Reasoning about Action and Change: Actions with Abnormal Effects

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 present a logic-based method for reasoning about action and change. In contrast to most of the other approaches aimed at this kind of inference, our proposal admits actions with abnormal effects. More specifically, with each action A we associate a pair of specifications, $\mathcal{S}_1$ and $\mathcal{S}_2$, representing respectively normal and abnormal performance of A. The intention is that each action is performed according to its normal specification unless the contrary follows from observations.

To represent effects of actions, we use Dijkstra's formalism, originally developed for reasoning about programs. To distinguish between normal and abnormal performances of actions, Dijkstra's approach is combined here with Reiter's default logic.

1 Introduction

Most of the research devoted to reasoning about action and change has been based on the assumption that each action behaves in a fixed way. More specifically, to each action A there is assigned a unique specification $\mathcal{S}$ describing the effects of A in terms of a state in which A is performed.[1] For instance, the well-known action *shoot* is usually defined as making a gun unloaded and a turkey dead, provided that a gun was loaded. Accordingly, each time the action is executed in a state in which the gun is loaded, it is taken for granted that the turkey is made dead.

In this paper, we generalize the above assumption by admitting actions that may exhibit abnormal behaviour. More precisely, with each action A we associate a pair of specifications, $\mathcal{S}_1$ and $\mathcal{S}_2$, corresponding respectively to a normal and an abnormal behaviour of A. The intention, of course, is that A behaves according to $\mathcal{S}_1$ unless the contrary follows from observations. Reconsider the action *shoot*. We may define its normal behaviour as making a gun unloaded and a turkey dead, provided that a gun is loaded, and the abnormal one as making only the

*The author was supported by the ESPRIT Basic Research Action No. 6156 - DRUMS II and KBN grant 3 P406 01 06.

†The author was supported by MEDICIS CHRXCT 920054 Project.

[1]Depending on whether A is deterministic or not, $\mathcal{S}$ can be formally viewed as an unary function or a binary relation, specified on a space of all possible states.

gun unloaded. Now, if all we know is that the action was performed in a state in which the gun was loaded, we infer that the turkey was made dead. However, if we additionally observe that the turkey is alive after executing the action, we are forced to assume that the action behaved abnormally.

To formalize effects of actions, we use Dijkstra's approach, originally developed for reasoning about programs [2, 3]. The strength of Dijkstra's proposal is its efectiveness. It has been showed in our earlier papers [6, 7], where Dijkstra's methodology was employed to formalize conventional forms of inference about action and change. To distinguish between normal and abnormal behaviour of actions, Dijkstra's formalism is combined here with Reiter's default logic [9].[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 show how to define action languages using Dijkstra's methodology, illustrate this by specifying a simple action language, and introduce a notion of action scenario. Section 4 defines the kind of inference we shall be interested in, whereas section 5 provides a method realizing this type of reasoning. Section 6 contains a number of examples. Finally, in section 7 we provide conclude remarks and discuss the future work.

We assume that the reader is familiar with Reiter's default logic.

The proofs of all theorems will be given in the full version of the paper.

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 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 obtained from β by simultaneously replacing all free occurrences of $x_1, \ldots, x_n$ by $\alpha_1, \ldots, \alpha_n$, respectively.

[2] The problem of formalizing actions with abnormal effects has been recently addressed by Radzikowska [8], who works in the framework of Sandewall's Features and Fluents [10].

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

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.
- $sp.S(\alpha)$ 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]. Seman-

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

[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.

tics 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. Its semantics 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

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

If the variable x does not occur in the expression e, the equation (1) can be simplified. In this case

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

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

$$\textbf{if} \quad B_1 \rightarrow S_1 \quad [\!] \quad B_2 \rightarrow S_2 \quad [\!] \quad \cdots \quad [\!] \quad B_n \rightarrow S_n \quad \textbf{fi} \tag{4}$$

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

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

[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.

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

2.2 Main results

Dijkstra and Scholten [3] consider various classes of computations. One of them 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, written $[S](\alpha, \beta)$, 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 [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. ■

In this paper, we shall be interested in a more general class of computations. This class, written $[S_1; \ldots; S_n](\alpha_0, \ldots, \alpha_n)$, where $S_1, \ldots, S_n$ are commands and $\alpha_0, \ldots, \alpha_n$ are formulae, represents the class of all computations under control of $S_1; \ldots; S_n$ that start in a state satisfying α_0, terminate in a state satisfying α_n and, in addition, any state of any computation from this class that can be reached after executing $S_1; \ldots; S_i$ $(0 < i < n)$ satisfies α_i.

