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

Witold Łukaszewicz Ewa Madalińska-Bugaj

Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warsaw, POLAND
emails: 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 [2, 3] 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.[1] 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 α, returns a formula representing the set of all states that can be achieved by starting execution of A in some state satisfying α. The latter, when applied to a formula α, returns a formula providing a description of all states such that whenever execution of

*This is a revised and extended version of the paper [10]. The authors were supported by the ESPRIT Basic Research Action No. 6156 - DRUMS II.

[1]Formula transformers are often referred to as predicate transformers in the Computer Science literature.

A starts in any one of them and terminates, the output state satisfies α. [2]

The idea of employing formula transformers to specify effects of actions is not new in the AI literature and goes back to STRIPS system [4]. Waldinger [18] introduces a notion of a *regression operator* which corresponds closely to the weakest precondition transformer. Pednault [13, 14, 15] employes regression operators in the context of plan synthesis. In [13] 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 [12, 8, 9, 5, 1] or Features and Fluents [16, 17]. 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.

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.

2 Introduction to Dijkstra's semantics

In [3] we are provided with a simple programming language whose semantics is specified in terms of formula trasformers. More specifically, with each command S there are associated three formula 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

[2] The reader familiar with Dijkstra's approach may wonder why 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. However, since all actions we consider in this paper terminate, the weakest precondition and the weakest liberal precondition coincide for them.

transformers, we have to make some remarks and introduce some terminology.

First of all, 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 as a matter of fact 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 the members of V into the truth-values $\{0,1\}$.

An *assertion language* over a set V of Boolean variables, denoted by $\mathcal{L}(\mathcal{V})$, is the set of all formulae constructable in the usual way from members of V, 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* $\alpha \in \mathcal{L}(\mathcal{V})$ *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.

It follows from the clauses (i)–(ii) above that quantified formulae can be reduced into equivalent Boolean expressions. (Note that formulae of the form $\forall x.\alpha$ can be always replaced by $\alpha[x \leftarrow T] \wedge \alpha[x \leftarrow F]$, whereas formulae of the form $\exists x.\alpha$ by $\alpha[x \leftarrow T] \vee \alpha[x \leftarrow F]$.)[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:

- $wp.S(\alpha)$ is the formula whose models are precisely all states such that execution of S begun in any one of them is guaranteed to terminate in a state satisfying α.

- $wlp.S(\alpha)$ is the formula whose models are precisely all states such that whenever execution of S starts in any one of them and terminates, the output state satisfies α.

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

[4] It follows, therefore, that assertion languages we consider here are in fact propositional.

- $sp.S(\alpha)$ is the formula whose models are precisely all states such that each of them can be reached by starting execution of S in some state satisfying α.

For example, let S be the assignment command $x := T$, and consider the formula $x \vee y$, where x and y are Boolean variables of the programming language under consideration.[5] Clearly, $wp.S(x \vee y) = T$, for if σ is any state, then execution of S begun in σ terminates in a state satisfying $x \vee y$. Since the command $x := T$ always terminates, $wlp.S(x \vee y) = wp.S(x \vee y)$. On the other hand, $sp.S(x \vee y) = x$ because the states satisfying x are precisely those that can be reached by starting execution of S in some state satisfying $x \vee y$.

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[6]. Semantics of these commands is specified as follows.[7]

1. **The *skip* command.** This is the "empty" command in that its execution does not change the computation state. The semantics of *skip* is thus 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 effect of the command is to replace the value of x by the value of e. 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 a bit 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 the variable x does not occur in the expression e, the equation (1) can be simplified. In this case

   $$sp(x := e, \alpha) = (x \equiv e) \wedge \exists x.\alpha$$

[5] In this paper the letters T and F denote the propositional constants $True$ and $False$, respectively.

