Defeasible Reasoning and Partial Order Planning

Diego R. García, Alejandro J. García, and Guillermo R. Simari

Artificial Intelligence Research and Development Laboratory
Department of Computer Science and Engineering
Universidad Nacional del Sur – Av. Alem 1253, (8000) Bahía Blanca
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)*

Abstract. Argumentation-based formalisms provide a way of considering the defeasible nature of reasoning with partial and often erroneous knowledge in a given environment. This problem affects every aspect of a planning process. We will present an argumentation-based formalism that an agent could use for constructing plans starting from a previously introduced formalism. In such a formalism, agents represent their knowledge about their environment in Defeasible Logic Programming, and have a set of actions they can execute to affect their environment. These actions are defined in combination with a defeasible argumentation formalism. We will analyze the interplay of arguments and actions when constructing plans using Partial Order Planning techniques.

1 Introduction

In this paper, we introduce an argumentation-based formalism an agent could use for constructing plans using partial order planning techniques. In our proposed approach, actions and arguments will be combined by the agent to construct plans. As we will explain next, actions' preconditions can be satisfied by other actions' effects (as usual) or by conclusions supported by arguments that are based on inference rules and other actions effects. We will also show that besides those effects declared in the definition of actions, there could be more effects that the agent will be able to deduce using the argumentation-based reasoning formalism.

Defeasible argumentation is a powerful formalism suitable for reasoning with potentially contradictory information and in dynamic environments (see [11, 3, 2, 10, 8]). The formalism presented here is based on Defeasible Logic Programming (DeLP) [3], a defeasible argumentation formalism grounded in Logic Programming. For dealing with contradictory and dynamic information in DeLP, arguments for conflicting pieces of information are build and then compared to decide which one prevails. The argument that prevails provides a warrant for the information that it supports.

* Partially supported by SGCyT Universidad Nacional del Sur, CONICET (PIP 5050) and Agencia Nacional de Promoción Cientfica y Tecnológica (PICT 2002 Nro 13096).

S. Hartmann and G. Kern-Isberner (Eds.): FoIKS 2008, LNCS 4932, pp. 311–328, 2008.

In [13] a formalism that combines actions and defeasible argumentation was introduced. There, they show the problems related to the combination of their formalism with simple planning algorithms.

Extending the mentioned work, in this paper we will analyze the interaction of arguments and actions when they are combined to construct plans using Partial Order Planning techniques. When actions and arguments are combined in a partial order plan, new types of interferences appear (called threats in [7]). These interferences need to be identified and resolved to obtain valid plans.

The main contribution of this paper will be to show meaningful examples and the description of the proposed solution for the combination of Partial Order Planning and Defeasible Argumentation. Thus, our work focuses on improving the capabilities and scope of current planning technology and not in improving the efficiency of current planning implementations.

2 Motivation

To solve a planning problem, an agent should be provided with an appropriate set of actions to perform. The representation of these actions must consider all the preconditions and effects that are relevant to solve the problem. Consider for example the consequences of the action of striking a match. A relevant effect can be to produce fire. However, there are many other consequences that can be entailed and could be considered irrelevant and not be included in the representation of the action (e. g., to produce light, to raise the temperature of the room, to make smoke, etc).

We propose that, instead of considering all the possible effects in the representation of the actions, agents should be provided with a defeasible reasoning formalism for obtaining those consequences that are entailed from action's effects. For example, an action for turning the switch on to light a room (called "*turn_switch_on*" from now on) should have as a precondition that the *switch is set to off* and the effect should be that the *switch is set to on*. The effect "*there is light in the room*" should be entailed as a consequence of the effect the *switch is on*. Thus, besides the action *turn_switch_on*, we propose to include the (defeasible) rule: "*if the switch is set to on then there is a reason to believe that there is light in the room*". It is important to note that if "*there is light in the room*" is considered as an effect of the action *turn_switch_on*, then it will be difficult to consider exceptions like "*there is no electricity* ". However, this kind of problems are easily handled by the argumentation formalism. For example, this situation could be represented by the defeasible rule "*if the switch is set to on but there is no electricity then there is a reason to believe that there is no light in the room*".

3 Defeasible Argumentation and Actions

In this section, we introduce a formalism that combines actions and defeasible argumentation based on a previous work reported in [13, 12, 4]. Our formalism

follows the logic programming paradigm for knowledge representation called Defeasible Logic Programming (DeLP) [3]. Thus, the agent's knowledge will be represented by a DeLP program and the agent will be able to perform defeasible reasoning over this knowledge.

The agent's knowledge base will be a defeasible logic program $\mathcal{K} = (\Psi, \Delta)$, where Ψ should be a consistent set of *facts*, and Δ a set of *defeasible rules*. Defeasible Rules are denoted $L_0 \prec L_1, \ldots, L_n$, where L_0 is a ground literal and $\{L_i\}_{i>0}$ is a set of ground literals. A defeasible rule "$Head \prec Body$" is the key element for introducing *defeasibility* [8] and is understood as expressing that *"reasons to believe in the antecedent Body of a rule provide reasons to believe in its consequent, Head"* [14]. Following Lifschitz [6], DeLP rules could be represented as *schematic rules* with variables, making abstraction of the object constants. The resulting DeLP programs are therefore *schematic programs*.

Strong negation "$\sim$" can appear in the head of defeasible rules, and it could be used to represent conflicting information. In DeLP arguments for conflicting pieces of information are built and then compared to decide which one prevails. Since the notion of argument will be extensively used in this paper its definition adapted from [3] is included below:

Definition 1 [Argument]
Let L be a literal, and $\mathcal{K} = (\Psi, \Delta)$ a defeasible logic program. We say that $\langle \mathcal{A}, L \rangle$ is an argument for L (or L is supported by $\mathcal{A}$) if $\mathcal{A}$ is a set of defeasible rules of Δ, such that:

1. there exists a derivation for L from $\Psi \cup \mathcal{A}$,
2. the set $\Psi \cup \mathcal{A}$ is non-contradictory, and
3. $\mathcal{A}$ is minimal: there is no proper subset $\mathcal{A}'$ of $\mathcal{A}$ such that $\mathcal{A}'$ satisfies conditions (1) and (2).