Suppose that $\mathcal{C} = [S_1; S_2; \ldots; S_n](\alpha_0, \ldots, \alpha_n)$. We define two sequences of formulae, $SP_1^{\mathcal{C}}, \ldots, SP_n^{\mathcal{C}}$ and $WLP_n^{\mathcal{C}}, \ldots, WLP_0^{\mathcal{C}}$, by the following induction:

$$SP_0^{\mathcal{C}} = \alpha_0; \qquad WLP_n^{\mathcal{C}} = \neg\alpha_n$$

and for $0 < i \leq n$

$$SP_i^{\mathcal{C}} = \alpha_i \wedge sp(S_i, SP_{i-1}^{\mathcal{C}}); \quad WLP_{n-i}^{\mathcal{C}} = \neg\alpha_{n-i} \vee wlp(S_{n-i+1}, WLP_{n-i+1}^{\mathcal{C}}).$$

Let $\mathcal{C}=[S_1; \ldots; S_n](\alpha_0, \ldots, \alpha_n)$. For each $0 \leq i \leq n$, we write $ST_{\mathcal{C}}(i)$ to denote 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 $SP_i^{\mathcal{C}} \wedge \neg WLP_i^{\mathcal{C}}$ holds in a state σ iff there exists a computation from $\mathcal{C}$ such that $\sigma \in ST_{\mathcal{C}}(i)$, $0 \leq i \leq n$. ■

Let $\mathcal{C}=[S_1; \ldots; S_n](\alpha_0, \ldots, \alpha_n)$. We write $D_i(\mathcal{C})$, $0 \leq i \leq n$, to denote the formula $SP_i^{\mathcal{C}} \wedge \neg WLP_i^{\mathcal{C}}$. In view of Theorem 2, $D_i(\mathcal{C})$ provides the complete characterization of i-states. Accordingly, it will be referred to as the *description of the i-states of* $\mathcal{C}$.

By a *subcommand* S of a command S' we mean a substring S of S' which is also a command. In particular, for any command S, S is a subcommand of S.

We say that a variable x is *inaffected* by a command S iff S does not contain a subcommand of the form $x := e$.

The following result will be useful.

Proposition 1 Let σ and σ' be initial and final states, respectively, of some computation under control of S. If x is inaffected by S, then $\sigma(x) = \sigma'(x)$. ■

Corollary 1 Let $\mathcal{C}=[S_1;\ldots;S_n](\alpha_0,\ldots,\alpha_n)$. Suppose that $\sigma \in ST_{\mathcal{C}}(i)$ and $\sigma' \in ST_{\mathcal{C}}(j)$ $(0 \leq i,j \leq n)$. Assume further that σ and σ' are members of the same computation $c \in \mathcal{C}$. If x is inaffected by any of $S_1,\ldots,S_n$, then $\sigma(x) = \sigma'(x)$. ■

3 Action languages

In this section, we show how to define action languages using Dijkstra's methodology and illustrate this by specifying a simple action language $\mathcal{L}$ that will be used in examples.

To define an action language, one starts with an alphabet consisting of primitive symbols from the following pairwise disjoint classes: (1) A countable set of Boolean variables, called *fluents* (these serve to describe the application domain under consideration); (2) A denumerable set of auxiliary Boolean variables, called *abnormality variables*: ab_1,ab_2,...; (3) A countable set of *action symbols*; (4) Two truth-constants: T and F; (5) Usual sentential connectives.

The classes (2), (4)-(5) are fixed, whereas the classes (1) and (3) varies from an alphabet into an alphabet. Accordingly, each alphabet is uniquely determined by its fluents and action symbols.

The alphabet of our action language $\mathcal{L}$ contains three fluents, a, l and s, standing for *alive*, *loaded* and *stolen*, respectively, and three action symbols *load* (a gun), *shoot* (a turkey) and *leave* (a car overnight in a garage).

