Reasoning about Action and Change Using Dijkstra's Semantics for Programming Languages: Preliminary Report*

Witold Lukaszewicz and Ewa Madaliriska-Bugaj

Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warsaw, POLAND
witlu@mimuw.edu.pl, ewama@mimuw.edu.pl

Abstract

We apply Dijkstra's semantics for programming languages to formalization of reasoning about action and change. The basic idea is to view actions as formula transformers, i.e. functions from formulae into formulae.

The major advantage of our proposal is that it is very simple and more effective than most of other approaches. Yet, it deals with a broad class of actions, including those with random and indirect effects. Also, both temporal prediction and postdiction reasoning tasks can be solved without restricting initial nor final states to completely specified.

1 Introduction

We apply Dijkstra's semantics for programming languages [Dijkstra, 1976; Dijkstra and Scholten, 1990] to formalization of reasoning about action and change. The basic idea is to specify effects of actions in terms *of formula transformers,* i.e. functions from formulae into formulae. More specifically, with each action *A* we associate two formula transformers, called the *strongest postcondition for A* and the *weakest liberal precondition for A.* The former, when applied to a formula a, returns a formula representing the set of all states that can be achieved by starting execution of *A* in some state satisfying *a.* The latter, when applied to a formula *a,* returns a formula providing a description of all states such that whenever execution of *A* starts in any one of them and terminates, the output state satisfies a.[1]

The idea of employing formula transformers to specify effects of actions is not new in the AI literature. Waldinger [1977], in the context of STRIPS system [Fikes and Nilsson, 1971], introduces a notion of a repression operator which corresponds closely to the weakest precondition transformer. Pednault [1986; 1988; 1989] employes regression operators for plan synthesis. In [Pednault, 1986] a notion of progression operator, corresponding to Dijkstra's strongest postcondition, is introduced and analysed.

Formula transformers approach to reasoning about action and change has one major advantage and one major weakness when compared to purely logical formalisms such as Situation Calculus [Hayes and McCarthy, 1969; Lifschitz, 1988; Lifschitz and Rabinov, 1989; Gelfond *et a*/., 1991; Baker, 1991] or Features and Fluents [Sandewall, 1994]. On the positive side, describing effects of actions in terms of formula transformers decreases computational complexity. The price to pay for it is the loss of expressibility.

Our proposal combines computational effectiveness with expressibility. Although not so expressible as Situation Calculus or Feature and Fluents, the formalism specified here allows to deal with a broad class of actions, including those with random and indirect effects. Also, both temporal prediction and postdiction reasoning tasks can be solved without restricting initial nor final states to completely specified.[2]

The paper is organized as follows. Section 2 is a brief introduction to Dijkstra's semantics for a simple programming language. In section 3, we outline a general procedure to define action languages using Dijkstra's methodology, illustrate this procedure by specifying a simple "shooting" language, and introduce a notion of an *action scenario.* Section 4 defines the kind of reasoning we shall be interested in, and provides a simple method of realizing this type of inference. In section 5, we illustrate this method by considering a number of examples, well-known from the AI literature. Section 6 is devoted to actions with indirect effects. Finally, section 7 contains discussion and ideas for future work.

For lack of space, we omit proofs of the results provided here. The full version of the paper is available in pub/papers/CRIT/Dijkstra/ijcai95.ps by anonymous ftp to ftp.mimuw.edu.pl.

*This research was supported in part by the ESPRIT Basic Research Action No. 6156 - DRUMS II and by KBN grant 3 P406 019 06.

[1] We do not use the *weakest precondition* transformer *(wp)* which plays a prominent role in reasoning about programs. The reason is that, in general, the *wp* transformer is slightly too strong for our purposes.

[2] This paper is part of a general programme of applying Dijkstra's approach to reasoning about action and change. In [Lukaszewicz and Madalinska-Bugaj, 1994], we used this methodology to formalize deterministic actions without indirect effects. In [Lukaszewicz and Madaliriska-Bugaj, 1995], we combined Dijkstra's semantics with Reiter's default logic to deal with actions where abnormal effects are allowed.

2 Introduction to Dijkstra's semantics

In [Dijkstra and Scholten, 1990] we are provided with a simple programming language whose semantics is specified in terms of formula trasformers. More specifically, with each command 5 there are associated three such transformers, called the *weakest precondition,* the *weakest liberal precondition* and the *strongest postcondition,* denoted by *wp.S, wlp.S* and *sp.S,* respectively. Before providing the meaning of these transformers, we have to make some remarks and introduce some terminology.

We assume here that the programming language under consideration contains one type of variables only, namely Boolean variables. This assumption may seem overly restrictive, but no other variables will be needed for our purpose.

Let V be a set of Boolean variables. A *state over* V is any function σ from V into the truth-values $\{0,1\}$.