Example 1. Let (Ψ, Δ) be a knowledge base, where $\Psi = \{a, b, c, d\}$ and $\Delta = \{(p \prec b),\ (q \prec r),\ (r \prec d),\ (\sim r \prec s), (s \prec b),\ (\sim s \prec a, b),\ (w \prec b), (\sim w \prec b, c)\}$. From the defeasible logic program (Ψ, Δ) the literal p is supported by the argument $\mathcal{A}$=$\{p \prec b\}$, the literal q by $\mathcal{A}_1$= $\{(q \prec r),\ (r \prec d)\}$, the literal $\sim r$ by $\mathcal{A}_2$= $\{(\sim r \prec s),\ (s \prec b)\}$, the argument $\mathcal{A}_3$= $\{(\sim s \prec a, b)\}$ supports $\sim s$, and $\mathcal{A}_4$= $\{s \prec b\}$ supports the liteal s. Observe that $\mathcal{A}_4$ is a subargument of $\mathcal{A}_2$, *i.e.,* $\mathcal{A}_4$ is a subset of $\mathcal{A}_2$ that supports an inner conclusion in $\mathcal{A}_2$.

Given a defeasible logic program it is possible to generate arguments that are in conflict. For instance, in Example 1, $\mathcal{A}_3$ and $\mathcal{A}_4$ are in conflict because both support contradictory conclusions. Thus, $\mathcal{A}_3$ is a *counterargument* for $\mathcal{A}_4$ (and viceversa). Observe that a counterargument can also be in conflict with an inner part of other argument. For instance, $\mathcal{A}_2$ is a counterargument for $\mathcal{A}_1$, because $\mathcal{A}_2$ is in conflict with the subargument $\{r \prec d\}$ of $\mathcal{A}_1$. In this case, we also say that $\mathcal{A}_2$ attacks $\mathcal{A}_1$ at the point r.

Since conflicting arguments can be generated, DeLP provides a mechanism for deciding which argument prevails and therefore, which literals are warranted. Next, we will describe briefly this mechanism (see [3] for further details).

In DeLP, a literal L is *warranted* from (Ψ, Δ) if there exists a non-defeated *argument* $\mathcal{A}$ supporting L. To establish whether $\langle \mathcal{A}, L \rangle$ is a non-defeated argument, *counter-arguments* that could be *defeaters* for $\langle \mathcal{A}, L \rangle$ are considered. An argument $\mathcal{B}$ is a defeater for $\mathcal{A}$, if $\mathcal{B}$ is counter-argument for $\mathcal{A}$ and by some comparison criterion is preferred to $\langle \mathcal{A}, L \rangle$. In the examples in this paper we will use *generalized specificity* [15], a criterion that favors two aspects in an argument: it prefers (1) a *more precise* argument (*i.e.,* with greater information content) or (2) a *more concise* argument (*i.e.,* with less use of rules). A defeater $\mathcal{D}$ for an argument $\mathcal{A}$ can be *proper* ($\mathcal{D}$ is preferred to $\mathcal{A}$) or *blocking* (same strength). In Example 1, $\mathcal{A}_3$ is preferred to $\mathcal{A}_2$ (more precise) hence $\mathcal{A}_3$ is a proper defeater for $\mathcal{A}_2$. As stated above, the argument $\mathcal{A}_1$ is a counterargument for $\mathcal{A}_2$. Since the subargument $\{r \prec d\}$ of $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same strength, then $\mathcal{A}_2$ is a blocking defeater for $\mathcal{A}_1$. It is important to note that in DeLP the argument comparison criterion is modular and can be replaced. Thus, the most appropriate criterion for the domain that is being represented can be selected.

Since defeaters are arguments, there may exist defeaters for them, and defeaters for these defeaters, and so on. Thus, a sequence of arguments called *argumentation line* appears, where each argument defeats its predecessor in the line (see Example 2).

To avoid undesirable sequences, that may represent circular or fallacious argumentation lines, in DeLP an argumentation line is *acceptable* if it satisfies certain constraints. That is, the argumentation line has to be finite, an argument can not appear twice, and supporting arguments, *i.e.,* arguments in odd positions, (resp. interfering arguments) have to be not contradictory (see [3] for details). Given an acceptable argumentation line $[\mathcal{A}_1,\ldots,\mathcal{A}_n]$ we will say that C *is acceptable wrt* $[\mathcal{A}_1,\ldots,\mathcal{A}_n]$ if $[\mathcal{A}_1,\ldots,\mathcal{A}_n,\mathcal{C}]$ is an acceptable argumentation line.

Clearly, there can be more than one defeater for a particular argument $\mathcal{A}$. Therefore, many acceptable argumentation lines could arise from $\mathcal{A}$, leading to a tree structure. Given an argument $\langle \mathcal{A}, h \rangle$, a *dialectical tree* [3] for $\langle \mathcal{A}, h \rangle$, denoted $\mathcal{T}(\langle \mathcal{A}, h \rangle)$, is a tree where every node is an argument. The root of $\mathcal{T}(\langle \mathcal{A}, h \rangle)$ is $\langle \mathcal{A}, h \rangle$, and every inner node is a defeater (proper or blocking) of its parent. Leaves correspond to non-defeated arguments. In a dialectical tree every path from the root to a leaf corresponds to a different acceptable argumentation line. Thus, a dialectical tree provides a structure for considering all the possible acceptable argumentation lines that can be generated for deciding whether an argument is defeated. We call this tree *dialectical* because it represents an exhaustive dialectical analysis for the argument in its root.

Given a literal h and an argument $\langle \mathcal{A}, h \rangle$, to decide whether a literal h is warranted, every node in the dialectical tree $\mathcal{T}(\langle \mathcal{A}, h \rangle)$ is recursively marked as "D" (*defeated*) or "U" (*undefeated*), obtaining a marked dialectical tree $\mathcal{T}^*(\langle \mathcal{A}, h \rangle)$. Nodes are marked by a bottom-up procedure that starts marking all leaves in $\mathcal{T}^*(\langle \mathcal{A}, h \rangle)$ as "U"s. Then, for each inner node $\langle \mathcal{B}, q \rangle$ of $\mathcal{T}^*(\langle \mathcal{A}, h \rangle)$, $\langle \mathcal{B}, q \rangle$ will be marked as "U" iff every child of $\langle \mathcal{B}, q \rangle$ is marked as "D", or $\langle \mathcal{B}, q \rangle$ will be marked as "D" iff it has at least a child marked as "U".

