathfrak{U}'} \gamma_F^+(\boldsymbol{x}, a, s) \right\}$$
$$\cup \left\{ \frac{\langle \boldsymbol{x}, s\rangle}{\langle \boldsymbol{x}, do(a,s)\rangle} \middle| \langle \boldsymbol{x}, s\rangle \in Z_{\mathcal{F}^+}, \models_{\mathfrak{U}'} \neg\gamma_F^-(\boldsymbol{x}, a, s) \right\}.$$

Similarly, for each set $\mathcal{F}^-$,

$$\Re_{\mathcal{F}^-} = \left\{ \frac{\emptyset}{\langle \boldsymbol{x}, do(a,s)\rangle} \middle| \models_{\mathfrak{U}'} \gamma_F^-(\boldsymbol{x}, a, s) \right\}$$
$$\cup \left\{ \frac{\langle \boldsymbol{x}, s\rangle}{\langle \boldsymbol{x}, do(a,s)\rangle} \middle| \langle \boldsymbol{x}, s\rangle \in Z_{\mathcal{F}^-}, \models_{\mathfrak{U}'} \neg\gamma_F^+(\boldsymbol{x}, a, s) \right\}.$$

The initial structure $\mathfrak{U}$ is modified every time an inductive rule is applied. We use prime in $\models_{\mathfrak{U}'}$ to refer to the current structure. In the inductive definition above, the rules defining $\mathcal{F}^+$ and $\mathcal{F}^-$ may include occurrences of other fluents, also defined inductively. This type of definition is called simultaneous induction [Moschovakis, 1974]. Notice that the value of each fluent in situation s is determined prior to determining its value in situation $do(a,s)$, and that negation may occur both in the premises and conclusions of the rules. The inductive definition above assumes it is possible that there is no information about whether a particular fluent (or a negated fluent, respectively) holds or not in the initial situation. This feature provides a way to deal with incomplete information. In other words, the rule-based inductive definition is parameterized with respect to the base case, the initial situation.

Whenever a contribution to set $\mathcal{F}^+$ (respectively, to set $\mathcal{F}^-$) is made, we say that fluent F is caused to be true (respectively, false) in the successor situation. As always for an inductive definition, we need a closure condition: *The rules above characterize all the conditions under which action a can cause F to become true* *(respectively, false) in the successor situation.* This closure condition is in fact the same as the informal Causal Completeness Assumption [Reiter, 1991] which is necessary to solve the frame problem. We formulate it as a second order induction principle. The formula corresponding to the rule-based inductive definition is a conjunction of the following formulas for each fluent F.

$$\begin{aligned}\Phi_{\mathcal{F}^*}(Z_{\mathcal{F}^*}, \boldsymbol{x}, s) \stackrel{\text{def}}{\equiv}\ & (\exists s', a).s = do(a, s') \wedge \gamma_F^+(\boldsymbol{x}, a, s') \\ \vee(\exists s', a).[s = do(a, s') \wedge & Z_{\mathcal{F}^*}(\boldsymbol{x}, s') \wedge \neg\gamma_F^-(\boldsymbol{x}, a, s')].\end{aligned}$$

The star $*$ stands either for $+$ or for $-$. Thus, for each fluent F we define two constructors, one for $\mathcal{F}^+$, and one for $\mathcal{F}^-$. The inductive rules above can be rewritten as causal rules in the following way.

$$\begin{aligned}F(\boldsymbol{x}, do(a, s)) &\leftarrow \gamma_F^+(\boldsymbol{x}, a, s) \\ F(\boldsymbol{x}, do(a, s)) &\leftarrow F(\boldsymbol{x}, s) \wedge \neg\gamma_F^-(\boldsymbol{x}, a, s)\end{aligned}$$

$$\begin{aligned}\neg F(\boldsymbol{x}, do(a, s)) &\leftarrow \gamma_F^-(\boldsymbol{x}, a, s) \\ \neg F(\boldsymbol{x}, do(a, s)) &\leftarrow \neg F(\boldsymbol{x}, s) \wedge \neg\gamma_F^+(\boldsymbol{x}, a, s)\end{aligned}$$

The arrow represents a causal dependence and is stronger than implication.

Our main result says that the simultaneous inductive definition of all fluents is expressible by a collection of first order successor state axioms, provided a consistency condition holds. Every successor state axiom represents the inductive definition for both $\mathcal{F}^+$ and $\mathcal{F}^-$.

Theorem 2. *Under consistency condition* $\neg(\exists \boldsymbol{x}, a, s).\ \gamma_F^+(\boldsymbol{x}, a, s) \wedge \gamma_F^-(\boldsymbol{x}, a, s)$, *the simultaneous inductive definition of all fluents is logically equivalent to the conjunction of successor state axioms for each fluent* F *of the form*

$$\begin{aligned}&(\forall \boldsymbol{x}, s, a).\ F(\boldsymbol{x}, do(a, s)) \equiv \\ &\qquad [\gamma_F^+(\boldsymbol{x}, a, s) \vee F(\boldsymbol{x}, s) \wedge \neg\gamma_F^-(\boldsymbol{x}, a, s)].\end{aligned}$$

Proof (outline) The idea is to consider the definition of each set separately and then to combine the definitions of $\mathcal{F}^+$ and $\mathcal{F}^-$, recalling that $\mathcal{F}^+$ is represented in the language by F, and $\mathcal{F}^-$ is represented by $\neg F$. The result is exactly the successor state axiom for F. Considering the definition of each set separately is possible because the simultaneous induction principle is equivalent to the conjunction of induction principles for each fluent. □

Example 1. Suppose we have the following causal laws (inductive rules) specifying direct effects of actions.

