

Reconciling Situation Calculus and Fluent Calculus

Stephan Schiffel and Michael Thielscher

Department of Computer Science
Dresden University of Technology
{stephan.schiffel,mit}@inf.tu-dresden.de

Abstract

The Situation Calculus and the Fluent Calculus are successful action formalisms that share many concepts. But until now there is no formal relation between the two calculi that would allow to formally analyze the relationship between the two approaches as well as between the programming languages based on them, Golog and FLUX. Furthermore, such a formal relation would allow to combine Golog and FLUX and to analyze which of the underlying computation principles is better suited for different classes of programs. We develop a formal translation between domain axiomatizations of the Situation Calculus and the Fluent Calculus and present a Fluent Calculus semantics for Golog programs. For domains with deterministic actions our approach allows an automatic translation of Golog domain descriptions and execution of Golog programs with FLUX.

Introduction

The Situation Calculus(SC) (McCarthy 1963; Reiter 2001b) and the Fluent Calculus(FC) (Thielscher 2005b) are action formalisms. These formalisms provide theories for intelligent systems to reason about actions and how they change the state of the world. Both calculi provide a different solution to the well known frame problem (McCarthy & Hayes 1969).

In the SC effects of actions are defined by successor state axioms (Reiter 1991), each one describing the value of an individual fluent (an atomic property of the world) after executing an action in terms of what holds in the situation before the action. The FC extends the SC by the concept of a *state* and in which the effects of actions are specified by action-based state update axioms (Thielscher 1999). Those axioms update an explicit description of the state of the world upon execution of an action. Because of these different approaches both calculi have different properties. Therefore one might be suited better than the other to solve certain classes of problems (Thielscher 2005a).

However there exists no proper analysis about that issue until now. For comparing the different properties of the SC and the FC a formal relation between the two calculi is necessary. We develop such a relation in terms of a translation

between domain descriptions in both calculi. Another benefit of a formal relation between the SC and the FC is the possibility to compare extensions made separately for the two calculi, such as concurrency. Furthermore it might even allow to translate current or future extensions made only for one of the calculi to the other.

Golog is a programming language for intelligent agents that combines elements from classical programming (conditionals, loops, etc.) with reasoning about actions. Primitive statements in Golog programs are *actions* to be performed by the agent. Conditional statements in Golog are composed of fluents. The execution of a Golog program requires to reason about the effects of the actions the agent performs, in order to determine the values of fluents when evaluating conditional statements in the program.

Existing semantics for Golog are based on the SC, and existing implementations (Levesque *et al.* 1997; Giacomo, Lespérance, & Levesque 2000) use successor state axioms when evaluating conditional statements in a Golog program. Accordingly, the implementations use pure regression to evaluate a fluent condition, which means that a given sequence of actions is rolled back through the successor state axioms. A consequence is that the evaluation of a condition in a Golog program in general depends on the length of the history and the number of fluents whose (past) values have an influence on the conditional statement. Alternatively, progression (Lin & Reiter 1997) can be used in combination with successor state axioms, which means to employ regression to infer the values of all fluents after an action, and then to store these values for future evaluation. This avoids to store an ever increasing history, but the computational effort of a single progression step depends on the overall number of fluents of a domain.

In this paper, we present an alternative semantics for Golog based on the FC, to overcome this disadvantage. Our new semantics lays the foundation for an implementation of Golog in FLUX (Thielscher 2005a), a programming language based on the the FC. In FLUX the inference principle of progression is employed to update a state specification upon the performance of an action. The advantage over regression is that fluent conditions can be evaluated directly against the updated world model. Moreover, progression via state update axioms requires to consider the affected fluents only.

The implementation of a Golog interpreter in FLUX adds an implementation independent programming language to the FLUX system, a feature that FLUX was lacking so far. This allows to conduct systematic experiments and to analyze which underlying computation principle, progression or regression, is better suited for different classes of Golog programs.

The remainder of the paper is organized as follows. First we repeat the basic definitions of the Situation Calculus and Golog. We use a variant of Golog that extends the original version by a search operator, which allows to interleave planning and execution (Giacomo, Lespérance, & Levesque 2000). We also briefly recapitulate the Fluent Calculus. Next we present the formal translation of a SC domain description to an equivalent one in the Fluent Calculus. After this, we present our Fluent Calculus semantics for Golog programs. A brief discussion of the results and possible future work concludes the paper.