Given an argument $\langle \mathcal{A}, h\rangle$ obtained from (Ψ, Δ), if the root of $\mathcal{T}^*(\langle \mathcal{A}, h\rangle)$ is marked as "U", then we will say that $\mathcal{T}^*(\langle \mathcal{A}, h\rangle)$ *warrants* h and that h is *warranted* from (Ψ, Δ).

Example 2 (Extends Example 1)**.** Argument $\mathcal{A}$ for p is undefeated because there is no counter-argument for it. Hence, p is warranted.

The literal q has the argument $\mathcal{A}_1$ that is defeated by $\mathcal{A}_2$ that attacks r, an inner point in $\mathcal{A}_1$. The argument $\mathcal{A}_2$ is in turn defeated by $\mathcal{A}_3$= $\{(\sim s \prec a, b)\}$. Thus, the argumentation line $[\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3]$ is obtained. The literal q is warranted because its supporting argument $\mathcal{A}_1$ has only one defeater $\mathcal{A}_2$ that is defeated by $\mathcal{A}_3$, and $\mathcal{A}_3$ has no defeaters.

Observe that there is no warrant for $\sim r$ because $\mathcal{A}_2$ is defeated by $\mathcal{A}_3$. The literals t and $\sim t$ have no argument, so neither of them is warranted. Finally note that every fact of Ψ is warranted, because no counter-argument can defeat a fact. Thus, the set of warranted literals from (Ψ, Δ) is $\{a, b, c, d, p, q, r, \sim s, \sim w\}$.

Besides its knowledge base $\mathcal{K}$, an agent will have a set of actions Γ that it may use to change its world. The formal definitions that were introduced in [12, 13] are recalled below.

Definition 2 [Action]. An action A is an ordered triple $A = \langle \mathsf{X}, \mathsf{P}, \mathsf{C}\rangle$, where A is a ground atom representing the name of the action, X is a consistent set of ground literals representing consequences of executing A, P is a set of ground literals representing preconditions for A, and C is a set of constraints of the form *not* L, where L is a ground literal. We will denote actions as follows:

$$\{X_1, \ldots, X_n\} \xleftarrow{A} \{P_1, \ldots, P_m\}, not\ \{C_1, \ldots, C_k\}$$

where *not* $\{C_1, \ldots, C_k\}$ represents $\{not\ C_1, \ldots, not\ C_k\}$.

Example 3. Let Γ be the set of available actions of for agent:

$$\Gamma = \left\{ \begin{array}{c} \{\sim a, d, x\} \xleftarrow{Ac_1} \{a, p, q\}, not\ \{t, \sim t, w\} \\ \{e\} \xleftarrow{Ac_2} \{p\}, not\ \{\} \\ \{e\} \xleftarrow{Ac_3} \{t\}, not\ \{w\} \\ \{\sim p\} \xleftarrow{Ac_4} \{b\}, not\ \{q\} \end{array} \right\}$$

Note that all the atoms and literals considered in an action definition are ground. However, *schematic actions* (operators) can be defined using non-ground atoms and literals. An *schematic action* stands for the set of all possible ground action instances generated using the object constants.

The condition that must be satisfied before an action $A = \langle \mathsf{X}, \mathsf{P}, \mathsf{C}\rangle$ can be executed contains two parts: P, which mentions the literals that *must* be warranted, and C, which mentions the literals that *must not* be warranted. In this way, the conditions that must be satisfied to execute an action could also depend on the fact that some information is unknown (*un-warranted*).

Definition 3 [Applicable Action]. Let $\mathcal{K} = (\Psi, \Delta)$ be an agent's knowledge base. Let Γ be the set of actions available to this agent. An action A in Γ, is applicable if every precondition P_i in P has a warrant built from (Ψ, Δ) and every constraint C_i in C fails to be warranted.

Example 4. Let $\mathcal{K}$= (Ψ, Δ) be the agent's knowledge base as defined in Example 1, and the set of actions Γ from Example 3. As shown in Example 2, the set of warranted literals from (Ψ, Δ) is $\{a, b, c, d, p, q, r, \sim s, \sim w\}$. Then, from (Ψ, Δ) the action Ac_1 is applicable because every literal in its precondition set ({a,p,q}) is warranted, and no constraints in $\{t, \sim t, w\}$ are warranted. The action Ac_2 is also applicable because it has no constraint and its precondition p is warranted. Finally, action Ac_3 is not applicable because its precondition t is not warranted, and action Ac_4 is not applicable because its constraint q it is warranted.

It is clear that only applicable actions can be executed, and the agent has to plan which one to execute. Once the agent selects an action to apply, its execution will affect directly the agent environment. That is, once an action has been applied, the effect of the action will change both the environment and the set $\mathcal{K}$. The effect of the execution of an applicable action in our formalism is defined below:

Definition 4 [Action Effect]. Let $\mathcal{K} = (\Psi, \Delta)$ be an agent's knowledge base. Let Γ be the set of actions available to this agent. Let A be an applicable action in Γ defined by:

$$\{X_1, \ldots, X_n\} \xleftarrow{A} \{P_1, \ldots, P_m\}, not\ \{C_1, \ldots, C_k\}$$

The effect of executing A is the revision of Ψ by X, i.e. $\Psi^{*\mathsf{X}} = \Psi^{*\{X_1,\ldots,X_n\}}$. Revision will consist of removing any literal in Ψ that is complementary of any literal in X and then adding X to the resulting set. Formally:

$$\Psi^{*\mathsf{X}} = \Psi^{*\{X_1,\ldots,X_n\}} = (\Psi \setminus \overline{\mathsf{X}}) \cup \mathsf{X}$$

where $\overline{\mathsf{X}}$ is the set of complements of members of X.