An *assertion language* over a set V of Boolean variables, denoted by $\mathcal{L}(V)$, is the set of all formulae constructable in the usual way from members of V, truth-constants T and F, sentential connectives $(\neg, \supset, \wedge, \vee, \equiv)$ and quantifiers $(\forall, \exists)$.[3] In what follows, the term 'formula' refers always to a formula of some fixed assertion language. A formula α is said to be a *Boolean expression* if it contains no quantifiers. If β is a formula, $\alpha_1, \ldots, \alpha_n$ are Boolean expressions and $x_1, \ldots, x_n$ are variables, then we write $\beta[x_1 \leftarrow \alpha_1, \ldots, x_n \leftarrow \alpha_n]$ to denote the formula which obtains from β by simultaneously replacing all free occurrences of $x_1, \ldots, x_n$ by $\alpha_1, \ldots, \alpha_n$, respectively.

The *value of a formula* $\alpha \in \mathcal{L}(V)$ *in a state* σ, written $\| \alpha \|_\sigma$, is an element from $\{0,1\}$ specified by the usual rules for classical propositional logic, together with:

(i) $\| \forall x.\alpha \|_\sigma = 1$ iff $\| \alpha \|_{\sigma'} = 1$, for each σ' identical to σ except on the variable x.

(ii) $\| \exists x.\alpha \|_\sigma = 1$ iff $\| \alpha \|_{\sigma'} = 1$, for some σ' identical to σ except on the variable x.[4]

A state σ is said to *satisfy* a formula α iff $\| \alpha \|_\sigma = 1$. If σ satisfies α, then σ is called a *model* of α. When we say that σ satisfies α, we always implicitly assume that σ provides an interpretation for all Boolean variables occurring in α.

The formula transformers mentioned above are to be understood as follows. For each command S and each formula α: (1) $wp.S(\alpha)$ is the formula whose models are precisely the states such that execution of S begun in any one of them is guaranteed to terminate in a state satisfying α. (2) $wlp.S(\alpha)$ is the formula whose models are precisely the states such that whenever execution of S starts in any one of them and terminates, the output state satisfies α. (3) $sp.S(\alpha)$ is the formula whose models are precisely the states such that each of them can be reached by starting execution of S in some state satisfying α.

In what follows, for each formula transformer $X \in \{wp, wlp, sp\}$, we shall write $X(S,\alpha)$ instead of $X.S(\alpha)$.

2.1 List of commands

The considered language consists of *skip* command, *assignment* to simple variables, *alternative* command and *sequential composition* of commands[5]. Semantics of these commands is specified as follows.[6]

1. **The *skip* command.** This is the "empty" command in that its execution does not change the computation state. The semantics of *skip* is given by

$$wlp(skip,\alpha) = sp(skip,\alpha) = \alpha.$$

2. **The *assignment* command.** This command is of the form $x := e$, where x is a (Boolean) variable and e is a (Boolean) expression. The weakest liberal precondition of this command is given by

$$wlp(x := e,\ \alpha) = \alpha[x \leftarrow e].$$

The strongest postcondition for assignment command is more complex. In general, it is given by

$$sp(x := e,\alpha) = \exists y.((x \equiv e[x \leftarrow y]) \wedge \alpha[x \leftarrow y]). \quad (1)$$

If x does not occur in e, (1) can be simplified. In this case $sp(x := e,\alpha)$ can be replaced by

$$(x \equiv e) \wedge \exists x.\alpha$$

or equivalently

$$(x \equiv e) \wedge (\alpha[x \leftarrow T] \vee \alpha[x \leftarrow F]). \quad (2)$$

In the sequel we shall often deal with assignment commands, $x := e$, where e is T or F. In this case, $sp(x := e,\alpha)$ can be reduced to

$$\begin{cases} x \wedge (\alpha[x \leftarrow T] \vee \alpha[x \leftarrow F]) & \text{if } e \text{ is } T \\ \neg x \wedge (\alpha[x \leftarrow T] \vee \alpha[x \leftarrow F]) & \text{if } e \text{ is } F \end{cases} \quad (3)$$

3. **The *sequential composition* command.** This command is of the form $S_1; S_2$, where S_1 and S_2 are any commands. Its semantics is given by

$$wlp(S_1; S_2, \alpha) = wlp(S_1, wlp(S_2, \alpha))$$
$$sp(S_1; S_2, \alpha) = sp(S_2, sp(S_1, \alpha)).$$

4. **The *alternative* command.** This command is of the form

$$\textbf{if}\ \ B_1 \rightarrow S_1\ \ [\!]\ \cdots\ [\!]\ \ B_n \rightarrow S_n\ \ \textbf{fi} \quad (4)$$

where $B_1, \ldots, B_n$ are Boolean expressions and $S_1, \ldots, S_n$ are any commands. In the sequel, we refer to (4) as IF. The command is executed as follows. If none of B_i is true, then the execution aborts. Otherwise, one command S_i with true B_i is *randomly* selected and executed.[7] The semantics of IF is given by

$$wlp(\text{IF},\alpha) = \bigwedge_{i=1}^{n}(B_i \supset wlp(S_i,\alpha))$$
$$sp(\text{IF},\alpha) = \bigvee_{i=1}^{n}(sp(S_i, B_i \wedge \alpha)).$$

[3] Note that quantifiers can be applied to Boolean variables only.