Background

Situation Calculus

The SC as described in (Reiter 2001b) is a sorted logical language designed for axiomatizing dynamic domains. There are predefined sorts for fluents, actions and situations. All changes to the environment are caused by *actions*. A sequence of actions is called a *situation*. Situations are built using the situation constant S_0 and the function $Do(a, s)$, where a is an action term and s is a situation.

Non-static properties of the world are described by fluents, i.e. predicates or functions with a situation term as last argument. Properties of a situation s are described by so called *uniform formula in s* . A formula is uniform in a situation s if it does not mention any other situation besides s , that means it can be evaluated with regard to situation s only. The values of the fluents in the initial situation are described by the *initial database*, a set of formulas uniform in S_0 . In situation $Do(a, s)$ the values of fluents are defined in terms of the values of fluents in the previous situation s and the action a that was executed by so called *successor state axioms* (Reiter 1991). A successor state axiom for a fluent predicate $F(\vec{x}, s)$ looks like this:

$$F(\vec{x}, Do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg \gamma_F^-(\vec{x}, a, s)) \quad (1)$$

Here $\gamma_F^+(\vec{x}, a, s)$ and $\gamma_F^-(\vec{x}, a, s)$ are formulas uniform in s with free variables among \vec{x}, a . If $\gamma_F^+(\vec{x}, a, s)$ holds then the fluent $F(\vec{x})$ is called a positive effect of action a in situation s . Consequently, the fluent is a negative effect of a , if $\gamma_F^-(\vec{x}, a, s)$ holds. Thus, successor state axioms describe the effects of actions. An example is the following successor state axiom for a fluent called $Closed(s)$, indicating the status of a door:

$$\begin{aligned} Closed(Do(a, s)) &\equiv & (2) \\ a = Close &\vee (Closed(s) \wedge \neg a = Open) \end{aligned}$$

The execution of an action a in situation s is only possible if its preconditions are fulfilled in s . The preconditions are defined by a set of precondition axioms of the form

$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$, where Π_A is a formula uniform in s .

Possibility of actions leads to the definition of an *executable situation* (Reiter 2001b, 4.2.4) $executable(s)$, which intuitively means a situation s is executable if each action in the history of s was possible in the respective situation.

Golog

We consider the original Golog defined in (Levesque *et al.* 1997) augmented by the search operator introduced in (Giacomo & Levesque 1999).

Being a high-level language for agent control, Golog uses actions (of the agent) as primitive statements and fluents for tests, i.e., conditional statements. These basic ingredients are embedded in a language that has standard elements of imperative programming. Specifically, a Golog program can be composed of these constructs:¹

nil	empty program
a	action
$\phi?$	test
$\delta_1; \delta_2$	sequence
$\delta_1 \mid \delta_2$	nondeterministic choice (of programs)
$\pi v. \delta$	nondeterministic choice (of parameters)
δ^*	nondeterministic iteration
$\Sigma \delta$	search

In addition, the following macros are used:

$$\begin{aligned} \text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 &\stackrel{\text{def}}{=} (\phi?; \delta_1) \mid (\neg \phi?; \delta_2) \\ \text{while } \phi \text{ do } \delta &\stackrel{\text{def}}{=} (\phi?; \delta)^*; \neg \phi? \end{aligned}$$

The intuitive meaning of the operator $\Sigma \delta$ is to search for a terminating run of sub-program δ and then to execute this run.²

Existing semantics for Golog are based on the SC. In (Giacomo, Lespérance, & Levesque 2000), a transition semantics for Golog is given by an axiomatic definition of two predicates: $Trans(\delta, s, \delta', s')$, meaning that the execution of the next action or the next test in program δ leads from situation s to situation s' and to the remaining program δ' . The second predicate, $Final(\delta, s)$, means that program δ does not require to execute any more action or test in situation s . The core of this semantics are the definitions for executing an action and for evaluating a test:

$$\begin{aligned} Trans(a, s, \delta', s') &\equiv Poss(a, s) \wedge \\ &\quad \delta' = nil \wedge s' = Do(a, s) \\ Trans(\phi?, s, \delta', s') &\equiv \phi[s] \wedge \\ &\quad \delta' = nil \wedge s' = s \end{aligned} \quad (3)$$

Here, $Poss(a, s)$ denotes that action a is possible in situation s , and $\phi[s]$ means that condition ϕ holds in situation

¹Below, $\delta, \delta_1, \delta_2$ are Golog programs, ϕ is a formula with fluents as atoms, and v is a variable.