$$\begin{aligned}on(sw, do(a, s)) &\leftarrow \neg on(sw, s) \wedge a = toggle(sw) \\ on(sw, do(a, s)) &\leftarrow on(sw, s) \wedge \neg[on(sw, s) \wedge a = toggle(sw)] \\ \neg on(sw, do(a, s)) &\leftarrow on(sw, s) \wedge a = toggle(sw) \\ \neg on(sw, do(a, s)) &\leftarrow \neg on(sw, s) \wedge \neg[\neg on(sw, s) \wedge a = toggle(sw)]\end{aligned}$$

The corresponding successor state axiom is

$$on(sw, do(a, s)) \equiv$$
$$\underbrace{\neg on(sw, s) \wedge a = toggle(sw)}_{\gamma^+_{on}} \vee on(sw, s) \wedge \neg \underbrace{[on(sw, s) \wedge a = toggle(sw)]}_{\gamma^-_{on}}.$$

After simplification we obtain

$$\begin{gathered} on(sw, do(a, s)) \equiv \\ a = toggle(sw) \wedge \neg on(sw, s) \vee on(sw, s) \wedge a \neq toggle(sw). \end{gathered} \tag{6}$$

5 Ramification Problem

The *Ramification problem* is the question of how to handle indirect effects of actions. These effects are usually associated with domain constraints, sentences that rule out illegal states of the world. However it has been observed that not all of the consequences derived from domain constraints correspond to our intuition. Consider a simple example from [Thielscher, 1996][6]. Suppose a light is on if and only if two switches are on. This can be expressed by the domain constraint $(\forall s).\ on(sw_1, s) \wedge on(sw_2, s) \equiv light(s)$, where sw_1 and sw_2 are constants. Suppose that in a particular situation S_0, $on(sw_1, S_0)$ and $light(S_0)$ are false, and $on(sw_2, S_0)$ is true. Toggling the first switch, aside from the direct effect of $on(sw_1, do(toggle(sw_1), S_0))$ becoming true, leads to the light bulb turning on. This indirect effect is derived using implication $(\forall s).\ on(sw_1, s) \wedge on(sw_2, s) \supset light(s)$. This formula is logically entailed by the domain constraint. Notice that the domain constraint also entails the implication $(\forall s).\ on(sw_1, s) \wedge \neg light(s) \supset \neg on(sw_2, s)$. According to this sentence, the result of toggling the first switch is that the second switch magically changes its position. This conclusion contradicts to our intuition. The reason for the inadequacy of the naive approach is that domain constraints usually do not contain causal information.

In this paper, we propose to encode the causal laws describing indirect effects of action as rules of an inductive definition, exactly as we have done for causal laws describing direct effects of actions. Our goal is to obtain successor state axioms in the correct form, i.e., without occurrences of the function *do* on the right hand side of the equivalence. A necessary condition to achieve this goal is that the dependency graph for causal rules specifying indirect effects of actions does not contain cycles. This condition can be relaxed, but in this case successor state axioms will not be obtained.

McIlraith [McIlraith, 1997] describes a solution to the ramification problem referring to the notion of stratified logic program. She justifies the informal causal completion assumption, used in compiling effect axioms and ramification constraints into successor state axioms, by a prioritized circumscription policy. We use the idea behind the compilation procedure, however, instead of prioritized circumscription we appeal to the induction principle. The power of our inductive

[6] Essentially the same example was considered in [Lin, 1995] and other works.

view is that the solution to the ramification problem becomes an elementary consequence of the previous theorem.

We construct sets $\mathcal{F}^+$ and $\mathcal{F}^-$ as above, but the set of rules now includes causal laws specifying indirect effects of actions in addition to causal laws describing direct effects. The inductive rules for $\mathcal{F}^+$ are as follows.

$$\begin{aligned} \Re_{\mathcal{F}^+} = & \left\{ \frac{\emptyset}{\langle \boldsymbol{x}, do(a,s)\rangle} \,\middle|\, \models_{\mathfrak{U}} \ \gamma_F^+(\boldsymbol{x},a,s) \right\} \\ \cup & \left\{ \frac{\emptyset}{\langle \boldsymbol{x}, do(a,s)\rangle} \,\middle|\, \models_{\mathfrak{U}'} \ \mathcal{R}[v_F^+(\boldsymbol{x}, do(a,s))] \right\} \\ \cup & \left\{ \frac{\langle \boldsymbol{x}, s\rangle}{\langle \boldsymbol{x}, do(a,s)\rangle} \,\middle|\, \langle \boldsymbol{x}, s\rangle \in Z_{\mathcal{F}^+}, \ \models_{\mathfrak{U}'} \ \neg[\gamma_F^-(\boldsymbol{x},a,s) \vee \mathcal{R}[v_F^-(\boldsymbol{x}, do(a,s))]] \right\}. \end{aligned}$$

The first two lines represent causal laws, for direct and indirect effects of actions, respectively; the third line represents the law of inertia. Similarly, for each set $\mathcal{F}^-$,

$$\begin{aligned} \Re_{\mathcal{F}^-} = & \left\{ \frac{\emptyset}{\langle \boldsymbol{x}, do(a,s)\rangle} \,\middle|\, \models_{\mathfrak{U}'} \ \gamma_F^-(\boldsymbol{x},a,s) \right\} \\ \cup & \left\{ \frac{\emptyset}{\langle \boldsymbol{x}, do(a,s)\rangle} \,\middle|\, \models_{\mathfrak{U}'} \ \mathcal{R}[v_F^-(\boldsymbol{x}, do(a,s))] \right\} \\ \cup & \left\{ \frac{\langle \boldsymbol{x}, s\rangle}{\langle \boldsymbol{x}, do(a,s)\rangle} \,\middle|\, \langle \boldsymbol{x}, s\rangle \in Z_{\mathcal{F}^-}, \ \models_{\mathfrak{U}'} \ \neg[\gamma_F^+(\boldsymbol{x},a,s) \vee \mathcal{R}[v_F^+(\boldsymbol{x}, do(a,s))]] \right\}, \end{aligned}$$

In these rules, $\mathcal{R}$ is the regression operator. This operator uses successor state axioms to generate a formula equivalent to its argument but without the term $do(a,s)$. Formula $v_F^+(\boldsymbol{x},s)$ (respectively, $v_F^-(\boldsymbol{x},s)$) denotes the conditions under which fluent F is caused to hold (respectively, not to hold) according to causal rules specifying static dependencies.