Example 5 (Extends Example 4)**.** The action Ac_1 was shown to be applicable from (Ψ, Δ). If Ac_1 is executed, Ψ then becomes $\Psi_1 = \{b, c, \sim a, d, x\}$. Observe that the precondition a was "consumed" by the action, and the literals $\sim a$ and x were added. It is important to note that from (Ψ_1, Δ) the set of warranted literals changes to $\{\sim a, x, b, c, d, p, s, \sim w\}$. Therefore, the action Ac_1 is now not applicable again because from (Ψ_1, Δ) there is no warrant for a and q. However, the action Ac_4 that was not applicable from (Ψ, Δ) is now applicable from (Ψ_1, Δ) because its constraint q is not warranted. Observe that action Ac_2 remains applicable.

The argumentation formalism described above allows an agent to represent knowledge about the environment and to define the actions it can perform. It also defines when an action is applicable and how to compute its effects. However, it does not describe how to construct a plan to achieve the agent's goals.

In our approach a *planning problem* is defined by the tuple (Ψ, Δ, *Goal*, Γ) where Ψ is a set of literals that represents the *initial state*, Δ is the set of defeasible rules that agent can use for reasoning, *Goal* is a set of literals representing the agent's *goals* and Γ is a set of *actions* that the agent can perform. The agent will satisfy its goals when, through the execution of a sequence of actions, it reaches some state Ψ' where each literal of *Goal* is warranted from (Ψ',Δ).

Next, we will describe how partial order planning techniques can be combined with the formalism described above to provide the agent with the ability to build plans.

4 Argumentation in Partial Order Planning

The basic idea behind a regression Partial Order Planning (POP) algorithm [7] is to search through the plan space. The planner starts with an initial plan consisting solely of a *start* step (whose effects encode the initial state conditions) and a $finish$ step (whose preconditions encode the goals) (see Figure 1(a)). Then it attempts to complete this initial plan by adding new steps (actions) and constraints until all step's preconditions are guaranteed to be satisfied. The main loop in a traditional POP algorithm makes two types of choices:

- *Supporting unsatisfied preconditions*: all steps that could possibly achieve a selected unsatisfied precondition are considered. It chooses one step nondeterministically and then adds a causal link to the plan to record that the selected precondition is achieved by the chosen step.
- *Resolve threats*: If a step might possibly interfere with the precondition being supported by a casual link, it nondeterministically chooses a method to resolve this *threat*: either by reordering steps in the plan (adding *ordering constraints*) or posting additional subgoals.

Recall that in the argumentation formalism described in the previous section, an action is applicable if every precondition of the action has a warrant built from the agent's current knowledge base, and every constraint fails to be warranted. To combine this formalism with POP, we must consider the use of arguments for supporting unsatisfied preconditions, besides actions.

In this section we will illustrate with an example how to build a plan using actions and arguments. When actions and arguments are combined to construct plans, new types of interferences (threats) appear that need to be resolved to obtain a valid plan. In Section 5 we will identify these new types of threats and methods to resolve each of them will be proposed. Finally, in Section 6 we will propose an extension to the traditional POP algorithm that use actions and arguments to built plans and resolve the new types of threats.

The following definitions are introduced for identifying different sets of literals present in an argument, that will be considered when a plan is constructed.

Definition 5 [Heads-Bodies-Literals]. Given an argument $\langle B, h\rangle$, *heads*($\mathcal{B}$) is the set of all literals that appear as heads of rules in $\mathcal{B}$. Similarly, *bodies*($\mathcal{B}$)

is the set of all literals that appear in the bodies of rules in $\mathcal{B}$. The set of all literals appearing in $\mathcal{B}$, denoted *literals*($\mathcal{B}$) is the set $heads(\mathcal{B}) \cup bodies(\mathcal{B})$.

Definition 6 [Argument base]. Given an argument $\langle B, h\rangle$ we will say that the base of $\mathcal{B}$ is the set $base(\mathcal{B}) = bodies(\mathcal{B}) - heads(\mathcal{B})$.

Definition 7 [Conclusion]. Given an argument $\langle B, h\rangle$ we will say that the conclusion of $\mathcal{B}$ is the literal $conclusion(\mathcal{B}) = heads(\mathcal{B}) - bodies(\mathcal{B})$.

Example 6. Given the argument $\langle B, b\rangle$ where $\mathcal{B}$= $\{(b \prec c, d), (c \prec e)\}$, the corresponding sets are:

$$\begin{array}{ll} heads(\mathcal{B}) = & \{b, c\} \\ bodies(\mathcal{B}) = & \{c, d, e\} \\ literals(\mathcal{B}) = & \{b, c, d, e\} \end{array} \qquad \begin{array}{ll} base(\mathcal{B}) = & \{d, e\} \\ conclusion(\mathcal{B}) = & \{b\} \end{array}$$

The combined use of argumentation and actions to build plans introduces new issues not present in the traditional POP algorithm that need to be addressed. Following, we will present an example to illustrate how traditional POP algorithm can be extended to consider arguments as planning steps. We will also introduce the basic terminology and graphical representation that will be used in the rest of the paper. For simplicity, we present a propositional planning problem that defines actions without constraints.

Example 7. Consider an agent that works at night and its job is cleaning rooms in a building. The agent arrives to a room where the light switch is set to off and has to build a plan for having that room cleaned. The agent has the following knowledge base: $\Psi = \{switch\_off\}$ and

$$\Delta = \left\{ \begin{array}{c} light\_in\_room \prec switch\_on \\ \sim light\_in\_room \prec switch\_on, \sim electricity \end{array} \right\}$$

The agent's goal is $G = \{room\_clean\}$, and the available actions are:

$$\{room\_clean\} \overset{clean\_room}{\longleftarrow} \{light\_in\_room\}, not\ \{\}$$
$$\{switch\_on, \sim switch\_off\} \overset{turn\_switch\_on}{\longleftarrow} \{switch\_off\}, not\ \{\}.$$

Figure 1(a) shows the initial plan for example 7 and Figure 1(b) depicts an incomplete plan where only actions (not arguments) were considered to achieve the unsatisfied preconditions. Finally, Figure 1(c) shows a complete plan obtained using actions and arguments.