²The Golog variant of (Giacomo, Lespérance, & Levesque 2000) contains additional constructs, e.g., for interrupts and procedures, which we will not deal with in this paper for the sake of simplicity.

s. This requires a background theory which contains action knowledge of the application domain in form of precondition and effect axioms. Specifically, effects are described by successor state axioms (Reiter 1991), which define the value of a particular fluent after an action (that is, in situation $Do(a, s)$), in terms of the values of fluents in situation *s*.

Fluent Calculus

The FC extends the SC by a predefined sort for *states*. The function $State(s)$ denotes the state in situation *s*. The world can be in the same state in different situations, but the state in every situation is unique. State terms are constructed from fluents (as singleton states) and the function $z_1 \circ z_2$, where z_1 and z_2 are states. The foundational axioms of the FC stipulate that function “ \circ ” shares essential properties with the union operation for sets. A fluent *f* is then defined to hold in a state *z* if and only if the former is a sub-state of the latter:

$$Holds(f, z) \stackrel{\text{def}}{=} (\exists z') z = f \circ z'$$

The addition of states to the SC allows to define fluents to hold in a situation, written $Holds(f, s)$, by referring to the state in the situation:

$$Holds(f, s) \stackrel{\text{def}}{=} Holds(f, State(s))$$

Based on the notion of a state, the frame problem (McCarthy & Hayes 1969) is solved in the FC by state update axioms, which define the effects of an action *a* as the difference between the state prior to the action, $State(s)$, and the successor $State(Do(a, s))$. A state update axioms for action $A(\vec{x})$ looks like this:

$$\begin{aligned} Poss(A(\vec{x}), s) \supset \\ (\exists \vec{y}_1)(\Delta_1(s) \wedge State(Do(A(\vec{x}), s)) = \\ State(s) - \vartheta_1^- + \vartheta_1^+) \\ \vee \dots \vee \\ (\exists \vec{y}_n)(\Delta_n(s) \wedge State(Do(A(\vec{x}), s)) = \\ State(s) - \vartheta_n^- + \vartheta_n^+) \end{aligned} \quad (4)$$

Where ϑ_i^- and ϑ_i^+ are the negative and positive effects of the action $A(\vec{x})$ in situation *s* under the condition that $\Delta_i(s)$ holds, and \vec{y}_i are free variables of ϑ_i^- , ϑ_i^+ and $\Delta_i(s)$. $\Delta_i(s)$ is a so-called *situation formula*, that is a formula with free situation variable *s* and without any occurrence of states or situations other than in expressions of the form $Holds(f, s)$ and without actions.

In contrast to successor state axiom in the SC, state update axioms provide an action-based description of the effects of actions as opposed to a fluent based description. That means inference is done by updating the entire state rather than individual fluents. This leads directly to an efficient implementation that is based on the principle of progression (Thielscher 2003). The following state update axioms describe the effects of the actions *Open* and *Close* for opening and closing a door, whose status is described by the

one fluent *Closed*:

$$\begin{aligned} Poss(Open, s) \supset \\ State(Do(Open, s)) = State(s) - Closed \quad (5) \\ Poss(Close, s) \supset \\ State(Do(Close, s)) = State(s) + Closed \end{aligned}$$

FLUX

The FC provides the formal underpinnings of the logic programming method FLUX (Thielscher 2005a). FLUX is based on the encoding of (possibly incomplete) state knowledge with the help of constraints. The effects of actions are inferred on the basis of state update axioms, which effect an update of the set of constraints that encode a given state. An example is the following state update axiom for the action *close* of closing the door of an elevator:

$$\begin{aligned} \text{state_update}(Z1, \text{close}, Z2) :- \\ \text{update}(Z1, [\text{closed}], [], Z2). \end{aligned}$$

where $update(z_1, \vartheta^+, \vartheta^-, z_2)$ means that z_2 is z_1 updated by, respectively, positive and negative effects ϑ^+ and ϑ^- . FLUX is thus amenable to the computation principle of progression: The current state description is updated upon the performance of an action, which allows to evaluate conditions on the successor situation directly against the new state.