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

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

or equivalently

$$sp(x := e, \alpha) = (x \equiv e) \wedge (\alpha[x \leftarrow T] \vee \alpha[x \leftarrow F]). \tag{2}$$

In the sequel we shall often deal with assignment commands, $x := e$, where e is T or F. In this case the equation (2) can be replaced by

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

Reconsider the example given earlier, where α is $x \vee y$ and the assignment command is $x := T$. Applying (3), we get $sp(x := T, x \vee y) = x \wedge (T \vee y \vee F \vee y)$ which reduces to x.

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

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

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

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

where $B_1, \ldots, B_n$ are Boolean expressions and $S_1, \ldots, S_n$ are any commands. $B_1, \ldots, B_n$ are called *guards* and expressions of the form $B_i \rightarrow S_i$ are called *guarded commands*. In the sequel, we refer to the general command (4) as IF. The command is executed as follows. If none of the guards is true, then the execution aborts. Otherwise, one guarded command $B_i \rightarrow S_i$ with true B_i is *randomly* selected and S_i is executed.[8] The semantics of IF is given by

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

2.2 Main results

Dijkstra and Scholten [3] 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

[8] Note that when more than one guard is true, the selection of a guarded command is nondeterministic.

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 [3].

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 $\sigma \in ST_{\mathcal{C}}(i)$, $0 \leq i \leq n$.[9]

Proof If $i = 0$ or $i = n$, the proof immediately follows from Theorem 1. Assume therefore that $0 < i < n$.

($\Rightarrow$) Assume that 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. We have to show that there is a computation c from the class $[S_1; S_2; \ldots; S_n]^{\alpha}_{\beta}$ such that $\sigma \in ST_{\mathcal{C}}(i)$. Since σ satisfies $sp(S_1; \ldots; S_i, \alpha)$, σ is the final state of some computation c', under control of $S_1; \ldots; S_i$, which starts in some state satisfying α. Similarly, since σ satisfies $\neg wlp(S_{i+1}; \ldots; \neg\beta)$, σ is the initial state of some computation c'', under control of $S_{i+1}; \ldots; S_n$, which terminates in some state satisfying β. Taking c to be the composition of c' and c'', it is immediately seen that c is the member of the class $\mathcal{C} = [S_1; S_2; \ldots; S_n]^{\alpha}_{\beta}$ and $\sigma \in ST_{\mathcal{C}}(i)$.

($\Leftarrow$) Assume now that $\sigma \in ST_{\mathcal{C}}(i)$, for some computation c from the class $[S_1; S_2; \ldots; S_n]^{\alpha}_{\beta}$. We have to show that σ satisfies the formula $sp(S_1; \ldots; S_i, \alpha) \wedge$

[9] 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 $i = n$.

$\neg wlp(S_{i+1}; \ldots; S_n, \neg\beta)$. Since the initial state of c satisfies α and $sp(S_1; \ldots; S_i, \alpha)$ is true in any state that can be reached from some state satisfying α (by performing $S_1; \ldots; S_i$), it must be the case that $sp(S_1; \ldots; S_i, \alpha)$ is true in σ. Similarly, since the final state of c satisfies β and $wlp(S_{i+1}; \ldots; S_n, \neg\beta)$ holds in all states such that execution of $S_{i+1}; \ldots; S_n$ begun in any one of them either terminates in a state satisfying $\neg\beta$ or loops forever, it must be the case that $\neg wlp(S_{i+1}; \ldots; S_n, \neg\beta)$ is true in σ. In conclusion, σ satisfies $sp(S_1; \ldots; S_i, \alpha) \wedge \neg wlp(S_{i+1}; \ldots; S_n, \neg\beta)$. ■

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.[10]

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

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).

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

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

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

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

Proof We start with a few observations whose proofs can be easily obtained by induction on i $(0 \leq i \leq n)$.

(1) If σ satisfies $sp^{\gamma}(S_1; \ldots; S_i, \alpha \wedge \gamma)$, then σ is the final state of some computation from the class $[S_1; \ldots; S_i]^{\alpha}_{T}(\gamma)$.