A *formula* is a Boolean combination of fluents, abnormality variables and truth-constants. A formula containing no abnormality variables is said to be an *observation*.

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

$$SC = \langle\alpha_0\rangle\ A_1; \langle\alpha_1\rangle\ A_2; \ldots; \langle\alpha_{n-1}\rangle\ A_n\ \langle\alpha_n\rangle \tag{5}$$

where $\alpha_0,\ldots,\alpha_n$ are observations and $A_1,\ldots,A_n$ are action symbols. The scenario has the following intuitive interpretation: α_0 was observed to hold in the initial state, then the actions $A_1,\ldots,A_n$ were sequentially performed, and α_i, for $0 < i \leq n$, was observed to hold after performing the action A_i. In particular, α_n is the formula observed in the final state. In intermediate states, if no observation is made, i.e. $\alpha_i \equiv T$, we omit this element in the scenario.

Example 1 Below is a variant of the famous Yale Shooting Scenario [4].

$\langle a \wedge \neg l\rangle\ load; shoot\ \langle a\rangle$ ■

As we remarked earlier, our approach is based on the assumption that to each action symbol A, there are assigned two specifications, describing normal and abnormal performance of A. For the action symbols from the language $\mathcal{L}$, these specifications are the following. The action *load*, when performed normally, makes the gun loaded; when performed abnormally, does not cause any changes in the world. The action *shoot*, when performed normally, makes the gun unloaded and the turkey dead, provided that the gun was loaded before and, when performed abnormally, makes the gun unloaded. Finally, *leave*, when performed normally does not cause any changes in the world; when performed abnormally, it makes the car stolen.

The crucial point in using Dijkstra's methodology is to interpret action symbols occurring in a scenario as commands of the programming language defined in the previous section. The construction of such a command is a two step process. Firstly, with each action symbol A we associate two commands, denoted $S_1(A)$ and $S_2(A)$, describing respectively normal and abnormal performance of A.[9] For action symbols occurring in the language $\mathcal{L}$, $S_1(load)$ is $l := T$, $S_2(load)$ is $skip$, $S_1(shoot)$ is **if** $l \rightarrow a := F; l := F \;[\!]\; \neg l \rightarrow skip$ **fi**, $S_2(shoot)$ is $l := F$, $S_1(leave)$ is $skip$ and $S_2(leave)$ is $s := T$. Given $S_1(A)$ and $S_2(A)$, we define a *command schema* corresponding to A, written $S(A)$, by

if $\neg ab \rightarrow S_1(A)$ $[\!]$ $ab \rightarrow S_2(A)$ **fi**

where ab is a parameter which can be replaced by any abnormality variable from $\{ab_1, ab_2, \ldots\}$.[10] Accordingly,

$S($ *load*$)$ is **if** $\neg ab \rightarrow l := T$ $[\!]$ $ab \rightarrow skip$ **fi**,

$S($ *shoot*$)$ is **if** $\neg ab \rightarrow$ **if** $l \rightarrow a := F; l := F \;[\!]\; \neg l \rightarrow skip$ **fi** $[\!]$ $ab \rightarrow l := F$ **fi**

$S($ *leave*$)$ is **if** $\neg ab \rightarrow skip$ $[\!]$ $ab \rightarrow s := T$ **fi**.

Given the above command schemata, the chosen action symbols can be provided with Dijkstra-style semantics. Performing routine calculations one easily obtains:

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

[9] The ability to represent the effects (normal and abnormal) of chosen action symbols as commands of the programming language is the necessary condition to use Dijkstra's approach to reasoning about action and change. We do not claim here that this is always possible. However, most of the actions that have been considered in the AI literature enjoy this property.

[10] It should be stressed that normality/abnormality does not concern an action, but rather its performance. Accordingly, we should use different abnormality variables not only for different action symbols, but also for different occurrences of the same action symbol.

4 Action scenarios as classes of computations

Let SC be an action scenario given by (5). We write $\mathcal{C}(SC)$ to denote the class of computations

$$[S^1(A_1); \ldots ; S^n(A_n)](\alpha_0, \ldots, \alpha_n) \tag{6}$$

where $S^i(A_i)$ is $S(A_i)$ with ab replaced by ab_i.