The difference between this set of rules and the one from the previous section is that we have included causal rules representing indirect effects of actions. We could simply replace every occurrence of $\gamma_F^*(\boldsymbol{x},a,s)$ by $[\gamma_F^*(\boldsymbol{x},a,s) \vee \mathcal{R}[v_F^*(\boldsymbol{x}, do(a,s))]]$, where $*$ stands for either $+$ or $-$.

The inductive rules above can be rewritten as causal rules as follows.

$$\begin{aligned} F(\boldsymbol{x}, do(a,s)) &\leftarrow \gamma_F^+(\boldsymbol{x},a,s) \\ F(\boldsymbol{x}, do(a,s)) &\leftarrow F(\boldsymbol{x},s) \wedge \neg\gamma_F^-(\boldsymbol{x},a,s) \end{aligned}$$

$$\begin{aligned} \neg F(\boldsymbol{x}, do(a,s)) &\leftarrow \gamma_F^-(\boldsymbol{x},a,s) \\ \neg F(\boldsymbol{x}, do(a,s)) &\leftarrow \neg F(\boldsymbol{x},s) \wedge \neg\gamma_F^+(\boldsymbol{x},a,s) \end{aligned}$$

$$\begin{aligned} F(\boldsymbol{x}, do(a,s)) &\leftarrow \mathcal{R}[v_F^+(\boldsymbol{x},s)] \\ \neg F(\boldsymbol{x}, do(a,s)) &\leftarrow \mathcal{R}[v_F^-(\boldsymbol{x},s)] \end{aligned}$$

As before, we are interested in the sufficient conditions for a first order representation of successor state axioms. A result, similar to one for the case without domain constraints, holds for the case when such constraints are present. The consistency statement $\neg(\exists \boldsymbol{x}, a, s).\ \gamma_F^+(\boldsymbol{x}, a, s) \wedge \gamma_F^-(\boldsymbol{x}, a, s)$ should be reformulated taking into account both γ- and v-formulas: the conditions causing fluent F to become true and those making it false are never true simultaneously.

$$\begin{array}{c}\neg(\exists \boldsymbol{x}, a, s).\ [\gamma_F^+(\boldsymbol{x}, a, s) \wedge \gamma_F^-(\boldsymbol{x}, a, s)\\ \vee \mathcal{R}[v_F^+(\boldsymbol{x}, do(a, s))] \wedge \mathcal{R}[v_F^-(\boldsymbol{x}, do(a, s))]\\ \vee \gamma_F^+(\boldsymbol{x}, a, s) \wedge \mathcal{R}[v_F^-(\boldsymbol{x}, do(a, s))]\\ \vee \mathcal{R}[v_F^+(\boldsymbol{x}, do(a, s))] \wedge \gamma_F^-(\boldsymbol{x}, a, s)].\end{array} \tag{7}$$

Theorem 3. *Under consistency condition (7), the simultaneous inductive definition of all fluents on situations other than S_0 is logically equivalent to the conjunction of successor state axioms for each fluent F of the form*

$$\begin{array}{c}(\forall \boldsymbol{x}, s, a).\ F(\boldsymbol{x}, do(a, s)) \equiv\\ [\gamma_F^+(\boldsymbol{x}, a, s) \vee \mathcal{R}[v_F^+(\boldsymbol{x}, do(a, s)]\\ \vee F(\boldsymbol{x}, s) \wedge \neg[\gamma_F^-(\boldsymbol{x}, a, s) \vee \mathcal{R}[v_F^-(\boldsymbol{x}, do(a, s))]]].\end{array}$$

Proof. This is a simple corollary of theorem 2. □

Example 2. Suppose we have the following causal laws specifying direct and indirect effects of actions.

$$\begin{array}{l}light(do(a, s)) \leftarrow false\\ light(do(a, s)) \leftarrow light(s)\\ \qquad\qquad\qquad\quad \wedge\neg[light(s) \wedge [a = toggle(sw_1) \vee a = toggle(sw_2)]]\end{array}$$

$$\begin{array}{l}\neg light(do(a, s)) \leftarrow light(s) \wedge [a = toggle(sw_1) \vee a = toggle(sw_2)]\\ \neg light(do(a, s)) \leftarrow \neg light(s) \wedge \neg false\end{array}$$

$$\begin{array}{r}light(do(a, s)) \leftarrow on(sw_1, s) \wedge on(sw_2, s)\\ \neg light(do(a, s)) \leftarrow false\end{array}$$

In this case $\gamma_F^+(\boldsymbol{x}, a, s) \equiv v_F^-(\boldsymbol{x}, s) \equiv false$. The corresponding intermediate successor state axiom is

$$\begin{array}{c}light(do(a, s)) \equiv\\ \mathcal{R}\,\underbrace{[on(sw_1, do(a, s)) \wedge on(sw_2, do(a, s))]}_{v^+_{light}}\\ \vee light(s) \wedge \neg\,\underbrace{[light(s) \wedge [a = toggle(sw_1) \vee a = toggle(sw_2)]]}_{\gamma^-_{light}}.\end{array}$$