In our approach, we will distinguish between two types of steps: *action steps* (*i.e.,* steps that represent the execution of an action) and *argument steps* (*i.e.,* arguments used in the plan to support the precondition of some action step). *Action steps* are depicted by square nodes labeled with the action name. The squares labeled START and FINISH represent the *start* and $finish$ steps respectively. The literals that appear below an action step represent the preconditions

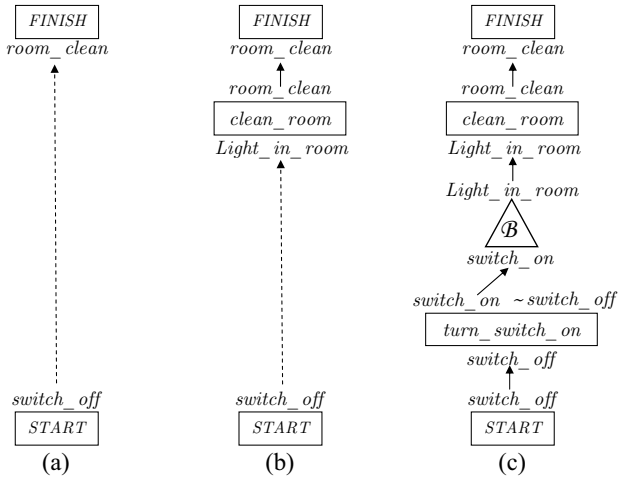

Fig. 1. Different partial plans for Example 7

of the action step, and the literals that appear above represent its effects. Triangles represent *argument steps* and are labeled with the argument name. The literal at the top of the triangle is the conclusion of the argument (Definition 7), and the literals at the base of the triangle represent the base of the argument (Definition 6).

The solid arrows that link an effect of an action step with a precondition of another action step, or with a literal in the base of an argument step, represent *causal links*. The solid arrows between the conclusion of an argument step and a precondition of an action step represent *support links*. *Causal* and *support links* are used to explicitly record the source for each literal during planning.

Dashed arrows represent *ordering constraints* and are used to explicitly establish an order between two steps. By definition the *start* step comes before the $finish$ step and the rest of the steps are constrained to come after the *start* step and before the $finish$ step. All causes are constrained to come before their effects, so a causal link also represents an ordering constraint.

In Figure 1(a) there is only one unsatisfied precondition *room_clean*, and the only possible way to satisfy it is by the action *clean_room*. A new step *clean_room* is added (Figure 1(b)) to the plan and it's precondition *light_in_room* becomes a new unsatisfied subgoal. Observe that none of the actions available achieve *light_in_room*, then it is not possible to obtain a plan if only actions are considered.

However, from the rules Δ of the agent's knowledge base it is possible to construct the (potential) argument $\mathcal{B}$=$\{$ $(light\_in\_room \prec switch\_on)$ $\}$ that supports *light_in_room*. Therefore, an alternative way to achieve *light_in_room* would be to use $\mathcal{B}$ for supporting *light_in_room*, and then to find a plan for satisfying all the literals in the base of $\mathcal{B}$ ($base(\mathcal{B}) = \{switch\_on\}$). Figure 1(c) shows this situation. The argument $\mathcal{B}$ is chosen to support *light_in_room* and the literal

switch_on becomes a new subgoal of the plan. Then, the action *turn_switch_on* is selected to satisfy *switch_on* and the corresponding step is added to the plan. The precondition $switch\_off$ of the step *turn_switch_on* is achieved by the *start* step, so the corresponding causal link is added and a plan is obtained.

Note that $\mathcal{B}$=$\{$ $(light\_in\_room \prec switch\_on)$ $\}$ is a "*potential argument*" because it is conditioned to the existence of a plan that satisfies its base. This argument can not be constructed from a set of facts, as usual in DeLP. The reason is that at the moment of the argument construction it is impossible to know which literals are true, because they depend on steps that will be chosen later in the planning process. A formal definition of potential argument follows:

Definition 8 [Potential Argument]
Let h be a literal, and Δ a set of defeasible rules. We say that $\langle\langle \mathcal{A}, h\rangle\rangle$ is a potential argument for h (or that $\mathcal{A}$ is a potential argument supporting h), if $\mathcal{A}$ is a set of defeasible rules of Δ, such that:

1. there exists a defeasible derivation for h from $base(\mathcal{A}) \cup \mathcal{A}$
2. the set $base(\mathcal{A}) \cup \mathcal{A}$ is non-contradictory, and
3. $\mathcal{A}$ is minimal: there is no proper subset $\mathcal{A}'$ of $\mathcal{A}$ such that $\mathcal{A}'$ satisfies conditions 1. and 2.

Another thing to consider is that the existence of the argument $\mathcal{B}$ in the plan shown in Figure 1(c) is not enough to have a warrant for *light_in_room*, because it could exist a defeater for $\mathcal{B}$ (for example, when there is no electricity in the building). Recall that to be able to apply an action, all its preconditions have to be warranted. The existence of a defeater for $\mathcal{B}$ will depend on the decisions made later in the planning process, that is, a defeater for $\mathcal{B}$ could appear as new action steps are added to the plan.

5 Interferences among Actions and Arguments

When only actions are considered, there is only one type of destructive interference that can arise in a plan. In the traditional POP algorithm, this interference is captured by the notion of *threat* (see Figure 2). When actions and arguments are combined to construct plans, new types of interferences appear that need to be identified and resolved to obtain a valid plan. We will extend the notion of *threat* to identify all the different types of interferences that could arise in a plan and propose methods to resolve each of them.

Figure 2(a) shows the kind of threat that appears in the traditional POP algorithm: the precondition p of A_1, supported by A_2, is threatened by the action step A_3 because it negates p. Note that $\overline{p}$ is an effect of A_3, where $\overline{p}$ stands for the complement of p with respect to strong negation, *i.e.,* $\overline{p}$ is $\sim p$ and $\overline{\sim p}$ is p. The way to resolve this threat is to add an ordering constraint to make sure that A_3 is not executed between A_2 and A_1. There are two alternatives: A_3 is forced to come before A_2 (called *demotion*, see Figure 2(b)) or A_3 is forced to come after A_1 (called *promotion*, see Figure 2(c)).