[4] Note that quantified formulae can be reduced into equivalent Boolean expressions. Namely, formulae of the form $\forall x.\alpha$ can be replaced by $\alpha[x \leftarrow T] \wedge \alpha[x \leftarrow F]$, whereas formulae of the form $\exists x.\alpha$ by $\alpha[x \leftarrow T] \vee \alpha[x \leftarrow F]$. Accordingly, assertion languages we consider here are in fact propositional.

[5] The original Dijkstra's language contains *abort* command and *iterative* commands as well, but they are not needed for our purpose.

[6] In what follows, we do not specify the *wp* formula tranformers for the considered language, because they will not be needed in the sequel.

[7] Note that when more than one of B_i is true, the selection of a command to execute is nondeterministic.

2.2 Main results

Dijkstra and Scholten [1990] consider various classes of computations. The class we are primarily interested in here is called by these authors *initially α and finally β under control of S*, where α and β are formulae and S is a command. This class, which will be denoted by $[S]^{\alpha}_{\beta}$, represents the set of all computations under control of S that start in a state satisfying α and terminate in a state satisfying β.

Suppose that $c \in [S]^{\alpha}_{\beta}$. Obviously, since S terminates in a state satisfying β and $wlp(S, \neg\beta)$ represents the set of all states such that S begun in any of them either terminates in a state satisfying $\neg\beta$ or loops forever, it must be the case that the initial state of c satisfies $\alpha \wedge \neg wlp(S, \neg\beta)$. Similarly, since S starts in a state satisfying α and $sp(S, \alpha)$ represents the set of all states such that any of them can be reached by starting execution of S in some state satisfying α, we conclude that the final state of c satisfies $\beta \wedge sp(S, \alpha)$. An interesting question is whether the formulae $\alpha \wedge \neg wlp(S, \neg\beta)$ and $\beta \wedge sp(S, \alpha)$ provide a complete description of the initial and final states of the computations from the class $[S]^{\alpha}_{\beta}$. That the answer is positive follows from the following result which can be found in [Dijkstra and Scholten, 1990].

Theorem 1 The formula $\alpha \wedge \neg wlp(S, \neg\beta)$ (resp. $\beta \wedge sp(S, \alpha)$) holds in a state σ iff there exists a computation c from $[S]^{\alpha}_{\beta}$ such that σ is the initial (resp. final) state of c. ■

Consider now the class of computations $\mathcal{C} = [S]^{\alpha}_{\beta}$, where S is the sequence $S_1; S_2; \ldots; S_n$. Let $ST_{\mathcal{C}}(i)$, $0 \leq i \leq n$, be the set of all states satisfying the following condition: for each $\sigma \in ST_{\mathcal{C}}(i)$, there exists a computation $c \in \mathcal{C}$ such that σ is reached by c after executing $S_1; \ldots; S_i$. In what follows, the members of $ST_{\mathcal{C}}(i)$ will be referred to as *i-states* of $\mathcal{C}$. Clearly, 0-states are initial states and n-states are final states. The following theorem provides a complete characterization of i-states.

Theorem 2 The formula $sp(S_1; \ldots; S_i, \alpha) \wedge \neg wlp(S_{i+1}; \ldots; S_n, \neg\beta)$ holds in a state σ iff there exists a computation from the class $\mathcal{C} = [S_1; S_2; \ldots; S_n]^{\alpha}_{\beta}$ such that $\sigma \in ST_{\mathcal{C}}(i)$, $0 \leq i \leq n$.[8] ■

There is another class of computations that we shall be interested in. This class, denoted by $[S_1; \ldots; S_n]^{\alpha}_{\beta}(\gamma)$, represents the set of all computations under control of $S_1; \ldots; S_n$ that start in a state satisfying α, terminate in a state satisfying β and, in addition, any state of any computation from this class that can be reached after executing $S_1; \ldots; S_i$ $(0 \leq i \leq n)$ satisfies γ. The class $[S_1; \ldots; S_n]^{\alpha}_{\beta}(\gamma)$ will be referred to as *initially α, finally β and always γ under control of $S_1; \ldots; S_n$*.[9]

The next theorem provides a complete characterization of i-states of the introduced class of computations.