We substitute $on(sw_1, do(a, s))$ and $on(sw_2, do(a, s))$ with equivalent formulas according to successor state axiom for on (6) from example 1 and simplify. We obtain

$$\begin{array}{c}light(do(a, s)) \equiv\\ a = trigger(sw_1) \wedge \neg on(sw_1, s) \wedge on(sw_2, s)\\ a = trigger(sw_2) \wedge on(sw_2, s) \wedge on(sw_1, s)\\ \vee light(s) \wedge a \neq toggle(sw_1) \wedge a \neq toggle(sw_2).\end{array}$$

6 GOLOG

GOLOG [Levesque *et al.*, 1996] is a situation calculus-based logic programming language that allows for defining complex actions and recursive procedures. The interpreter of the language is a logic programming implementation of the inductive definition of the transition relation $Trans$. We assume the existence of a unary function tilde ($\tilde{}$) mapping primitive actions to processes. For example, $\widetilde{a}$ denotes a process consisting of one primitive action a. Well-formed terms of sort process are composed using the following non-logical functions[7]:

nil	empty process
$\widetilde{a}$	primitive action-process
$\phi?$	wait for a condition (test)
$(\delta_1;\delta_2)$	sequence
$(\delta_1 \vert \delta_2)$	nondeterministic choice between actions
$\pi x.\delta$	nondeterministic choice of arguments

In this section we consider the same structure $\mathfrak{U}$ for the language of the situation calculus.

$$\mathfrak{U} = (\ ..., nil^{\mathfrak{U}}, \ \sim^{\mathfrak{U}}, \ ?^{\mathfrak{U}}, \ ;^{\mathfrak{U}}, \ |^{\mathfrak{U}}, \ \pi^{\mathfrak{U}} \ )$$

The dots now stand for the part of the structure described in Sec. 3. We write $\delta \doteq \sigma$ to indicate that process δ behaves as process σ. This metamathematical notation is useful for defining recursive procedures. An example of a recursive program is a simple clock which performs a $tick$ action and then behaves as clock again. $Clock \ \doteq \ tick \ ; \ Clock$. We introduce notations:

$$\texttt{if } \phi? \texttt{ then } \delta \texttt{ else } \delta' \ \doteq \ \phi?;\delta \ \mid \ \neg\phi?;\delta'$$

$$\texttt{while } \phi? \texttt{ do } \delta \ \doteq \ \texttt{if } \phi? \texttt{ then } \delta; (\texttt{while } \phi? \texttt{ do } \delta) \texttt{ else } nil$$

In fact, the "while"-notation is not strictly necessary because the last recursive specification above could be rewritten as $X \ \doteq \ \texttt{if } \phi? \texttt{ then } \delta; X \texttt{ else } nil$. However, for convenience we prefer to keep this notation as a special case of a recursive specification.

Definition 1. *A* configuration $\langle\delta, s\rangle$ *is a pair, where the first element* (δ) *is a well-formed term of sort process, and the second element* (s) *is a situation.*

A *transition* is a quadruple $\langle\delta, s, \quad \delta', s'\rangle$, where the first two elements, $\langle\delta, s\rangle$, and the last two elements, $\langle\delta', s'\rangle$, are configurations. The intuitive sense of a transition is that a program in situation s and with state of execution δ may move to the situation s' where it has the state of execution δ'. The transition relation $Trans$, representing the set of possible transitions, is defined inductively

[7] The concurrency operator of process algebra can also be introduced, following the lines of [De Giacomo *et al.*, 1997a]. Since we do not use concurrency in our example, we omit this operator.

below[8]. The base case of induction is the transitions caused by a primitive action or a test action. A set of inductive rules allows building the transition relation on larger instances of processes. The induction principle for $Trans$ says that no transitions other than those specified by the induction rules exist.

Definition 2. *A* transition system definable by a program *is a pair*

$$\mathcal{T} = (\ C,\ Trans\ )$$

where C is a non-empty set of configurations, $Trans$ is a non-empty set of transitions.

Notice that this transition system corresponds to what is usually understood as a transition system unwound into a tree. Recall that for simplicity we have assumed that the arguments of actions all are of one sort, and they range over sub-domain $\mathcal{O}$. $Trans$ is inductively defined as the smallest set Z by the set of rules

$$\Re_{\text{trans}} = \left\{ \frac{\emptyset}{\langle \tilde{a}, s,\ \ nil, do(a,s)\rangle} \right\} \cup \left\{ \frac{\emptyset}{\langle \phi?, s,\ \ nil, s\rangle} \middle| \models_{\mathfrak{U}'}\ \phi(s) \right\}$$

$$\cup \left\{ \frac{\langle \delta_2, s,\ \ \delta_2', s'\rangle}{\langle [\delta_1;\delta_2], s,\ \ \delta_2', s'\rangle} \middle| \models_{\mathfrak{U}'}\ Final(\delta_1, s),\ \langle \delta_2, s,\ \ \delta_2', s'\rangle \in Z \right\}$$

$$\cup \left\{ \frac{\langle \delta_2, s,\ \ \delta_1', s'\rangle}{\langle [\delta_1;\delta_2], s,\ \ [\delta_1';\delta_2], s'\rangle} \middle| \langle \delta_2, s,\ \ \delta_1', s'\rangle \in Z \right\}$$

$$\cup \left\{ \frac{\langle \delta_1, s,\ \delta', s'\rangle}{\langle [\delta_1|\delta_2], s,\ \delta', s'\rangle} \middle| \langle \delta_1, s,\ \delta', s'\rangle \in Z \right\} \cup \left\{ \frac{\langle \delta_2, s,\ \delta', s'\rangle}{\langle [\delta_1|\delta_2], s,\ \delta', s'\rangle} \middle| \langle \delta_2, s,\ \delta', s'\rangle \in Z \right\}$$