Translation of Domain Specifications

Domain specifications Σ in the Situation Calculus and the Fluent Calculus consist of:

- Σ_{fd} , the foundational axioms of the SC or the FC respectively,
- Σ_{init} , a definition of the initial situation,
- Σ_{ap} , a set of action precondition axioms,
- Σ_{dc} , a set of domain constraints,
- Σ_{effect} , a set of axioms describing the effects of the actions,
- Σ_{una} , a set of unique names axioms and
- Σ_{aux} , a set of auxiliary axioms, i.e. situation independent axioms.

In the following we will use a superscript of *s* or *f* to distinguish between elements of the SC and the FC domains with the same name but different definitions. The functions *t* and *t'* are used to denote a translation of a SC formula to the FC and vice versa.

Because of space constraints we will only present the translations of Σ_{init} and Σ_{effect} in this paper. The translation of the remaining elements is mostly straightforward.

For the translation we make certain assumptions about the domain descriptions:

- We will only consider deterministic domains, i.e. there are no nondeterministic actions.
- The actions must not have infinitely many effects, so called open effects, because one can't describe open effects in a standard state update axiom of the FC (Thielscher 1999).

- The successor state axioms have to be of the form as in equation 1.³
- For the sake of simplicity we don't consider domain descriptions with fluent functions in this paper. Fluent functions can be part of a SC domain description, but can only be indirectly expressed in the FC by fluent relations with the value of the fluent function as additional argument. A straightforward translation of a domain description with fluent functions to one without is possible.

Situation Formulas and Initial Situations

Situation formulas in the Fluent Calculus and so called uniform formulas in the Situation Calculus are similar in that both state facts about exactly one situation. A uniform formula ϕ can be translated to a situation formula $t(\phi)$ by replacing each fluent atom $P(\vec{x}, s)$ by $Holds(P(\vec{x}), s)$ (Thielscher 1999) and vice versa.

The initial situation in the SC is defined by a set of first order sentences that are uniform in S_0 , the so called initial database. In a FC domain the initial situation is defined by a situation formula $\Psi(S_0)$. W.l.o.g. we can assume that the initial database consists only of the single uniform formula $\Phi(S_0)$. Hence the translation of the initial situation descriptions reduces to the translation of a uniform formula to a situation formula and vice versa.

Effects of Actions

(Thielscher 1999) gives an in-depth account on how to translate successor state axioms to essentially equivalent state update axioms.

We reverse this method to translate state update axioms to equivalent successor state axioms. Consider a state update axiom as in equation 4. From this we can extract positive and negative effect axioms for action A :

$$\begin{aligned}\epsilon_{A,F_i}^+ &\supset F_i(\vec{y}_i, Do(A(\vec{x}), s)) \\ \epsilon_{A,F_i}^- &\supset \neg F_i(\vec{y}_i, Do(A(\vec{x}), s))\end{aligned}$$

with

$$\begin{aligned}\epsilon_{A,F_i}^+ &\stackrel{\text{def}}{=} \epsilon_{A,F_i}^+(\vec{x}, \vec{y}_i, s) \stackrel{\text{def}}{=} \bigvee_{j \in I^+} [t'(\Delta_j(s))] \\ \epsilon_{A,F_i}^- &\stackrel{\text{def}}{=} \epsilon_{A,F_i}^-(\vec{x}, \vec{y}_i, s) \stackrel{\text{def}}{=} \bigvee_{j \in I^-} [t'(\Delta_j(s))]\end{aligned}$$

The sets I^+ and I^- are the sets of indices of those ϑ_j^+ and ϑ_j^- , that contain $F(\vec{y}_i)$. That means ϵ_{A,F_i}^+ is the disjunction of all those conditions Δ_j for which $F(\vec{y}_i)$ is a positive effect of the action A , and similarly for ϵ_{A,F_i}^- . It is easy to see that if the state update axiom is built from effect axioms as described in (Thielscher 1999), then the disjunction of Δ_j is equivalent to the original effect axiom. In our example

of the door we obtain the following effect axioms from the state update axiom (5):

$$\begin{aligned}\top &\supset \neg Closed(Do(Open, s)) \\ \top &\supset Closed(Do(Close, s))\end{aligned}$$