We would like to identify the scenario SC with the class of computations $\mathcal{C}(SC)$. Unfortunately, $\mathcal{C}(SC)$ is too large to properly represent the scenario SC. The reason is that replacing an action symbol A, occurring in the i-th position of SC, by the command **if** $\neg ab_i \rightarrow S_1(A)$ $[\!]$ $ab_i \rightarrow S_2(A)$ **fi** we make no distinction between normal and abnormal performance of the action represented by A.[11] On the other hand, our approach is based on the implicite assumption that normal performances of actions are to be preferred over abnormal ones. In the rest of this section, we specify a subclass of $\mathcal{C}(SC)$ that captures this intuition.

We start with some preliminary terminology.

Let c be a computation from the class $\mathcal{C}(SC)$. We write $c(i)$, $0 \leq i \leq n$, to denote the state that is reached by c after performing the commands $S^1(A_1); \ldots ; S^i(A_i)$. We say that an action A_i is *realized normally in a computation* c iff the variable ab_i is assigned the value 0 in the state $c(i-1)$.

Proposition 2 An action A_i from a scenario SC is realized normally in a computation $c \in \mathcal{C}(SC)$ iff the variable ab_i is assigned the value 0 in all states of c. ■

Let c be a computation from the class $\mathcal{C}(SC)$, where SC is given by (5). We write $AB(c)$ to denote the set of these members of $\{ab_1, \ldots, ab_n\}$ which are assigned the value 1 in all states from c.[12]

Let c and c' be two computations from the class $\mathcal{C}(SC)$. We write $c \leq_{AB} c'$ iff $AB(c) \subseteq AB(c')$. We say that c is *AB-minimal* in the class $\mathcal{C}(SC)$ iff for every $c' \in \mathcal{C}(SC)$, if $c' \leq_{AB} c$ then $AB(c) = AB(c')$.

Clearly, the intended computations are those in which as many actions as possible are realized normally:

Definition 1 Let SC be an action scenario given by (5). The *intended class of computations corresponding to* SC, written $\mathcal{IC}(SC)$, is the class of all AB-minimal elements from $\mathcal{C}(SC)$. ■

5 Reasoning about scenarios

In this section, we provide a method to reason about action scenarios.

We shall be interested in the following reasoning task: "Given an action scenario $SC = \langle \alpha_0 \rangle\ A_1; \langle \alpha_1 \rangle\ A_2; \ldots ; \langle \alpha_{n-1} \rangle A_n\ \langle \alpha_n \rangle$, a formula γ and an integer k such that $0 \leq k \leq n$, determine whether γ holds after performing the actions

[11] In the sequel we shall not distinguish between an action symbol and the action it represents.

[12] By Proposition 2, $AB(c)$ characterizes these actions from $\{A_1, \ldots, A_n\}$ which are realized abnormally in a computation c.

$A_1, \ldots, A_k$".[13] Identifying a scenario SC with the class of computations $\mathcal{IC}(SC)$, the above reasoning task can be stated more formally: "Given a scenario $SC = \langle\alpha_0\rangle\ A_1; \langle\alpha_1\rangle\ \ A_2; \ldots; \langle\alpha_{n-1}\rangle A_n\ \langle\alpha_n\rangle$, a formula γ and an integer k such that $0 \leq k \leq n$, determine whether γ holds in all k-states of $\mathcal{IC}(SC)$.[14]

Consider a scenario $SC = \langle\alpha_0\rangle\ A_1; \langle\alpha_1\rangle\ \ A_2; \ldots; \langle\alpha_{n-1}\rangle A_n\ \langle\alpha_n\rangle$ and the class of computations $\mathcal{C}(SC)$ given by (6). In view of Theorem 2, we know that the set of all k-states of $\mathcal{C}(SC)$ is characterized by the formula

$$D_k(\mathcal{C}(SC)) = SP_k^{\mathcal{C}(SC)} \wedge \neg WLP_k^{\mathcal{C}(SC)} \tag{7}$$

where the formulae $SP_k^{\mathcal{C}(SC)}$ and $WLP_k^{\mathcal{C}(SC)}$ have been defined in section 2.2. To characterize the set of k-states of the class $\mathcal{IC}(SC)$, we use Reiter's default logic [9]. More precisely, we define a default theory $\mathcal{T}_k(SC) = \langle A, \Delta\rangle$, where A (the set of axioms) consists of the formula $D_k(\mathcal{C}(SC))$ and Δ (the set of defaults) is specified by $\{: \neg ab_i / \neg ab_i | 1 \leq i \leq n\}$. The next theorem shows that extensions of the theory $\mathcal{T}_k(SC)$ provide a complete description of k-states of the class $\mathcal{IC}(SC)$.