$$\cup \left\{ \frac{\langle \delta|_c^x, s,\ \delta', s'\rangle}{\langle \pi x.\delta, s,\ \delta', s'\rangle} \middle| \langle \delta|_c^x, s,\ \delta', s'\rangle \in Z \text{ for some } c \in \mathcal{O} \right\}$$

$$\cup \left\{ \frac{\langle \sigma(\boldsymbol{x})|_{\boldsymbol{p}}^{\boldsymbol{x}}, s,\ \delta', s'\rangle}{\langle \delta(\boldsymbol{p}), s,\ \delta', s'\rangle} \middle|\ \langle \sigma(\boldsymbol{x})|_{\boldsymbol{p}}^{\boldsymbol{x}}, s,\ \delta', s'\rangle \in Z,\ \delta(\boldsymbol{x}) \doteq \sigma(\boldsymbol{x}) \right\}.$$

In the last rule, $\boldsymbol{p}$ are actual parameters of the procedure[9]. $Final(\delta, s)$ above means that configuration $\langle \delta, s\rangle$ may eventually lead to the termination of δ. For example, nil is final, if δ_1 and δ_2 are final, then so is $\delta_1;\delta_2$, if either δ_1 or δ_2 is final, then so is $\delta_1|\delta_2$. The formal definition of this predicate is similar to that of $Trans$ and we omit it here.

We define $Trans(\delta, s, \delta', s')$ as a least fixed point of $\Phi_{\text{trans}}(Z, \delta, s, \delta', s')$, a formula defining the corresponding operator:

$$Trans(\delta, s, \delta', s') \stackrel{\text{def}}{\equiv} \mu Z(\delta, s, \delta', s').\Phi_{\text{trans}}(Z, \delta, s, \delta', s').$$

[8] This definition is a more detailed version of the definition proposed in [De Giacomo *et al.*, 1997b].

[9] If we want to use the concurrency operator, one more rule should be added to this definition.

The relation $Trans$ represents a single step transition. Its transitive closure, $Trans^*$, computes the many step transitions performed by the entire program. The extent of $Trans^*$ is constructed by the set of rules:

$$\Re_{\mathrm{tr}^*} = \left\{ \frac{\emptyset}{\langle \delta, s, \delta, s \rangle} \middle| \models_{\mathfrak{U}'} (\exists \delta', s').Trans(\delta, s, \delta', s') \vee (\exists \delta', s').Trans(\delta', s', \delta, s) \right\}$$

$$\cup \left\{ \frac{\langle \delta, s, \quad \delta', s' \rangle}{\langle \delta, s, \quad \delta'', s'' \rangle} \middle| \langle \delta, s, \quad \delta', s' \rangle \in Z, \ \models_{\mathfrak{U}'} \ Trans(\delta', s', \delta'', s'') \right\}.$$

$Trans^*$ is defined as the least fixed point of the operator Φ_{tr^*} constructed from the set of rules $\Re_{\mathrm{tr}^*}$. In fact, a Prolog implementation of the inductive definition of $Trans^*$, provided that the definition of $Trans$ is implemented as well, defines the interpreter of GOLOG. The query $(\exists \delta', s')$. $Trans^*(\delta_0, S_0, \delta, s)$ determines the execution trace of the program δ_0 starting at S_0. An example of GOLOG program together with an appropriate axiomatization of primitive actions can be found below. This program specifies the behavior of a robot serving coffee in an office environment.

7 Example of GOLOG program

A coffee server takes orders for coffee from customers. For each coffee order, a cup of coffee is served. We intentionally simplify the example and do not update the database of requests. A more complicated example with a queue of requests is considered in [De Giacomo *et al.*, 1997b]. We abstract from limitations of resources (i.e., coffee is always available) and the possibility of failures.

Primitive actions

- $request\_coffee(person)$: person requests coffee (an exogenous event).
- $pickup\_coffee$: the robot picks up a cup of coffee from the coffee machine.
- $give\_coffee(person)$: the robot gives a cup of coffee to $person$.
- $start\_go(loc1, loc2)$: the robot starts to go from $loc1$ to $loc2$.
- $end\_go(loc1, loc2)$: the robot ends the action of going from $loc1$ to $loc2$.

Fluents

- $robotLocation(CM, s)$: the robot's location in situation s. Locations are either coffee machine (CM) or a person's office.
- $has\_coffee(person, s)$: $person$ has coffee in situation s.
- $going(loc1, loc2, s)$: in situation s, the robot is going from $loc1$ to $loc2$.
- $holding\_coffee(s)$: the robot is holding a cup of coffee in situation s.
- $waiting\_coffee(person, s)$: $person$ is waiting for coffee after he or she has requested it.

Situation Independent Functions

- CM is a 0-ary function denoting the coffee machine's location.
- Sue, $Mary$, $Bill$ and Joe are constants denoting people.
- $office(person)$ denotes the office of $person$.