[8] We assume here that $sp(S_1; \ldots; S_i, \alpha)$ is α if $i = 0$ and $\neg wlp(S_{i+1}; \ldots; S_n, \neg\beta)$ is β if $i = n$.

[9] Note that if γ is T, then the class $[S_1; \ldots; S_n]^{\alpha}_{\beta}(\gamma)$ reduces to the class $[S_1; \ldots; S_n]^{\alpha}_{\beta}$.

Theorem 3 The formula $sp^{\gamma}(S_1; \ldots; S_i, \alpha \wedge \gamma) \wedge \neg wlp^{\gamma}(S_{i+1}; \ldots; S_n, \neg(\beta \wedge \gamma))$ holds in a state σ iff there exists a computation from the class $\mathcal{C} = [S_1; \ldots; S_n]^{\alpha}_{\beta}(\gamma)$ such that $\sigma \in ST_{\mathcal{C}}(i)$. Here sp^{γ} and wlp^{γ} are specified by the following recursive definitions (δ is any formula).

$$sp^{\gamma}(S_1, \delta) = \gamma \wedge sp(S_1, \delta) \quad (5)$$

$$sp^{\gamma}(S_1; \ldots; S_i, \delta) = \gamma \wedge sp(S_i, sp^{\gamma}(S_1; \ldots; S_{i-1}, \delta)) \quad (6)$$

$$wlp^{\gamma}(S_n, \delta) = \neg\gamma \vee wlp(S_n, \delta) \quad (7)$$

$$wlp^{\gamma}(S_{i+1}; \ldots; S_n, \delta) = \neg\gamma \vee wlp(S_{i+1}, wlp^{\gamma}(S_{i+2}; \ldots; S_n, \delta)). \quad (8)$$

3 Action languages and action scenarios

To define an action language one proceeds in three steps.

(1) First, we choose an assertion language to represent the effect of actions. The "shooting language" we use in the sequel uses two Boolean variables: *a* and *l*, standing for *alive* and *loaded,* respectively. To be in accord with the AI terminology, these variables will be referred to as *fluents.*

(2) The next step is to provide action symbols representing the actions under consideration. In the shooting language we have four such symbols: *load* (a gun), *wait, spin* (a chamber) and *shoot* (a turkey). The intention is that *load* makes the gun loaded, *wait* does not cause any changes in the world, the effect of *spin* is that randomly the gun is loaded or not after the action, regardless of whether it was loaded before or not, and *shoot* makes the gun unloaded and the turkey dead, provided that the gun was loaded before.

(3) The final step is to define Dijkstra-style semantics for the choosen actions. To perform this step for the shooting language, it suffices to note that the considered actions can be easily translated into the programming language specified in the previous section. More specifically, *load* corresponds to the assignment command $l := T$, *wait* is just the *skip* command, whereas *spin* and *shoot* are translated into alternative commands **if** $l \rightarrow a := F; l := F \;[\!]\; \neg l \rightarrow skip$ **fi** and **if** $T \rightarrow l := T \;[\!]\; T \rightarrow l := F$ **fi**, respectively.[10] Given the above translations, the chosen actions can be provided with Dijkstra-style semantics. Performing routine calculations one easily obtains:

- $wlp(load, \alpha) = \alpha[l \leftarrow T]$;
- $sp(load, \alpha) = l \wedge (\alpha[l \leftarrow T] \vee \alpha[l \leftarrow F])$;
- $wlp(wait, \alpha) = sp(wait, \alpha) = \alpha$;
- $wlp(spin, \alpha) = \alpha[l \leftarrow T] \wedge \alpha[l \leftarrow F]$;
- $sp(spin, \alpha) = \alpha[l \leftarrow T] \vee \alpha[l \leftarrow F]$;
- $wlp(shoot, \alpha) = (l \supset \alpha[a \leftarrow F, l \leftarrow F]) \wedge (\neg l \supset \alpha)$;
- $sp(shoot, \alpha) = (\neg a \wedge \neg l \wedge \alpha[a \leftarrow T, l \leftarrow T]) \vee (\neg a \wedge \neg l \wedge \alpha[a \leftarrow F, l \leftarrow T]) \vee (\neg l \wedge \alpha)$.

[10] Obviously, not every action can be translated into Dijkstra's programming language. However, most of the actions that can be found in the AI literature enjoy this property.

The objects we shall be primarily interested in are *action scenarios*. These are expressions of the form

$$[\alpha]\ A_1; \ldots; A_n\ [\beta] \tag{9}$$

where α and β are formulae and $A_1, \ldots, A_n$ are actions. The scenario has the following intuitive interpretation: α was observed to hold in the initial state, then the actions $A_1, \ldots, A_n$ were sequentially performed, and then β was observed to hold in the final state. Formally, the scenario (9) is to be viewed as representing the class of computations initially α and finally β under control of $A_1; \ldots; A_n$.