Now we have positive and negative effect axioms for each pair of action A_j and fluent F_i . Those can be combined to general effect axioms for each fluent:

$$\begin{aligned}\epsilon_{F_i}^+ &\supset F_i(\vec{x}_i, Do(a, s)) \\ \epsilon_{F_i}^- &\supset \neg F_i(\vec{x}_i, Do(a, s))\end{aligned}$$

where $\epsilon_{F_i}^+$ and $\epsilon_{F_i}^-$ are defined as follows:

$$\begin{aligned}\epsilon_{F_i}^+ &\stackrel{\text{def}}{=} \epsilon_{F_i}^+(\vec{x}_i, a, s) \stackrel{\text{def}}{=} \bigvee_j [\epsilon_{A_j, F_i}^+ \wedge a = A_j(\vec{x})] \\ \epsilon_{F_i}^- &\stackrel{\text{def}}{=} \epsilon_{F_i}^-(\vec{x}_i, a, s) \stackrel{\text{def}}{=} \bigvee_j [\epsilon_{A_j, F_i}^- \wedge a = A_j(\vec{x})]\end{aligned}$$

For our example the general effect axioms are:

$$\begin{aligned}a = Open &\supset \neg Closed(Do(a, s)) \\ a = Close &\supset Closed(Do(a, s))\end{aligned}$$

The general effect axioms for each fluent F_i can be combined to successor state axioms:

$$F_i(\vec{x}_i, Do(a, s)) \equiv \epsilon_{F_i}^+ \vee (F_i(\vec{x}_i, s) \wedge \neg \epsilon_{F_i}^-)$$

The successor state axiom for our example is exactly the one of equation (2).

Equivalence of Domain Specifications

The domain specifications obtained by the translation described above are equivalent, in the sense that each formula about a situation s holds under a domain specification exactly if the translated formula holds under the translated domain specification, as long as the situation is executable, i.e. the actions leading to the situation were possible.

Theorem 1.

$$\begin{aligned}\Sigma_s \models \phi(s) &\quad \text{iff} \quad t(\Sigma_s) \models t(\phi(s)) \\ \text{and} & \\ \Sigma_f \models \psi(s) &\quad \text{iff} \quad t'(\Sigma_f) \models t'(\psi(s))\end{aligned} \tag{6}$$

for each uniform formula $\phi(s)$, each situation formula $\psi(s)$ and each executable situation s .

The restriction to executable situations is necessary, because in the FC the successor state is undefined if an action is not possible, i.e. $State(s)$ is undefined if s is not executable.

Proof (Sketch). For $s = S_0$ (6) follows by induction over all formulas uniform in s or situation formulas in s respectively. As the translation of state update axioms to successor state axioms is the reversal of the method from (Thielscher 1999), for $s = Do(a, s')$ (6) follows from (Thielscher 1999), provided s' is an executable situation, a is possible in s' and (6) holds for s' . \square

³This differs from the definition in (Reiter 2001b), which allows for arbitrary formulas uniform in s on the right hand side of the axiom.

Fluent Calculus Semantics for Golog Programs

Our starting point for defining our new semantics is the transition semantics for Golog given in (Giacomo, Lespérance, & Levesque 2000) with the exception of the axioms for concurrency. With the help of the predicates *Trans* and *Final* the semantics of a Golog program can be defined as:

$$Do(\delta, s, s') \stackrel{\text{def}}{=} (\exists \delta') Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$$

where $Trans^*$ stands for the reflexive and transitive closure of *Trans*. The predicate $Do(\delta, s, s')$ means that the execution of a Golog program δ in situation s leads to a final situation s' with a finite number of transitions.

In the FC it is now possible to replace the situation s of a configuration by its associated state $State(s)$ and thus denote transitions by $Trans(\delta, z, \delta, z')$ and final configurations by $Final(\delta, z)$ where $z = State(s)$ and $z' = State(s')$. By doing this we do no longer have to calculate values of fluents by regression over the situation as in the Situation Calculus. Instead we progress the state on every execution of an action. Then a simple lookup in the FC state is sufficient to acquire a fluents value, as, by the Completeness Assumption, all fluents that hold in a situation are part of the associated state. By replacing situations by states, however, we lose the information about the performed actions. This information is necessary in order to be able to actually execute the actions in the physical environment after reasoning about their outcomes. Thus we introduce a list of actions not executed so far as additional result of a transition and will therefore denote a transition between two configurations by $Trans(\delta, z, \delta', z', h')$ where h' is the history of the actions to be executed in z in order to reach z' .