Successor State Axioms

$$\begin{gathered} robotLocation(loc2, do(a,s)) \equiv \\ a = end\_go(loc1, loc2) \wedge going(loc1, loc2, s) \\ \vee robotLocation(loc2, s) \wedge a \neq end\_go(l, ll), \end{gathered}$$

$$\begin{gathered} has\_coffee(person, do(a,s)) \equiv \\ a = give\_coffee(person) \wedge holding\_coffee(s) \\ \wedge robotLocation(office(person), s) \\ \vee has\_coffee(person, s), \end{gathered}$$

$$\begin{gathered} going(loc1, loc2, do(a,s)) \equiv \\ a = start\_go(loc1, loc2) \wedge \neg(\exists l, ll).\ going(l, ll, s) \\ \wedge loc1 \neq loc2 \wedge robotLocation(loc1, s) \\ \vee going(loc1, loc2, s) \wedge a \neq end\_go(loc1, loc2), \end{gathered}$$

$$\begin{gathered} holding\_coffee(do(a,s)) \equiv \\ a = pickup\_coffee \wedge robotLocation(CM, s) \wedge \neg holding\_coffee(s) \\ \vee holding\_coffee(s) \wedge a \neq give\_coffee(person), \end{gathered}$$

$$\begin{gathered} waiting\_coffee(person, do(a,s)) \equiv \\ a = request\_coffee(person) \\ \vee waiting\_coffee(person, s) \wedge a \neq give\_coffee(person). \end{gathered}$$

Initial situation

$$\begin{gathered} waiting\_coffee(Sue, S_0) \wedge waiting\_coffee(Bill, S_0) \\ \wedge has\_coffee(Joe, S_0) \wedge robot\_location(office(John), S_0). \end{gathered}$$

GOLOG procedures

$$\begin{gathered} Deliver\_coffee1 \doteq \texttt{while}\ (\exists person).\ waiting\_coffee(person, now) \\ \texttt{do}\ Serve\_a\_request;\ Goto(CM) \end{gathered}$$

$$\begin{gathered} Goto(loc2) \doteq (\pi loc).\ [robot\_location(loc, now)?; \\ start\_go(loc, loc2);\ end\_go(loc, loc2)] \end{gathered}$$

$$\begin{gathered} Serve\_coffee(person) \doteq Goto(CM);\ pickup\_coffee; \\ Goto(office(person));\ give\_coffee(person) \end{gathered}$$

$$\begin{gathered} Serve\_a\_request \doteq (\pi person).\ [waiting\_coffee(person, now)?; \\ Serve\_coffee(person)] \end{gathered}$$

8 Verification of GOLOG programs

The verification problem assumes two formal descriptions of behavior, S (specification) and I (implementation), and asks if I conforms with (satisfies, refines, implements) S. In our case, I is a GOLOG program, and S is a property this program must satisfy. Since both specification and implementation are written in the same language, a possibility of a mistake while translating from one language to another is excluded. Moreover, the view on the situation calculus presented in this paper provides a formal justification of applying known program verification techniques both of process algebra and of the propositional μ-calculus in a single formal framework. We next provide a brief discussion on the connections with these formal systems. At the end of this section we propose a method of verifying properties of GOLOG programs by computing fixed points of logical programs. This method is then demonstrated by an example in the next section.

GOLOG programs can be viewed as process algebra expressions. The rules of the inductive definition of $Trans$ can be understood as the rules of structural operational semantics for process algebra [Plotkin, 1981]. However, in process algebra the induction principle is implicit, whereas we use an explicit logical representation of this principle using a fixed point approach. Process algebras have been used for formal verification of nonterminating processes. Two terms of sort process, I and S, are considered to be equivalent if and only if one can be replaced by the other in any program without affecting the observed behavior of the program. A typical example of a behavioral equivalence is the bisimulation relation. To verify an implementation I, one has to prove that I is equivalent to S.

The μ-calculus, developed by [Park, 1970; Hitchcock and Park, 1973] and others, is a language for including inductive definitions with first order logic. We have shown that the second order definitions traditionally used in the situation calculus are expressible using the μ and ν operators of the μ-calculus, a language strictly intermediate in expressive power between first- and second-order logics.

Another logic [Pratt, 1981], [Kozen, 1983], [Emerson and Clarke, 1980] is closely connected to the inductive situation calculus and to the μ-calculus of [Park, 1970; Hitchcock and Park, 1973]. This logic is often referred to as propositional μ-calculus $L\mu$ and is interpreted with respect to a structure. This structure may be viewed as a transition system. The propositional μ-calculus provides a least fixed point operator (μ) and a greatest fixed point operator (ν) which make it possible to characterize temporal properties of programs. To obtain some intuition about temporal properties, consider the Computational Tree Logic CTL. This logic is weaker than $L\mu$; however, many interesting properties of programs are expressible in it. CTL allows basic temporal modalities of the following form: a path quantifier – either A ("for all futures") or E ("for some future") – followed by one of the linear time modalities, such as G ("always"), F ("sometime"), and X ("nexttime"). Note the following extremal fixed point characterizations for

CTL properties:

$$EF\,P \equiv \mu Z.[P \vee EX\ Z] \qquad EG\,P \equiv \nu Z.[P \wedge EX\ Z]$$
$$AG\,P \equiv \nu Z.[P \wedge AX\ Z] \qquad AF\,P \equiv \mu Z.[P \vee AX\ Z]$$

These temporal properties are expressible in the inductive situation calculus. Some examples of these properties can be found in [De Giacomo *et al.*, 1997b]. In the following section we define inductively and then prove a temporal property of the coffee delivery program which corresponds to the property of the form $EFhas\_coffee(p)$ of CTL.

Using the techniques of process algebra and the μ-calculus for proving properties of programs is not always necessary. Moreover, verification of temporal properties in $L\mu$ is restricted to the propositional case. We claim that proving temporal properties of GOLOG programs can be reduced to computing fixed points of logic programs. We use the facts that (i) a positive inductive definition can be encoded as a logic program, and (ii) the greatest fixed point operator ν is expressible using the least fixed point operator μ. This method is, of course, restricted to the case where the logic program which defines a property to be verified terminates.