Example 1 (Yale Shooting Scenario) Below is Hank and McDermot's Yale Shooting Scenario [1987].

$$[a \wedge \neg l]load; wait; shoot[T]\ \blacksquare$$

4 Reasoning about scenarios

We shall be interested in the following reasoning task: "Given a scenario $[\alpha]\ A_1; \ldots; A_n\ [\beta]$, a formula γ and an integer k such that $0 \leq k \leq n$, determine whether γ is assured to hold after performing the actions $A_1, \ldots, A_k$". Viewing a scenario as representing a class of computations, the above reasoning task can be stated more formally: "Given a scenario $[\alpha]\ A_1; \ldots; A_n\ [\beta]$, a formula γ and an integer k such that $0 \leq k \leq n$, determine whether γ is true in all k-states of the class of computations $[A_1; \ldots; A_n]^{\alpha}_{\beta}$.

In what follows, the symbol $\vdash$ stands for the provability relation of classical propositional logic.

Let $SC = [\alpha]\ A_1; \ldots; A_n\ [\beta]$ be a scenario and let $0 \leq k \leq n$. The *description of* the k^{th}*-state of* SC, written $DS_k(SC)$, is the formula given by

$$\begin{cases} \alpha \wedge \neg wlp(A_1; \ldots; A_n, \neg\beta) & \text{if } k = 0 \\ \beta \wedge sp(A_1; \ldots; A_n, \alpha) & \text{if } k = n \\ sp(A_1; \ldots; A_k, \alpha) \wedge & \\ \quad \neg wlp(A_{k+1}; \ldots; A_n, \neg\beta) & \text{otherwise} \end{cases}$$

The next result follows immediately from Theorem 2.

Theorem 4 Let $SC = [\alpha]\ A_1; \ldots; A_n\ [\beta]$ be a scenario. A formula γ is assured to hold after performing the actions $A_1; \ldots; A_k$ $(0 \leq k \leq n)$ iff $DS_k(SC) \vdash \gamma$. ■

Remark 1 All actions considered in this paper are guaranteed to terminate. If $A_1; \ldots; A_k$ is any sequence of such actions, then $wlp(A_1; \ldots; A_k, F) = F$. Accordingly, if SC is a scenario of the form $[\alpha]\ A_1; \ldots; A_n\ [T]$, then $DS_0(SC) = \alpha$ and, for $0 < k < n$, $DS_k(SC) = sp(A_1; \ldots; A_k, \alpha)$. We shall often make use of this fact in the sequel.

5 Examples

Example 1 (continued) The Yale Shooting Scenario (YSS, for short) is an example of the temporal prediction. The intended conclusion is that after performing the actions the turkey is dead and the gun is unloaded. We calculate $DS_3(YSS)$.[11]

$$\begin{aligned} DS_3(YSS) &= sp(load; wait; shoot, a \wedge \neg l) \\ &= sp(shoot, sp(wait, sp(load, a \wedge \neg l))) \\ &= sp(shoot, sp(wait, l \wedge ((a \wedge T) \vee (a \wedge F)))) \\ &\equiv sp(shoot, sp(wait, l \wedge a)) \\ &= sp(shoot, l \wedge a) \\ &= (\neg a \wedge \neg l \wedge T \wedge T) \vee (\neg a \wedge \neg l \wedge T \wedge F) \vee (\neg l \wedge l \wedge a) \\ &\equiv \neg a \wedge \neg l. \end{aligned}$$

Since $DS_3(YSS) \vdash \neg a \wedge \neg l$, we conclude that in the final state the turkey is dead and the gun is unloaded. ■

Example 2 (Russian Turkey Scenario) The Russian Turkey Scenario (RTS, for short) is an example of the temporal prediction, where actions with random effects are allowed.[12] The world of the scenario is the same as for the Yale Shooting Scenario, but *wait* is replaced by *spin*. The scenario is given by $[a \wedge \neg l]\ load; spin; shoot\ [T]$. The intended conclusion is that nothing can be said whether the turkey is alive or not in the final state. We calculate

$$\begin{aligned} DS_3(RTS) &= sp(load; spin; shoot, a \wedge \neg l) \\ &= sp(shoot, sp(spin, sp(load, a \wedge \neg l))) \\ &= sp(shoot, sp(spin, l \wedge ((a \wedge T) \vee (a \wedge F)))) \\ &\equiv sp(shoot, sp(spin, l \wedge a)) \\ &= sp(shoot, (l \wedge a)[l \leftarrow T] \vee (l \wedge a)[l \leftarrow F]) \\ &\equiv sp(shoot, a) \\ &= (\neg a \wedge \neg l \wedge T) \vee (\neg a \wedge \neg l \wedge F) \vee (\neg l \wedge a) \\ &\equiv \neg l. \end{aligned}$$