Theorem 3 Let $SC = \langle\alpha_0\rangle\ A_1; \langle\alpha_1\rangle\ \ A_2; \ldots; \langle\alpha_{n-1}\rangle A_n\ \langle\alpha_n\rangle$ be an action scenario. A formula γ holds in all $k - states$ of the class $\mathcal{IC}(SC)$, $0 \leq k \leq n$, iff γ is the member of all extensions of the default theory $\mathcal{T}_k(SC)$. ■

Corollary 2 Let SC be specified as before. A formula γ holds after performing the actions $A_1; \ldots; A_k$, $0 \leq k \leq n$, iff γ is the member of all extensions of the default theory $\mathcal{T}_k(SC)$. ■

Remark 1 A scenario $SC = \langle\alpha_0\rangle\ A_1; \langle\alpha_1\rangle\ \ A_2; \ldots; \langle\alpha_{n-1}\rangle A_n\ \langle\alpha_n\rangle$ gives rise to $n+1$ different default theories: $\mathcal{T}_0(SC), \mathcal{T}_1(SC), \ldots \mathcal{T}_n(SC)$. It can be shown, however, that the extensions of all these theories are based on the same sets of generating defaults.[15] Accordingly, when sets of generating defaults for $\mathcal{T}_i(SC)$, $0 \leq i \leq n$, are computed, they can be used for all default theories corresponding to the scenario SC. ■

6 Examples

In this section we consider a few examples to illustrate the method introduced in the previous section. Recall that a command $S^i(action)$ is a command schema $S(action)$ in which the parameter ab is replaced by ab_i. The clauses specifying $sp(S(action), \alpha)$ and $wlp(S(action), \alpha)$ have been provided in section 3.

Example 1 (continued) This scenario (SC, for short) is an example of the temporal prediction. We have a complete description of the initial state, a partial

[13] 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*.

[14] The set of k-states of $\mathcal{IC}(SC)$ is specified as that of $\mathcal{C}(SC)$ (see section 2.2) restricted to the states occurring in the computations from $\mathcal{IC}(SC)$.

[15] See [9], for the definition of this notion.

description of the final state and the question we are interested in is whether the gun was loaded in the final state. The scenario is given by

$$\langle a \wedge \neg l \rangle \; load; shoot \; \langle a \rangle.$$

First, we calculate $D_2(\mathcal{C}(SC))$, where $\mathcal{C}(SC) = [S^1(load); S^2(shoot)](a \wedge \neg l, T, a)$.[16]

$$\begin{array}{rcl} D_2(\mathcal{C}(SC)) & = & SP_2^{\mathcal{C}(SC)} \wedge \neg WLP_2^{\mathcal{C}(SC)} \\ & = & a \wedge sp(S^2(shoot), sp(S^1(load), a \wedge \neg l)) \\ & = & a \wedge sp(S^2(shoot), \neg ab_1 \wedge l \wedge a \vee ab_1 \wedge a \wedge \neg l) \\ & = & a \wedge (\neg ab_2 \wedge ((\neg a \wedge \neg l \wedge \neg ab_1) \vee (\neg l \wedge ab_1 \wedge a)) \vee ab_2 \wedge \neg l \wedge a) \\ & \equiv & \neg ab_2 \wedge ab_1 \wedge a \wedge \neg l \vee ab_2 \wedge \neg l \wedge a. \end{array}$$

The default theory $\mathcal{T}_2(SC)$ consists of $D_2(\mathcal{C}(SC))$ as its unique axiom and the pair of defaults $\{: \neg ab_1/\neg ab_1, : \neg ab_2/\neg ab_2\}$. $\mathcal{T}_2(SC)$ has two extensions, E_1 and E_2, given by $Th(\{D_2(\mathcal{C}(SC)), \neg ab_1\})$ and $Th(\{D_2(\mathcal{C}(SC)), \neg ab_2\})$, respectively.[17] Since $\neg l$ is the member of both E_1 and E_2, we conclude that the gun was unloaded in the final state.