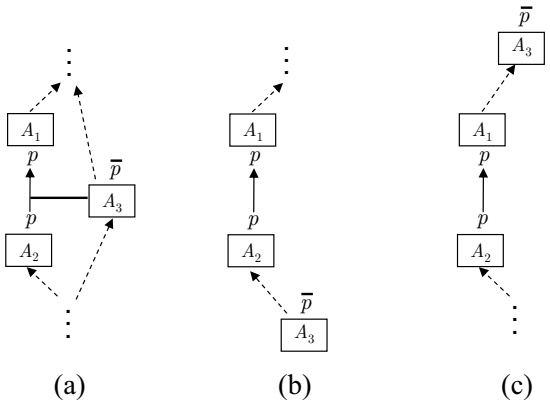

Fig. 2. An action step threatens the precondition supported by another action step

This type of threat involves only action steps and will be called *action-action threat*. However, as we consider also arguments to construct a plan, a different kind of threat could arise involving action steps and argument steps. Consider the situation shown in Figure 3. In this case, the action step A_3 threatens the argument step $\mathcal{B}$ because it negates a literal present in the argument $\mathcal{B}$. Note that $\overline{n}$ is an effect of A_3 and $n \in literals(\mathcal{B})$ (see definition 5). The argument step $\mathcal{B}$ was added to the plan to support the precondition b of the action step A_1. If A_3 makes $\overline{n}$ true before A_1 is executed, the argument $\mathcal{B}$ will not exist at the moment a warrant for b is needed to execute A_1. This type of threat will be called *action-argument threat*.

This *action-argument threat* can be resolved ensuring that A_3 does not make $\overline{n}$ true just before the execution A_1. In general, this can be accomplished adding

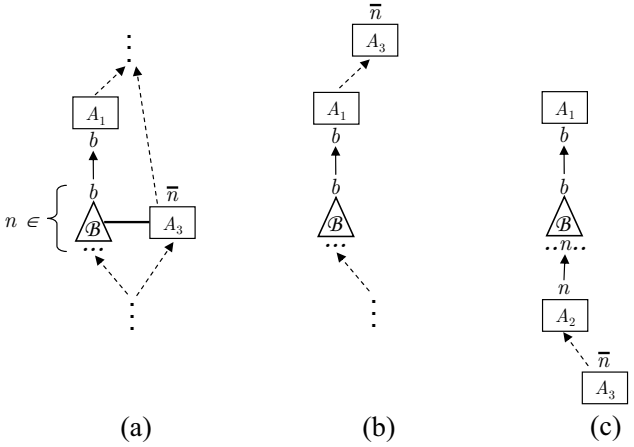

Fig. 3. An action step threaten and argument step

an ordering constraint to force A_3 to come after A_1. This method will be called *promotion** (see figure 3(b)). However, this is not the only way to resolve this threat. In the particular case that the literal n is present in the base of $\mathcal{B}$, there will be an action step A_2 in the plan that achieves it. If $A_2 \neq start$, then the threat can be resolved adding an ordering constraint to force A_3 to come before A_2. This method will de called *demotion** (see figure 3(c)).

Finally, there is another type of threat to consider involving only arguments. Arguments are introduced in the plan to have a warrant for the precondition of some action step. Since the arguments could be defeated by a counter-argument, the precondition would not be warranted. This situation is shown in figure 4(a). The argument step $\mathcal{B}$ was added to the plan to support the precondition b of the action step A_1. However, at the moment a warrant for b is needed to execute A_1, there exists a defeater $\mathcal{C}$ for $\mathcal{B}$. We will refer to this type of threat as *argument-argument threat*. Observe that the defeater $\mathcal{C}$ is not necessarily an argument step of the plan. The argument $\mathcal{C}$ could be any defeater that can be built from the effects of the actions steps that could be ordered to come before A_1 in the plan.

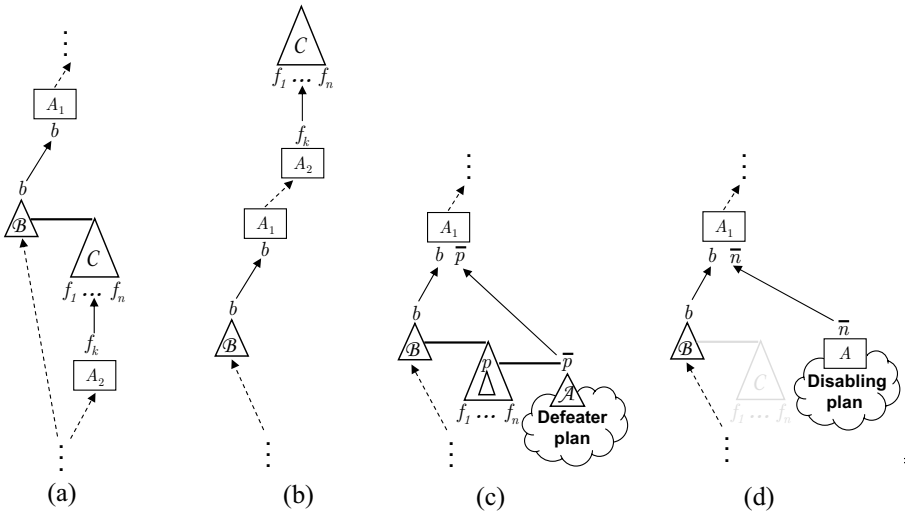

Fig. 4. An argument step is attacked by a counter-argument

This *argument-argument threat* is resolved ensuring that $\mathcal{B}$ is not defeated by $\mathcal{C}$. There are three alternative methods:

- *delaying the defeater*: Add orderings constraint to the plan to force *every* action step A_2 in the plan, that achieve a literal $f_k \in base(\mathcal{C})$, to come after A_1 (see figure 4(b)). This forces $\mathcal{C}$ to exist after A_1, therefore b is warranted at the moment is needed to execute A_1.
- *defeating the defeater*: add steps to the plan to force the defeater $\mathcal{C}$ to be defeated (see figure 4(c)). To accomplish this, the precondition $\overline{p}$ is added to

the step A_1, where $p \in heads(\mathcal{C})$, and the plan is completed to achieve this precondition. Since the intention is to defeat $\mathcal{C}$, the step chosen to support $\overline{p}$ must be an argument step.

– *disabling the defeater*: add steps to the plan to prohibit the existence of the defeater $\mathcal{C}$ (see figure 4(d)). To accomplish this, the precondition $\overline{n}$ is added to the step A_1, where $n \in literals(\mathcal{C})$, and the plan is completed to achieve this precondition. The step chosen to support $\overline{n}$ must be an action step, because the intention is to prohibit the existence of $\mathcal{C}$.