Since $DS_3(RTS) \not\vdash a$ and $DS_3(RTS) \not\vdash \neg a$, no conclusion can be derived with respect to whether a or $\neg a$ holds in the final state. ■

Example 3 (Stanford Murder Mystery) Consider the following problem (SMM).[13] The turkey is alive in the initial state, and after the actions *shoot* and *wait* are successively performed, it is dead. The story can be represented by the following scenario: $[a]\ shoot; wait\ [\neg a]$. The question we are interested in is when the turkey died and whether the gun was originally loaded (the temporal postdiction). The intended conclusion is that the gun was loaded in the initial state and the turkey died during the shooting. We calculate

$$\begin{aligned} DS_0(SMM) &= a \wedge \neg wlp(shoot; wait, a) \\ &= a \wedge \neg wlp(shoot, wlp(wait, a)) \\ &= a \wedge \neg wlp(shoot, a) \\ &= a \wedge \neg((l \supset F) \wedge (\neg l \supset a)) \\ &\equiv a \wedge l. \end{aligned}$$

Since $DS_0(SMM) \vdash l$, we immediately conclude that the gun was loaded in the initial state.

Now, we calculate $DS_1(SMM)$.

$$\begin{aligned} DS_1(SMM) &= sp(shoot, a) \wedge \neg wlp(wait, a) \\ &= ((\neg a \wedge \neg l \wedge T) \vee (\neg a \wedge \neg l \wedge F) \vee (\neg l \wedge a)) \wedge \neg a \\ &\equiv \neg a \wedge \neg l. \end{aligned}$$

Since $DS_1(SMM) \vdash \neg a$, we infer that the turkey was dead after performing the action *shoot*. ■

[11] Note the use of the symbols "=" and "≡" during the calculation. We write $X = Y$ if Y is obtained from X by employing the semantics of *wlp* or *sp*, whereas $X \equiv Y$ indicates that X and Y are logically equivalent.

[12] This example is from [Sandewall, 1994].

[13] This example is from [Baker, 1991].

6 Ramification problem

The *ramification problem* concerns efficient representation of the indirect effects of actions. In this paper we limit ourselves to the simplest class of ramifications, namely those introduced by *domain constraint axioms.*

The domain constraint axioms describe general facts that are assumed to hold in any state of the dynamically changing world under consideration. If α is such an axiom, then some fluents occurring in α may change their values even if these fluents are not included in the action's description. Consider, for instance, the Yale Shooting Scenario, augmented with the domain constraint axiom $a \equiv \neg d$, where *d* stands for *dead.* Given this axiom, the action *shoot* makes the turkey not only not alive, but also dead (provided, of course, that the gun is loaded).

Clearly, when the domain constraint axioms are involved, a scenario $[\alpha]\ A_1; \ldots; A_n\ [\beta]$ should not be regarded as representing the class of computations *initially a and finally B under control of* $A^1 \ldots; A_n$, but rather as the class *initially a, finally (3 and always* γ *under control of* $A_1; \ldots; A_n$, where γ is the conjunction of all domain constraint axioms associated with the scenario. Unfortunately, changing the class of computations only is not always sufficient to properly deal with the domain constraint axioms. To see why, consider an illustrating example.

Example 4 Consider the shooting language supplied with a new fluent w, standing for *walking.* Let *DC* be a domain constraint axiom given by $w \supset a$.. Let *WSS* be the scenario $[w \wedge l]\ shoot\ [T]$.

The intended conclusion is that the turkey is not alive and not walking in the final state. We calculate:

$$\begin{aligned} DS_1(WSS) &= sp^{DC}(shoot, w \wedge l \wedge DC) \\ &= sp(shoot, w \wedge l \wedge (w \supset a)) \wedge (w \supset a) \\ &= ((\neg a \wedge \neg l \wedge w \wedge T \wedge (w \supset T)) \vee (\neg a \wedge \neg l \wedge w \wedge T \\ &\wedge (w \supset F)) \vee (\neg l \wedge w \wedge l \wedge (w \supset a))) \wedge (w \supset a) \\ &\equiv ((\neg a \wedge \neg l \wedge w) \vee (\neg a \wedge \neg l \wedge w \wedge \neg w)) \wedge (w \supset a) \\ &\equiv (\neg a \wedge \neg l \wedge w) \wedge (w \supset a) \\ &\equiv F \end{aligned}$$

The description of the final state is inconsistent. ■

The inconsistency we have arrived at is due to the fact that the fluent *w* obeys the law of inertia. In our approach, all fluents that are not explicitly affected by an action retain their values when the action is performed. While this is a nice property from the standpoint of the frame problem, it leads to some difficulties when actions with indirect effects are involved.