9 Example of Verification

As an example, we consider a *response* property of our coffee delivery program. It refers to two assertions r and q, and requires that every configuration satisfying r is eventually followed by a configuration where q is true. Let $Response$ denote the set of triples such that $\langle p, \delta, s\rangle \in Response$ if and only if the following is true: if the person p is waiting for coffee in situation s, then it is always possible that the request will be satisfied eventually[10]. This property is sufficiently interesting to demonstrate our method without excessively cumbersome derivations. The set $Response$ is inductively defined by the set of rules:

$$\Re_{\text{response}} = \left\{ \frac{\emptyset}{\langle p, \delta, s\rangle} \middle| \models_{\mathfrak{U}'} \ Q(p, \delta, s) \right\}, \quad \text{where} \tag{8}$$

$$Q(p, \delta, s) \stackrel{\text{def}}{\equiv} [waiting\_coffee(p, s) \wedge s = S_0 \wedge \delta = DC$$
$$\vee Trans^*(DC, S_0, \delta, s) \wedge waiting\_coffee(p, s)] \wedge eventually\_has\_coffee(p, \delta, s).$$

Here $eventually\_has\_coffee$ means that the person p will have coffee sooner or later[11]. The formula induced by the inductive definition of $Response$ is $\Phi_{\text{response}}(Z, p, \delta, s) \equiv Z(p, \delta, s) \wedge Q(p, \delta, s)$. We are interested in the greatest set of configurations where the response property is true. We define $Response(p, \delta, s) \stackrel{\text{def}}{\equiv}$

[10] From now on, we use p, sometimes with tilde or prime, to range over persons; DC stands for $DeliverCoffee$.

[11] In the Computational Tree Logic (CTL) this formula would be written as $EF\ has\_coffee(p)$.

$\nu Z(p,\delta,s).\,\Phi_{\mathrm{response}}(Z,p,\delta,s)$. It is easy to check that by the definition of a greatest fixed point, $\nu Z(p,\delta,s).\ Z(p,\delta,s) \wedge Q(p,\delta,s) \equiv Q(p,\delta,s)$. Thus,

$$Response(p,\delta,s) \equiv Q(p,\delta,s) \equiv [waiting\_coffee(p,s) \wedge s = S_0 \wedge \delta = DC \\ \vee Trans^*(DC,S_0,\delta,s) \wedge waiting\_coffee(p,s)] \wedge eventually\_has\_coffee(p,\delta,s).$$

The set $eventually\_has\_coffee$ is defined inductively, for each particular p:

$$\Re_{\mathrm{ev}}(p) = \left\{ \frac{\emptyset}{\langle DC,S_0\rangle} \middle| \models_{\mathfrak{U}'} \ has\_coffee(p,S_0) \right\}$$
$$\cup \left\{ \frac{\emptyset}{\langle \delta,s\rangle} \middle| \models_{\mathfrak{U}'} \ has\_coffee(p,s) \wedge Trans^*(DC,S_0,\delta,s) \right\}$$
$$\cup \left\{ \frac{\langle \delta',s'\rangle}{\langle \delta,s\rangle} \middle| \langle \delta',s'\rangle \in X, \ \models_{\mathfrak{U}'} \ Trans(\delta,s,\delta',s') \right\}.$$

Thus, the set $eventually\_has\_coffee$ includes those configurations where the person has coffee and configurations preceding these. The corresponding formula is

$$\Phi^p_{\mathrm{ev}}(X,\delta,s) \equiv \delta = DC \wedge s = S_0 \wedge has\_coffee(p,s) \\ \vee has\_coffee(p,s) \wedge Trans^*(DC,S_0,\delta,s) \vee (\exists \delta',s').X(\delta',s') \wedge Trans(\delta,s,\delta',s').$$

We define $eventually\_has\_coffee(p,\delta,s)$ as the least fixed point of Φ^p_{ev}.

In practice, we often need to check the truth of a property at a particular configuration. A method similar to the one we would use for natural numbers, is applicable: to prove 2 is a natural number, we apply the first order construction principle for natural numbers; to prove -2 is not a natural number, we apply the induction principle. Let us see whether the response property is true for the initial configuration $\langle DC,S_0\rangle$. Since the response property is reducible to proving $eventually\_has\_coffee(p,DC,S_0)$ for those p such that $waiting\_coffee(p,S_0)$, we will apply the construction principle for this property,

$$\Phi^p_{\mathrm{ev}}(eventually\_has\_coffee,\delta,s) \equiv eventually\_has\_coffee(p,\delta,s).$$

We substitute the expression for Φ^p_{ev}:

$$eventually\_has\_coffee(p,\delta,s) \equiv \delta = DC \wedge s = S_0 \wedge has\_coffee(p,s) \\ \vee has\_coffee(p,s) \wedge Trans^*(DC,S_0,\delta,s) \\ \vee (\exists \delta',s').\ eventually\_has\_coffee(\delta',s') \wedge Trans(\delta,s,\delta',s').$$

By the definition of $Trans^*$, it follows that this formula is logically equivalent to

$$eventually\_has\_coffee(p,\delta,s) \equiv \\ (\exists \delta',s').\ has\_coffee(p,s') \wedge Trans^*(DC,S_0,\delta',s').$$

The easiest case is when a person already has coffee in the initial situation. From the description of the initial situation we conclude that Joe's request for coffee is already satisfied, $eventually\_has\_coffee(Joe,DC,S_0)$ is true.