It is worth noting that the extensions E_1 and E_2 correspond to two possible courses of events: either *load* was performed normally and *shoot* abnormally (E_1) or *load* was performed abnormally and *shoot* normally (E_2).

Consider now the scenario SC' given by $\langle a \wedge \neg l \rangle \; load; \; \langle l \rangle \; shoot \; \langle a \rangle$. We leave it to the reader to check that the default theory $\mathcal{T}_2(SC')$ has one extension containing $\neg ab_1$, ab_2 and $\neg l$. Accordingly, we conclude that *load* was performed normally, *shoot* was performed abnormally and the gun was unloaded in the final state. ■

Example 2 Consider now the scenario SC given by

$$\langle a \wedge \neg l \rangle \; load; shoot; load \; \langle a \rangle.$$

The inteded conclusion is that in the final state the gun is loaded. We start by calculating $D_3(\mathcal{C}(SC))$, where $\mathcal{C}(SC) = [S^1(load); S^2(shoot); S^3(load)](a \wedge \neg l, T, T, a)$.

$$\begin{array}{rcl} D_3(\mathcal{C}(SC)) & = & SP_3^{\mathcal{C}(SC)} \wedge \neg WLP_3^{\mathcal{C}(SC)} \\ & = & a \wedge sp(S^3(load), sp(S^2(shoot), sp(S^1(load), a \wedge \neg l))) \\ & = & a \wedge sp(S^3(load), sp(S^2(shoot), \neg ab_1 \wedge l \wedge a \vee ab_1 \wedge a \wedge \neg l)) \\ & = & a \wedge sp(S^3(load), \neg ab_2 \wedge (\neg a \wedge \neg l \wedge \neg ab_1 \vee \neg l \wedge ab_1 \wedge a) \\ & & \vee ab_2 \wedge (\neg l \wedge \neg ab_1 \wedge a \vee \neg l \wedge ab_1 \wedge a)) \\ & \equiv & a \wedge sp(S^3(load), \neg ab_2 \wedge (\neg a \wedge \neg l \wedge \neg ab_1 \vee \neg l \wedge ab_1 \wedge a) \\ & & \vee ab_2 \wedge \neg l \wedge a) \\ & = & a \wedge (\neg ab_3 \wedge l \wedge (\neg ab_2 \wedge (\neg a \wedge \neg ab_1 \vee a \wedge ab_1) \vee (ab_2 \wedge a)) \vee \\ & & ab_3 \wedge (\neg ab_2 \wedge (\neg a \wedge \neg l \wedge \neg ab_1 \vee \neg l \wedge ab_1 \wedge a) \vee ab_2 \wedge \neg l \wedge a)) \\ & \equiv & \neg ab_3 \wedge l \wedge \neg ab_2 \wedge ab_1 \wedge a \vee \neg ab_3 \wedge l \wedge ab_2 \wedge a \\ & & \vee ab_3 \wedge \neg ab_2 \wedge ab_1 \wedge a \wedge \neg l \vee ab_3 \wedge ab_2 \wedge \neg l \wedge a. \end{array}$$

[16] 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.

[17] Note that E_1 contains ab_2 and E_2 contains ab_1.

The default theory $\mathcal{T}_3(SC) = \langle\{D_3(\mathcal{C}(SC))\}, \{: \neg ab_1/\neg ab_1, \quad : \neg ab_2/\neg ab_2, : \neg ab_3/\neg ab_3\}\rangle$ has two extensions, $E_1 = Th(\{D_3(\mathcal{C}(SC)), \neg ab_1, \neg ab_3\})$ and $E_2 = Th(\{D_3(\mathcal{C}(SC)), \neg ab_2, \neg ab_3\})$. Since l belongs to both E_1 and E_2, we conclude that the gun was loaded in the final state.