The proposed solutions for solving the new types of threats will be used in an algorithm presented in following section (See Figure 6).

6 Proposed Algorithm

In this section we will present an extension of the traditional POP algorithm that we will call APOP. Figure 5, 6 and 7 show an outline of the APOP algorithm. The $\downarrow$ identify input parameters and the $\uparrow$ identify output parameters. The statements **choose** and **fail** are used to describe nondeterminism. The primitive **choose** allow the algorithm to make a choice between different alternatives and keeps track of the pending choices. If the algorithm encounter a **fail** statement the control is resumed at the point in the algorithm where the choice was made and the pending choices are considered.

```
function APOP(Ψ↓, Δ↓, Goal↓, Γ↓): Plan;
begin
  Plan:= Make_Initial_Plan(Ψ, Goal);
   loop do
       if Plan.Subgoals = ∅  then return Plan;
       Let (SubGoal, Step, SubGoalType, ArgLine) ∈ Plan.Subgoals;
       Plan.Subgoals:= Plan.Subgoals − {(SubGoal, Step, SubGoalType, ArgLine)};
       Choose_Step(Plan, Δ, Γ, Step, SubGoal, SubGoalType, ArgLine);
       Resolve_Threats(Plan, Δ, Γ);
   end
end
function Make_Initial_Plan(Ψ↓,Goal↓): Plan;
begin
   Plan.Action_Steps:= {(START,{},Ψ), (FINISH, Goal,{})};

   Plan.Argument_Steps:= ∅;
   Plan.Orderings:= {START ≺ FINISH};
   Plan.Subgoals:= {(g,FINISH, any,[])| g ∈ Goal};

   Plan.Causal_Links:= ∅;
   Plan.Support_Links:= ∅;
   Plan.Defeated_Args:= ∅;
end
```

Fig. 5. Outline of the APOP algorithm part 1

The function `APOP` (Figure 5) proceeds as the traditional pop algorithm: starts with an initial plan and attempts to complete it by adding new steps, resolving

the threats that may appear. However, it will consider arguments and actions as planning steps and resolve the new types of threats introduced in Section 5. Besides the initial state Ψ and the *Goal*, function `APOP` takes Δ and Γ as input parameters. The set Δ contains defeasible rules that can be used for building arguments and Γ is the set of available actions. In APOP a plan is represented by seven sets (see `Make_Initial_Plan` in Figure 5).

In contrast with the traditional POP algorithm the procedure `Choose_Step` (Figure 6) will consider arguments, besides actions, to support unsatisfied subgoals (`Plan.Subgoals`). Note that if no arguments can be constructed to support a subgoal (`Arg_Steps=`$\emptyset$ in statement `(2)`) then only actions steps will be considered and the algorithm will proceed as POP (statement `(6)`). However, if `Arg_Steps`$\neq \emptyset$, the algorithm will consider the inclusion of an argument step to support a subgoal (statement `(7)`). In each case the plan will updated acordingly.

```
procedure Choose_Step(Plan↓↑, Δ↓, Γ↓, S_need↓, p↓, SubGoalType↓, ArgLine↓)
begin

(1)Act_Steps:= {S | S ∈ Plan.Action_Steps that possibly S ≺ S_need
                    or S is created using an action A ∈ Γ and p ∈ Effects(S) };
(2)Arg_Steps:= {S | S is created using a potential argument ⟨⟨B, p⟩⟩ from Δ
                    acceptable wrt ArgLine};
(3) case SubGoalType of
      action:   Steps:= Act_Steps;
      argument: Steps:= Arg_Steps;
      any:      Steps:= Act_Steps ∪ Arg_Steps;
    endcase
(4) if Steps= ∅ then fail;
(5) choose S_add from Steps;
(6) if S_add ∈ Act_Steps then
       Plan.Orderings := Plan.Orderings ∪ {S_add≺S_need};
       Plan.Causal_Links:= Plan.Causal_Links ∪ {S_add --p--> S_need};
       if S_add ∉ Plan.Action_Steps then // S_add is a newly added step
           Plan.Action_Steps:= Plan.Action_Steps ∪ {S_add};
           Plan.Orderings:= Plan.Orderings ∪ {START ≺ S_add, S_add ≺ FINISH};
           Plan.Subgoals:= Plan.Subgoals ∪ {(g, S_add, any,[])| g ∈ Preconditions(S_add)};
       endif
    endif
(7) if S_add ∈ Arg_Steps then
       Plan.Argument_Steps:= Plan.Argument_Steps ∪ {S_add};
       Plan.Orderings:= Plan.Orderings ∪ {S_add≺S_need };
       Plan.Support_Links:= Plan.Support_Links ∪ {S_add ≻--p-- S_need};
       Plan.Subgoals:= Plan.Subgoals ∪ {(g, S_add, action,[])|g ∈ Argument_Base(S_add)};
    endif
end
```

Fig. 6. Outline of the APOP algorithm. Part 2: choosing and adding steps.

As new steps are added to the plan, new threats could appear. The function `Resolve_threats` (Figure 7) detects these threats and tries to resolve them using the methods proposed in section 5.