To deal with domain constraint axioms, we need a mechanism which, for a given action *A,* releases chosen fluents from obeying the law of inertia when the action *A* is executed. Fortunately, the release mechanism can be easily implemented in our formalism. Suppose that $f_1, \ldots, f_n$ are fluents which are to be released during executing an action *A.* To achieve the desired effect, *A* should be replaced by *A; releaee*(f_1)*;...; release*(f_n) where *release(fi)* is the command

$$\textbf{if } T \rightarrow f_i := T \ [\!]\ T \rightarrow f_i := F \textbf{ fi}.$$[14]

[14]The *release* pseudo-command corresponds closely to Sandewall's [1994] *occlusion* device. See also [Kartha and Lifschitz, 1994].

Example 4 (new solution) To properly deal with *WSS* scenario, the fluent *w* should be released when the action *shoot* is performed. Thus, we replace *shoot* by *shoot** given by $shoot; release(w)$. The description of the initial state is $w \wedge l \wedge DC$. We calculate:

$$\begin{aligned} DS_1(WSS) \wedge DC &= sp(shoot^*, (w \wedge l) \wedge DC) \wedge DC \\ &= sp(shoot; release(w), (w \wedge l) \wedge DC) \wedge DC \\ &\equiv sp(release(w), \neg a \wedge \neg l \wedge w) \wedge (w \supset a) \\ &\equiv \neg a \wedge \neg l \wedge (w \supset a). \end{aligned}$$

Since $DS_1(YSS) \wedge DC \vdash \neg a \wedge \neg w$, we conclude that the turkey is not alive and not walking in the final state. ■

We now provide a general method to reason about action scenarios with domain constraint axioms. We assume here that the number of these axioms is finite, so that they can be always regarded as a single formula.

Let $SC = [\alpha]\ A_1; \ldots; A_n\ [\beta]$ be an action scenario and suppose that DC is the conjunction of all domain constraint axioms assiociated with SC. Assume further that the fluents $f_i^1, \ldots, f_i^{m_i}$ are to be released from the law of inertia when the action A_i is executed. We write A^* to denote the sequential composition $A_i; release(f_i^1); \ldots; release(f_i^{m_i})$. The *description of* the k^{th}*-state of* SC with respect to DC, written $DS_k^{DC}(SC)$, is the formula given by

$$\begin{cases} \alpha \wedge \neg wlp^{DC}(A_1^*; \ldots; A_n^*, \neg(\beta \wedge DC)) & \text{if } k = 0 \\ sp^{DC}(A_1^*; \ldots; A_k^*, \alpha) \wedge & \\ \neg wlp^{DC}(A_{k+1}^*; \ldots; A_n^*, \neg(\beta \wedge DC)) & \text{if } 0 < k < n \\ \beta \wedge sp^{DC}(A_1^*; \ldots; A_n^*, \alpha) & \text{if } k = n \end{cases}$$

where sp^{DC} and wlp^{DC} are defined by the equations (5) - (8), with γ replaced by DC and S_i $(0 < i \leq n)$ replaced by A_i^*.

Theorem 3 immediately implies:

Theorem 5 Let $SC = [\alpha]\ A_1; \ldots; A_n\ [\beta]$ be an action scenario, DC be the conjunction of domain constrain axioms and suppose that A_i^* $(0 \leq i \leq n)$ is specified as before. A formula γ is assured to hold after performing the actions $A_1; \ldots; A_k$ if $DS_k^{DC}(SC) \vdash \gamma$. ■

Example 5 Consider a variant of the Yale Shooting Scenario. There are four fluents: a, l, d and u, standing for *alive*, *loaded*, *damaged* (gun) and *usable* (gun), respectively. The actions are: *load*, and *shoot*. Finally, we have a domain constraint axiom DC given by $l \wedge \neg d \equiv u$. The meaning of the actions is as before with one proviso: to successfully perform the action *shoot*, the gun must be usable. The translation of the new version of *shoot* is **if** $u \rightarrow a := F; l := F$ [] $\neg u \rightarrow skip$ **fi** and its semantics is given by

- $wlp(shoot, \alpha) = (u \supset \alpha[a \leftarrow F, l \leftarrow F]) \wedge (\neg u \supset \alpha)$
- $sp(shoot, \alpha) = sp(shoot, \alpha) = \neg a \wedge \neg l \wedge \exists a, l.(u \wedge \alpha) \vee \neg u \wedge \alpha$.

Because the fluent u is indirectly affected by *load* and *shoot*, it must be released when any of these actions is performed.

Let $VYSS$ be the scenario $[\neg d]\ load; shoot\ [T]$. The intended conclusion is that the turkey is not alive and the gun is unloaded, unusable and not damaged in the final state.