(2) If σ satisfies $\neg wlp^{\gamma}(S_{i+1}; \ldots; S_n, \neg(\beta \wedge \gamma))$, then σ is the initial state of some computation from the class $[S_{i+1}; \ldots; S_n]^{T}_{\beta}(\gamma)$.

(3) If $\sigma \in ST_{\mathcal{C}}(i)$ for some computation c from the class $\mathcal{C} = [S_1; \ldots; S_n]^{\alpha}_{\beta}(\gamma)$, then σ satisfies $sp^{\gamma}(S_1; \ldots; S_i, \alpha \wedge \gamma)$.

(4) If $\sigma \in ST_{\mathcal{C}}(i)$ for some computation c from the class $\mathcal{C} = [S_1; \ldots; S_n]^{\alpha}_{\beta}(\gamma)$, then σ satisfies $\neg wlp^{\gamma}(S_{i+1}; \ldots; S_n, \neg(\beta \wedge \gamma))$.

Given the observations (1)–(4), the proof of the theorem proceeds exactly like that of Theorem 2. ■

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

3 Action languages and action scenarios

In this section, we show how to specify action languages using Dijkstra's semantics, illustrate the procedure by defining a simple "shooting" language and introduce a notion of an action scenario.

To define an action language one proceeds in three steps.

(1) First, we choose an assertion language to represent the effect of actions. For the shooting language we take the language with 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

$$\begin{array}{lll} & spin & \\ \textbf{if} & & \\ & T \to\ l := T \quad [\!] & \\ & T \to\ l := F & \\ \textbf{fi} & & \end{array} \qquad \text{and} \qquad \begin{array}{lll} & shoot & \\ \textbf{if} & & \\ & l \to\ a := F; l := F \quad [\!] & \\ & \neg l \to\ skip & \\ \textbf{fi} & & \end{array}$$

respectively. Given the above translations, the chosen actions can be provided with Dijkstra-style semantics. Performing routine calculations one easily obtains:

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

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

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 the famous Yale Shooting Scenario [6].

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

4 Reasoning about scenarios

In this section, we provide a method to reason 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 $A_1, \ldots, A_k$".[11] 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

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

The next result follows immediately from Theorem 2.

[11] For $k = 0$, i.e. when we want to determine whether γ is true in the initial state, the above reasoning task is known as the *temporal postdiction problem*. For $k = n$, i.e. when we want to determine whether γ holds in the final state, the task is usually referred to as the *temporal prediction problem*.

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

In this section we consider a number of well-known examples to illustrate the method introduced in the previous section.

Example 1 (continued) The Yale Shooting Scenario (YSS, for short) is an example of the temporal prediction. We have a description of the initial state and we are interested what is guaranteed to hold in the final state. The intended conclusion is that after performing the actions the turkey is dead and the gun is unloaded.

We calculate $DS_3(YSS)$.[12]

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

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.[13] The world of the scenario is the same as for the Yale Shooting Scenario, but *wait* is replaced by *spin*. Recall that the effect of this latter action is that randomly the gun may or may not be loaded after its execution, regardless of whether it was loaded before or not. The scenario is given by

$$[a \wedge \neg l]\ load; spin; shoot\ [T].$$

[12] 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.

[13] This example is from [17].

The intended conclusion is that nothing can be said whether the turkey is alive or not in the final state.

We calculate $DS_3(RTS)$.

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

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, known in the AI literature as the Stanford Murder Mystery (SMM).[14] 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 $DS_0(SMM)$.

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

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

Now, we calculate $DS_1(SMM)$.

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

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

[14] This example is from [1].

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).