Now we define the semantics of a Golog program in the FC by:

$$Do(\delta, z, z', h') \stackrel{\text{def}}{=} (\exists \delta') Trans^*(\delta, z, \delta', z', h') \wedge Final(\delta', z')$$

where $Trans^*$ stands for the reflexive and transitive closure of *Trans* and h' is the concatenation of the action histories of the individual transitions. Similar to the original semantics, the predicate $Do(\delta, z, z', h')$ means that the execution of a Golog program δ in state z results in state z' and h' contains the actions executed in between.

Now we can define the relations *Trans* and *Final* inductively for all Golog programs. Most definitions are essentially just syntactical transformations of the original ones from (Giacomo, Lespérance, & Levesque 2000): Situations are replaced by states, and the action history h' is added to the *Trans* predicate. E.g. the definition of *Trans* and *Final* for nondeterministic branches are those:

$$\begin{aligned} Trans(\delta_1 | \delta_2, z, \delta', z', h') &\equiv \\ Trans(\delta_1, z, \delta', z', h') \vee Trans(\delta_2, z, \delta', z', h') \\ Final(\delta_1 | \delta_2, z) &\equiv Final(\delta_1, z) \vee Final(\delta_2, z) \end{aligned}$$

The major differences arise in the definitions for primitive actions and test actions. These *Trans* predicates in the FC

are:

$$\begin{aligned} Trans(a, z, \delta', z', h') &\equiv \\ Poss(a, z) \wedge \delta' = nil \wedge (\exists s) State(s) = z \wedge \\ z' = State(Do(a, s)) \wedge h' = a \\ Trans(\phi?, z, \delta', z', h') &\equiv \\ \phi[z] \wedge \delta' = nil \wedge z' = z \wedge h' = [] \end{aligned}$$

Here $\phi[z]$ is an abbreviation for a predicate $HoldsCond(\phi, z)$ which maps Golog condition expressions to FC state formulas, by replacing each Fluent f in ϕ with $Holds(f, z)$. $HoldsCond(\phi, z)$ is similar to $Holds$ from (Giacomo, Lespérance, & Levesque 2000), it is only renamed to avoid confusion with $Holds$ of the FC.

The major difference compared to equation 3 is that in the SC $s' = Do(a, s)$ is just a variable assignment but in the FC

$$(\exists s) State(s) = z \wedge z' = State(Do(a, s))$$

results in a state update which calculates the new state z' from z and the effects of executing the action a in z .

This first of all means that computing a transition for a primitive action with our semantics is more expensive in terms of calculation time than with the original semantics. The reward for this is that the evaluation of $\phi[s]$ in the SC means to do a regression over the situation for each fluent f in ϕ and for each fluent on which the value of f depends. In contrast, in the FC $\phi[z]$ can be evaluated by looking up the values of the fluents in the state z that was computed by the last transition.

Thus calculating a transition for test actions in the FC does not depend on the length of the action history and is therefore less expensive in cases with situations of a certain length.

The new semantics is equivalent to the original one in the following sense:

Theorem 2. For all Golog programs δ and ground situation terms s and s'

$$\Sigma^s \cup C^s \models Do(\delta, s, s') \text{ iff}$$

$$\Sigma^f \cup C^f \models Do(\delta, State(s), State(s'), history(s, s'))$$

Where $history(s, s')$ denotes the list of actions executed between s and s' and C stands for the set of axioms for *Trans* and *Final* plus those axioms needed for encoding Golog programs as terms in first-order logic.

Proof (Sketch). For each Golog program δ it can be shown that

$$\Sigma^s \cup C^s \models Trans(\delta, s, \delta', s') \text{ iff} \quad (7)$$

$$\Sigma^f \cup C^f \models Trans(\delta, State(s), \delta', State(s'), history(s, s'))$$

and similarly for $Final(\delta, s)$. For test actions, i.e. $\delta = \phi?$, (7) follows from the definitions of $HoldsCond(\phi, z)$ and Theorem 1. For primitive actions, i.e. $\delta = a$, (7) follows from Theorem 1. For complex actions (e.g. $\delta_1; \delta_2$) the property is proved by induction. \square

Conclusion

We have presented a formal translation between domain axiomatizations in the Situation Calculus and the Fluent Calculus. This formal relation between the two action formalisms provides the necessary foundations for a proper analysis of the differences of both calculi and the agent programming systems based on them. It is also the basis for comparing extensions made separately for both calculi, such as concurrency, and it might even allow to translate current or future extensions made only for one of the calculi to the other.