Since the fluent u is released during performing the actions *load* and *shoot*, we define $load^* = load; release(u)$ and $shoot^* = shoot; release(u)$. Performing straightforward calculations, one gets

$$DS_2^{DC}(VYSS) \equiv \neg a \wedge \neg l \wedge \neg u \wedge \neg d. \qquad (10)$$

In view of (10), we immediately conclude that the turkey is not alive and the gun is unloaded, unusable and not damaged in the final state. ■

7 Conclusions

We have applied Dijkstra's semantics for programming languages to formalization of reasoning about action and change. We believe that the results reported here are interesting and worth of further investigation. The presented approach can be employed to represent a broad class of action scenarios, including those where actions with random and indirect effects are permitted. In addition, both temporal prediction and postdiction tasks can be properly dealt with, without requiring initial or final situations to be completely specified. The major advantage of our proposal is that it is very simple and more effective than many other approaches directed at formalizing reasoning about action and change.

As we remarked earlier, recent work of Sandewall [1994] provides a very general framework to study logics of action and change. Obviously, the question of how our proposal fits in this framework should be investigated and will be pursued in the future. It is also interesting to compare our approch with *ARo* language introduced recently by Kartha and Lifschitz [1994].

The task of. implementation is another point of interest. Calculating *DSk(SC),* for a given scenario *SC,* amounts to simple syntactic manipulations on formulae and can be performed very efficiently. The only computational problem is to determine whether a given formula can be derived from the description of the state under consideration. This task can be realized by a theorem prover appropriate for the logic in which the effects of actions are described.

Acknowledgements

We would like to thank Wladysiaw M. Turski, Wiodek Drabent and Andrzej Szalas for their comments on the previous draft of this paper.

References

[Baker, 1991] A. B. Baker. Nonmonotonic reasoning in the framework of the situation calculus. *Artificial Intelligence,* 49(5):5-23, 1991.

[Dijkstra, 1976] E. W. Dijkstra. *A Discipline of Programming,* Prentice Hall, 1976.

[Dijkstra and Scholten, 1990] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics,* Springer-Verlag, 1990.

[Fikes and Nilsson, 1971] R. E. Fikes and R. E. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving". *Artificial Intelligence,* 2(3-4): 189-208, 1971.

[Gelfond *et al.,* 199l] M. Gelfond, V. Lifschitz and A. Rabinov. What are the limitations of situation calculus? In *Proc. AAAI Symposium of Logical Formalization of Commonsense Reasoning,* Stanford, 55-69, 1991.

[Hanks and McDermott, 1987] S. Hanks and D. McDermott. Nonmonotonic Logic and Temporal Projection. *Artificial Intelligence,* 33(3):379-412, 1987.

[Hayes and McCarthy, 1969] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence* 4. B. Meltzer and D. Michie (eds.), 1969, 463-502.

[Kartha and Lifschitz, 1994] G. N. Kartha and V. Lifschitz. Actions with indirect effects (preliminary report). In *Proc. KR-94,* Bonn, Germany, Morgan Kaufmann Publishers, San Francisco, 341-350, 1994.

[Lifschitz, 1988] V. Lifschitz. Formal theories of action. In *Readings in Nonmonotonic Reasoning,* M. Ginsberg (ed.), Morgan Kaufmann Publishers, Palo Alto, 35-57, 1988.

[Lifschitz and Rabinov, 1989] V. Lifschitz and A. Rabinov. Miracles in formal theories of action. *Artificial Intelligence,* 38(6):225-237, 1989.

[Lukaszewicz and Madaliriska-Bugaj, 1994] W. Lukaszewicz and E. Madaliriska-Bugaj. Program verification techniques as a tool for reasoning about action and change. In *Proc. of German AI Conference (KI-94)*- Lecture Notes in Artificial Intelligence, 861, Springer-Verlag, 226-236, 1994.

[Lukaszewicz and Madaliriska-Bugaj, 1995] W. Lukaszewicz and E. Madaliriska-Bugaj. Reasoning about action and change: actions with abnormal effects. Submitted to German AI Conference, KI-95.

[Pednault, 1986] E. P.D. Pednault. Toward a mathematical theory of plan synthesis. Ph. D. Thesis, Dept. of Electrical Engineering , Stanford University, Stanford, 1986.

[Pednault, 1988] E. P. D. Pednault. Synthesizing plans that contain actions with contex-dependent effects. *Computational Intelligence,* 4:356-372, 1988.

[Pednault, 1989] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. KR-89,* 324-333, 1989.

[Sandewall, 1994] E, Sandewall. *Features and Fluents: The Representation of Knowledge about Dynamical Systems,* Oxford Science Publications, Clarendon Press, Oxford, 1994.

[Waldinger, 1977] R. Waldinger. Achieving several goals simultaneusly. In *Machine Intelligence* 8. E. Ellock and D. Michie (eds.), Ellis Horwood, Edinburgh, 94-136.