To deal with the domain constraint axioms in our approach, one can try to take them as a part of the description of the k^{th}-state of a given action scenario. While this sometimes works (for instance, for the Yale Shooting Scenario, supplied with $a \equiv \neg d$), this method is generally inappropriate. 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

$$[w \wedge l]\ \mathit{shoot}\ [T].$$

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

Calculating $DS_1(WSS)$, we get

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

As we see, the description of the final state we obtained 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.[15] 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

$$A; release(f_1); \ldots; release(f_n)$$

where

$$release(f_i) \;=\; \textbf{if}\;\; T \to f_i := T \;\;[\!]\;\; T \to f_i := F \;\;\textbf{fi} \tag{10}$$

The composition of $release(f_1); \ldots; release(f_n)$ may be written for short as $release(f_1, \ldots, f_n)$.

There is another point that should be emphasized when domain constraint axioms are involved. These axioms should be taken as a part of the description of both the initial and final state.

Example 4 (new solution) To properly deal with WSS scenario, the fluent w should be released when the action *shoot* is performed. Accordingly, we replace *shoot* by $shoot^*$ given by $shoot; release(w)$. The description of the initial state is $w \wedge l \wedge DC$. We calculate $DS_1(WSS) \wedge DC$:

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

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 systematize our observations by providing 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.

Definition 1 We say that an action A is atomic iff A is represented by a sequence $S_1; \ldots; S_n$, $n \geq 1$, of assignment commands.■

Definition 2 Let A be an action and the fluents $f_1, \ldots, f_m$ are to be released from the law of inertia when the action A is executed. By A^* we denote the following action:

1. if A is represented by *skip* then $A^* = A$.
2. if A is atomic then $A^* = A; release(f_1, \ldots, f_m)$.

[15] Compare [7].

3. if A is represented by alternative command of the form

$$\textbf{if} \quad B_1 \to S_1 \quad [\!] \quad \cdots \quad [\!] \quad B_n \to S_n \quad \textbf{fi}$$

where S_i, $i = 1, \ldots, n$, is a command representing an auxiliary action A^i, then A^* is

$$\textbf{if} \quad B_1 \to (A^1)^* \quad [\!] \quad \cdots \quad [\!] \quad B_n \to (A^n)^* \quad \textbf{fi}$$

■

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.

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 $DS_k^{DC}(SC) \vdash \gamma$. ■

The different but more detailed approach to ramification problem we present in [11].

Example 5 Consider a variant of the Yale Shooting Scenario. There are four fluents: a, l, b and u, standing for *alive*, *loaded*, *broken* (gun) and *usable* (gun), respectively. The actions are: *load*, and *shoot*. Finally, we have a domain constraint axiom DC given by $l \wedge \neg b \equiv u$. 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

$$\textbf{if} \quad u \to a, l := F, F \ [\!] \ \neg u \to skip \ \textbf{fi}$$

and its semantics is given by

- $wlp(shoot, \alpha) = (u \supset \alpha[a \to F, l \to F]) \wedge (\neg u \supset \alpha)$

- $sp(shoot, \alpha) = sp(shoot, \alpha) = \neg a \wedge \neg l \wedge \exists a, l.(u \wedge \alpha) \vee \neg u \wedge \alpha$.

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

$$[\neg b]\ load; shoot\ [T].$$

The intended conclusion is that the turkey is not alive and the gun is unloaded, unusable and unbroken in the final state.

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

$$DS_2^{DC}(VYSS) \equiv \neg a \wedge \neg l \wedge \neg u \wedge \neg b. \tag{11}$$

In view of (11), we immediately conclude that the turkey is not alive and the gun is unloaded, unusable and unbroken 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.

The actions considered in this paper are very simple in that they are represented by identifiers. In practice, it is often convenient to have action schemata rather, each representing a class of similar actions applicable to various tuples of individuals. For example, we may have the action schema $Put(x, y)$ with the intended meaning that a block x is to be put on the top of a block y. There is no technical difficulty to admit action schemata in our system, but this topic will be disscussed in a future paper.

As we remarked earlier, recent work of Sandewall [16, 17] 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 $\mathcal{AR}$ language introduced recently by Kartha and Lifschitz [7].