Now we prove the response property for the unsatisfied requests held in S_0. It is easy to see that it reduces to proving the sentence:

$$(\exists\delta' s').\, Trans^*(DC, S_0, \delta', s') \wedge has\_coffee(Sue, s')$$
$$\wedge(\exists\delta' s').\, Trans^*(DC, S_0, \delta', s') \wedge has\_coffee(Bill, s').$$

The following configuration satisfies these two sentences:

$$\begin{aligned}
&s' = do(give_coffee(Bill), do(end_go(CM, office(Bill)),\\
&do(start_go(CM, office(Bill)), do(pickup_coffee,\\
&do(end_go(office(Sue), CM), do(start_go(office(Sue), CM),\\
&do(give_coffee(Sue), do(end_go(CM, office(Sue)),\\
&do(start_go(CM, office(Sue)), do(pickup_coffee,\\
&do(end_go(office(John), CM), do(start_go(office(John), CM), S_0)))))))))))),
\end{aligned}$$

$\delta' = nil$. The same configuration is obtained as a result of executing the following Prolog program, implementing the inductive definitions above.

```
tc(P,S,P,S).
tc(P,S,Pr,Sr):-trans(P,S,PP,SS), tc(PP,SS,Pr,Sr).
eventually_has_coffee(P,D1,S1) :-
              tc(pcall(deliverCoffee),s0,D1,S1), hasCoffee(P,S1).
eventually_has_coffee(P,D1,S1) :-
              trans(D1,S1,D2,S2), eventually_has_coffee(P,D2,S2).
response(P,D1,S1) :-
              waitingCoffee(P,S1), eventually_has_coffee(P,D1,S1).
```

In general, Prolog is not applicable for our purposes. However, in the case of a positive program as this one, it provides the correct answer.

10 Conclusion

We have presented the situation calculus from the general perspective of inductive definability. In particular, we have described an inductive solution to the frame and ramification problems. As we have argued, inductive definability provides a general framework both for reasoning about actions and for expressing properties of GOLOG programs. It offers the convenience of proving these properties without translating to another formal language. It also relates the situation calculus with process algebra and the μ-calculus, making the techniques of these formalisms applicable in the context of the situation calculus.

In addition to its usefulness in reasoning about actions and programs, the theory of inductive definability may be applicable to the formalization a common sense notion of causality. If we represent causal rules as rules of inductive definitions, then certain properties are easily obtained. For example, transitivity of causality follows simply from the transitivity of inductive relations (cf. [Moschovakis, 1974]).

We believe that the concept of inductive definability will facilitate investigation of the extent to which the techniques of logic programming can be applied to program verification. It is possible that it will provide new methods extending the range of properties provable automatically, especially in cases where complete information cannot be obtained. Moreover, the rich theories of actions may further extend the scope of provable properties, especially in new areas for program verification such as robotics.

Acknowledgments

I am grateful to Ric Hehner for the discussion we had about inductive principles and fixed points. This discussion was the starting point of this work. I am especially grateful to Ray Reiter for many advices concerning situation calculus structures. Thanks to the anonymous referees for giving a helpful feedback, and to Eric Harley for proofreading the paper.

References

[Aczel, 1977] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. Elsevier, 1977.

[Apt, 1990] K. R. Apt. Logic programming. In *Handbook of theoretical Computer Science*, volume 92, pages 493–574. Elsevier, 1990.

[Baeten and Weijland, 1990] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1990.

[Bell and Machover, 1977] J. Bell and M. Machover. *A Course in Mathematical Logic*. 1977.

[Cousot and Cousot, 1992] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Conference Record of the 19^{th} ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*, pages 83–94, New York, U.S.A., 1992. ACM Press.

[De Giacomo *et al.*, 1997a] G. De Giacomo, H. J. Levesque, and Y. Lespérance. Reasoning about concurrent executions, prioritized interrupts, and exogenous actions in the situation calculus. In *Proc. of IJCAI 97*, pages 1221–1226, 1997.

[De Giacomo *et al.*, 1997b] G. De Giacomo, E. Ternovskaia, and R. Reiter. Non-terminating processes in the situation calculus. In *Proc. of AAAI 97 workshop on Robots, Softbots, and Immobots: theories of action, planning and control*, pages 18–28, 1997.

[Emerson and Clarke, 1980] E. Emerson and E. Clarke. Characterizing correctness properties of parallel programs using fixpoints. volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer Verlag, 1980.

[Emerson, 1990] I.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. ESP, 1990.

[Hehner, 1993] E.C.R. Hehner. *A practical theory of programming*. Springer-Verlag, 1993.

[Hitchcock and Park, 1973] P. Hitchcock and D. Park. Induction rules and termination proofs. In *First Internat. Colloq. on Automata, Languages, and Programming*, pages 225–251, North-Holland, Amsterdam, 1973.

[Kozen, 1983] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Leivant, 1994] D. Leivant. Higher order logic. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 229–321. Elsevier, 1994.

[Levesque *et al.*, 1996] H.J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. Golog: a logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on actions and processes*, 4:655–678, 1996.

[Lin, 1995] F. Lin. Embracing causality in specifying the indirect effects of actions. In *Proc. of IJCAI 95*, pages 1985–1991, 1995.

[McIlraith, 1997] S.A. McIlraith. Representing actions and state constraints in model-based diagnosis. In *Proc. of AAAI 97*, pages 43–49, 1997.

[Milner, 1989] R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.

[Moschovakis, 1974] Y.N. Moschovakis. *Elementary Induction on Abstract Structures*. Amsterdam, North Holland, 1974.

[Park, 1970] D. Park. Fixpoint induction and proofs of program properties. In *Machine Intelligence*, volume 5, pages 59–78. Edinburgh University Press, 1970.

[Plotkin, 1981] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science department, Aarhus University,Denmark, 1981.

[Pratt, 1981] V. Pratt. A decidable μ-calculus (preliminary rept.). In *Proc. 22th IEEE Symp. on Foundations of Computer Science*, pages 421–427, 1981.

[Reiter, 1991] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[Reiter, 1993] R. Reiter. Proving properties of states in the situation calculus. *J. of Artificial Intelligence*, 64:337–351, 1993.

[Thielscher, 1996] M. Thielscher. Ramification and causality. Technical Report TR-96-003, International Computer Science Institute, Berkeley, CA 94704-1198, 1996.