We have also presented a new semantics for Golog based on the Fluent Calculus, by which the standard model for Golog is enriched with the notion of a state. The essential difference to previous semantics is that states, and not situations (i.e. histories of actions) are propagated when describing the execution of a Golog program. We have given a Fluent Calculus interpretation for all language elements of the original Golog (as defined by (Levesque *et al.* 1997)) augmented by the search operator introduced in (Giacomo, Lespérance, & Levesque 2000). All further constructs from the latter, extended dialect, e.g. those for interrupts and procedures, can be interpreted in our semantics in a straightforward way. The new semantics is proved to be equivalent with the original Situation Calculus semantics.

Our new semantics lays the foundation for interpreting Golog programs in FLUX. This allows to employ states and the inference principle of progression for state update. The motivation for the alternative semantics and implementation is that

- progression of states allows to evaluate conditions in Golog programs directly against the updated state;
- progression via state update axioms only requires to consider those fluents that are affected by an action;
- Golog provides an attractive, implementation independent programming language for agent systems, a feature that FLUX was lacking so far.

Based on the formal translation and the semantics defined in the previous sections we have developed a Prolog implementation of Golog interpreter in Flux⁴ based on the LeGolog interpreter⁵ as well as an implementation of the translation of a LeGolog domain description to Flux and vice versa. In an extended version of a standard scenario for Golog (Levesque *et al.* 1997), our new implementation proved to be a more efficient interpreter for Golog compared to the original LeGolog implementation. The results will be published on our webpage.

For future work, we intend to conduct systematic experiments and to analyze which computation principle is better suited for different classes of Golog programs. We are currently extending the translation of domain axiomatizations and our interpreter to knowledge-based Golog programs (Reiter 2001a), thereby using the full expressiveness of the Fluent Calculus and FLUX for incomplete states. This will allow us to use the existing constraint solving mechanism

for incomplete states in FLUX, which seems to be very efficient in comparison to other approaches (Sardina & Vassos 2005), to reason about knowledge in Golog programs.

References

- Giacomo, G. D., and Levesque, H. 1999. An incremental interpreter for high-level programs with sensing. In Levesque, H., and Pirri, F., eds., *Logical Foundations for Cognitive Agents*. Springer. 86–102.
- Giacomo, G. D.; Lespérance, Y.; and Levesque, H. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1–2):109–169.
- Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1–3):59–83.
- Lin, F., and Reiter, R. 1997. How to progress a database. *Artificial Intelligence* 92:131–167.
- McCarthy, J., and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence* 4:463–502.
- McCarthy, J. 1963. *Situations and Actions and Causal Laws*. Stanford University, CA: Stanford Artificial Intelligence Project, Memo 2.
- Reiter, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., ed., *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press. 359–380.
- Reiter, R. 2001a. On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logic* 2(4):433–457.
- Reiter, R. 2001b. *Knowledge in Action*. MIT Press.
- Sardina, S., and Vassos, S. 2005. The Wumpus World in INDIGOLOG: A Preliminary Report. The Sixth Workshop on Nonmonotonic Reasoning, Action, and Change at IJCAI.
- Thielscher, M. 1999. From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence* 111(1–2):277–299.
- Thielscher, M. 2003. Controlling semi-automatic systems with FLUX. (Extended abstract). In Palamidessi, C., ed., *Proceedings of the International Conference on Logic Programming (ICLP)*, volume 2916 of LNCS, 515–516. Mumbai, India: Springer.
- Thielscher, M. 2005a. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming* 5(4–5):533–565.
- Thielscher, M. 2005b. *Reasoning Robots: The Art and Science of Programming Robotic Agents*, volume 33 of *Applied Logic Series*. Kluwer.

⁴see <http://www.fluxagent.org/>

⁵<http://www.cs.toronto.edu/cogrobo/Legolog/>