The task of implementation is another point of interest. Calculating $DS_k(SC)$, 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 initial situation. This task can be realized by a theorem prover appropriate for the logic in which the effects of actions are described. For the shooting language all we need is a theorem prover for classical propositional logic. In more complex applications, a theorem prover for first-order logic will be required. In any case, we work in the framework of classical monotonic logic which makes our approach simpler and computationally more efficient than the approaches employing non-monotonic forms of reasoning.

Acknowledgements We would like to thank Władysław M. Turski and Włodek Drabent for their comments on the previous draft of this paper.

References

[1] Baker, A. B., "Nonmonotonic Reasoning in the Framework of Situation Calculus", *Artificial Intelligence*, **49**, 1991, 5-23.

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

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

[4] Fikes, R. E., Nilsson N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", *Artifficial Intelligence*, **2**, 1971, 189-208.

[5] Gelfond, M,. Lifschitz, V., Rabinov, A., "What Are the Limitations of Situation Calculus?", *AAAI Symposium of Logical Formalization of Commonsense Reasoning*, Stanford, 1991, 55-69.

[6] Hanks, S., McDermott, D., "Nonmonotonic Logic and Temporal Projection", *Artificial Intelligence*, **33**, 1987, 379-412.

[7] Kartha, G. N., Lifschitz, V., "Actions with Indirect Effects (Preliminary Report)", in: *Proc. KR-94*, Bonn, Germany, Morgan Kaufmann Publishers, San Francisco, 1994, 341-350.

[8] Lifschitz, V., "Formal Theories of Action", in: *Readings in Nonmonotonic Reasoning*, M. Ginsberg (ed.), Morgan Kaufmann Publishers, Palo Alto, 1988, 35-57.

[9] Lifschitz, V., Rabinov, A., "Miracles in Formal Theories of Action", *Artificial Intelligence*, **38**, 225-237.

[10] Łukaszewicz, W., Madalińska-Bugaj, E.. Reasoning about Action and Change Using Dijkstra's Semantics for Programming Languages: Preliminary Report. *Proceedings of 14th International Joint Conference on Artificial Intelligence*, 1995, 1950-1955.

[11] Madalińska-Bugaj, E. How to Solve Qualification and Ramification Using Dijkstra's Semantics for Programming Languages. *AI*IA-97: Advances in Artificial Intelligence, Proceedings of 5th Congress of the Italian Association for Artificial Intelligence*, Springer-Verlag, Lecture Notes in Artificial Intelligence, **1321**, 381-392, 1997.

[12] McCarthy, J., Hayes, P.J., "Some Philosophical Problems from the Standpoint of Artificial Intelligence", in: B. Meltzer and D. Michie (eds.), *Machine Intelligence* **4**, 1969, 463-502.

[13] Pednault, E. P. D., "Toward a Mathematical Theory of Plan Synthesis", Ph. D. Thesis, Dept. of Electrical Engineering , Stanford University, Stanford.

[14] Pednault, E. P. D., "Synthesizing Plans that Contain Actions with Contex-Dependent Effects", *Computational Intelligence*, **4**, 1988, 356-372.

[15] Pednault, E. P. D., "ADL: Exploring the Middle ground between STRIPS and the Situation Calculus", in: R. Brachman, H. Levesque and R. Reiter (eds.), *Proc. KR-89*, 1989, 324-333.

[16] Sandewall, E., "The Range of Applicability of Nonmonotonic Logics for the Inertia Problem", in: *Proc. IJCAI-93*, 1993, 738-743.

[17] Sandewall, E., "Features and Fluents: A Systematic Approach to the Representation of Knowledge about Dynamical Systems", Research Report LITH-IDA-R-94-15, Dept. of Computer and Information Science, Linköping University, Sweden.

[18] Waldinger, R., "Achieving Several Goals Simultaneusly", in: *Machine Intelligence* **8**. E. Ellock and D. Michie (eds.), Ellis Horwood, Edinburgh, 94-136.