Observe that the scenario SC gives rise to two possible courses of events: either both occurrences of *load* were performed normally and *shoot* was performed abnormally (E_1) or the first occurrence of *load* was performed abnormally and *shoot* and the second occurrence of *load* were performed normally (E_2). ■

Example 3 (Stolen Car Scenario) This example was first described without solution by Kautz [5] and then solved by Baker [1]. There is one action symbol, *leave*, for leaving a car overnight in a garage. The car is left for two successive nights and after the second night it turns out that the car is stolen. The scenario is given by

$$SC = \langle\neg s\rangle\ leave; leave\ \langle s\rangle.$$

No conclusion is intended with respect to if the car was stolen after the first night.

We first calculate $D_1(\mathcal{C}(SC))$, where $\mathcal{C}(SC) = [S^1(leave); S^2(leave)](\neg s, T, s)$.

$$\begin{array}{rcl} D_1(\mathcal{C}(SC)) & = & SP_1^{\mathcal{C}(SC)} \wedge \neg WLP_1^{\mathcal{C}(SC)} \\ & = & sp(S^1(leave), \neg s) \wedge \neg wlp(S^2(leave), \neg s) \\ & = & (\neg ab_1 \wedge \neg s \vee ab_1) \wedge \neg((\neg ab_2 \supset \neg s) \wedge (ab_2 \supset F)) \\ & \equiv & (\neg ab_1 \wedge \neg s \vee ab_1) \wedge \neg((ab_2 \vee \neg s) \wedge (\neg ab_2)) \\ & \equiv & (\neg ab_1 \wedge \neg s \vee ab_1) \wedge (\neg ab_2 \wedge s \vee ab_2) \\ & \equiv & (\neg ab_1 \wedge \neg s \wedge ab_2) \vee (\neg ab_2 \wedge s \wedge ab_1) \vee (ab_1 \wedge ab_2). \end{array}$$

The default theory $\mathcal{T}_1 = \langle\{D_1(\mathcal{C}(SC))\}, \{: \neg ab_1/\neg ab_1, : \neg ab_2/\neg ab_2\}\rangle$ has two extensions $E_1 = Th(\{D_1(\mathcal{C}(SC)), \neg ab_1\})$ and $E_2 = Th(\{D_1(\mathcal{C}(SC)), \neg ab_2\})$. Since $\neg s \in E_1$ and $s \in E_2$, no conclusion can be derived with respect to whether the car was stolen after the first night. ■

7 Conclusions

We have combined Dijkstra's semantics for programming languages with Reiter's default logic to formalization of reasoning about action and change, where actions with abnormal effects are permitted. The presented approach can be used to represent a broad class of action scenarios, including those where actions with random effects are allowed. In addition, both temporal prediction and postdiction tasks can be properly dealt with, without requiring initial or final states to be completely specified.

In [7] we showed how the ramification problem can be solved using Dijkstra's formalism. There is no difficulty to deal with this problem in the presented framework, but the topic will be discussed in the future paper.

The task of implementation is another point of interest. For a given scenario SC, calculating $D_k(\mathcal{C}(SC))$ amounts to simple syntactic manipulations on formulae and can be performed very efficiently. The only computational problem is to compute extensions of a default theory $\mathcal{T}_k(SC)$. However, for simple action

scenarios, as those considered in this paper, we deal with propositional default logic which is decidable.

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

[5] Kautz, H. A., "The Logic of Persistence", in: *Proc. AAAI-86*, 1986, 401-405.

[6] Łukaszewicz,W., Madalińska-Bugaj, E., "Program Verification Techniques as a Tool for Reasoning about Action and Change", in: *KI-94: Advances in Artificial Intelligence, Proceedings of 18th German Conference on Artificial Intelligence*, Springer-Verlag, Lecture Notes in Artificial Intelligence, **861**, 226-236, 1994.

[7] Łukaszewicz,W., Madalińska-Bugaj, E., "Reasoning about Action and Change Using Dijkstra's Semantics for Programming Languages: Preliminary Report", to appear in *Proc. IJCAI-95*, 1995.

[8] Radzikowska, A., "Circumscribing features and fluents: Reasoning about Action with Default Effects". To appear in *Proc. ECSQARU-95*.

[9] Reiter, R., "A Logic for Default Reasoning", *Artificial Intelligence Journal*, **13**, 81-132, 1980.

[10] Sandewall, E., "*Features and Fluents: The Representation of Knowledge about Dynamical Systems*", Oxford Logic Guides, 30, Oxford Science Publications, 1994. University, Sweden.