The statement `(1)` considers all *action-action threats* present in the plan and tries to resolve each of them choosing either *promotion* or *demotion* (see Figure 2). Following [7], we use "*possibly* $A_1 \prec A_3$" in the algorithm to express that $A_1 \prec A_3$ is consistent with the ordering constraint of the plan (`Plan.Orderings`).

```
procedure Resolve_Threats(Plan↓↑, Δ↓, Γ↓)
begin
(1)for each A2 -p-> A1 ∈ Plan.Causal_Links and A1 ∈ Plan.Action_Steps do
    for each A3 ∈ {A| A ∈ Plan.Action_Steps ∧ p̄ ∈ effects(A) ∧ possibly A2≺A≺A1} do
       choose either
          Promotion:
              if possibly A1≺A3 then
                Plan.Orderings:= Plan.Orderings ∪ {A1≺A3};
              else fail;
          Demotion:
              if possibly A3≺A2 then
                Plan.Orderings:= Plan.Orderings ∪ {A3≺A2};
              else fail;
        end
(2)for each B -b-≻ A1 ∈ Plan.Support_Links do
   begin
    S_A1 :={(Add,l)| ∃ Add ∈ Plan.Action_Steps ∧ l ∈ Effects(Add) ∧ possibly Add≺A1 ∧
                     ∄ Dell ∈ Plan.Action_Steps ∧ l̄ ∈ Effects(Dell) ∧ Add≺Dell≺A1 };
    Ψ_A1 :={l | (Add,l) ∈ S_A1};

(2.a) for each (A3,n̄) ∈ {(Add,n̄)| (Add,n̄) ∈ S_A1 ∧ n ∈ literals(B)} do
       choose either
          Promotion*:
             if possibly A1≺A3 then
               Plan.Orderings:= Plan.Orderings ∪ {A1≺A3};
             else fail;
          Demotion*:
             if n ∈ Base(B) and A2 -n-> B ∈ Plan.Causal_Links and possibly A3≺A2 then
                  Plan.Orderings:= Plan.Orderings ∪ {A3≺A2};
             else fail;
       end
(2.b) for each C ∈ {C| ⟨⟨C,q⟩⟩ is a potential argument from Δ that is a defeater for B,
        base(C)⊆ Ψ_A1, (C,B)∉ Plan.Defeated_Args and C is acceptable wrt Arg_Line(B)} do
        choose either
           Delaying the defeater:
              choose f from base(C)
              for each Add ∈ {Add |(Add,f) ∈ S_A1} do
                 if possibly A1≺Add then
                   Plan.Orderings:= Plan.Orderings ∪ {A1≺Add};
                 else fail;
           Defeating the defater:
               choose p from heads(C);
               Plan.Subgoals:= Plan.Subgoals ∪ (p̄,A1,argument,Arg_Line(B) + [⟨⟨C,q⟩⟩]);
               Plan.Defeated_Args:= Plan.Defeated_Args ∪ {(C,B)}

           Desabling the defater:
               choose n from literals(C);
               Plan.Subgoals:= Plan.Subgoals ∪ (n̄,A1, action,[]);

        end
  end
end
```

Fig. 7. Outline of the APOP algorithm. Part 3: Threats Resolution.

It is important to note that if the plan contains no argument steps then Plan.Support_Links= $\emptyset$, hence, statement `(2)` will not be executed and `Resolve_threats` will proceed as in traditional POP algorithm.

If the plan contains arguments, statement `(2.a)` considers all *action-argument threats* present in the plan and tries to resolve each of them choosing either

*promotion** or *demotion** (see Figure 3). Finally, statement `(2.b)` considers all *argument-argument threats* and tries to resolve each of them choosing either *delaying the defeater*, *defeating the defeater* or *disabling the defeater* (see Figure 4).

7 Related Work

The combination of defeasible reasoning and planning is not new [9,11]. However, in these works, the whole plan is viewed as an argument and then, defeasible reasoning is performed about complete plans. In contrast, our approach uses arguments for warranting subgoals, and hence, defeasible reasoning is used in a single step of the plan.

In [9], a planner is proposed that performs essentially the same search as POP, but by reasoning defeasibly about plans. That is, it reasons backwards from goals to subgoals, planning for conjunctive goals separately and then merging the plans for the individual goals into a combined plan for the conjunctive goal. Pollock argues that planning must be done defeasibly, making the default assumption that there are no threats and then modifying the plan as threats are discovered. Therefore, a planning agent will *infer defeasibly* that the merged plan is a solution to the planning problem. A *defeater* for this defeasible inference consists of discovering that the plan contains destructive interference. This interference refers to the traditional notion of *threat* that involves only actions. Although this approach combines defeasible reasoning and partial order planning, defeasible reasoning is not used in the same way as we propose in this work. He uses defeasible reasoning to reason about the plan as a whole, while we use defeasible reasoning to warrant subgoals *during* the planning process.

In [11], an argumentation-based approach for agents following the BDI model is introduced. They introduce different instantiations of Dung's abstract argumentation framework for generating consistent desires and consistent plans for achieving those desires. Although their work relates argumentation and plans, their approach differs considerably from ours. For them, a complete plan for a desire d is an "instrumental argument", *i.e.,* a set of planning rules that support d. Since complete plans are arguments, they introduce the notion of conflict among plans (arguments) and their approach defines which plan prevails after an argumentative analysis. Therefore, plans are more related to our notion of argument, than to our definition of plan. Finally, they use plans to justify the selected intentions of the agent.

8 Conclusions

In this paper we have introduced an argumentation-based formalism an agent could use for constructing plans using partial order planning technics. We have described how the traditional POP algorithm can be extended to consider arguments as planning steps.

When actions and arguments are combined to construct plans, new types of interferences appear. Therefore, we have extended the notion of threat to consider: *action-action*, *action-argument*, and *argument-argument threats*. Methods to resolve each type of threat have been proposed.

We have presented an algorithm called APOP, that extends the traditional POP algorithm to consider actions and arguments as planning steps and resolve the new types of threats using the proposed methods. A prototype implementation of this algorithm was implemented in Prolog.

This work was focused on improving the capabilities and scope of current planning technology and not in improving the efficiency of current planning implementations. Therefore, we have not made any comparison of the efficiency with other existing planners.

Future work includes the extension of our formalism to consider other features (e. g., conditional effects) that are present in other action representation languages like AL [1] and PDDL [5].

References

1. Baral, C., Gelfond, M.: Reasonig agents in dynamic domains. In: In Minker, J. (ed.) Logic-Based artificial intelligence, pp. 257–259. Kluwer Academic Publishers, Dordrecht (2000)
2. Chesñevar, C.I., Maguitman, A.G., Loui, R.P.: Logical Models of Argument. ACM Comp. Surveys 32(4) (December 2000)
3. García, A.J., Simari, G.R.: Defeasible logic programming: An argumentative approach. Theory and Practice of Logic Programming 4(1), 95–138 (2004)
4. Garcia, D.R., Simari, G.R., Garcia, A.J.: Planning and Defeasible Reasoning. In: AAMAS 2007. Proceedings of the Sixth Intl. Joint Conf. on Autonomous Agents and Multi-Agent Systems, pp. 856–858 (2007)
5. Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., Wilkins, D.: Pddl—the planning domain definition language (1998)
6. Lifschitz, V.: Foundations of logic programs. In: Brewka, G. (ed.) Principles of Knowledge Representation, pp. 69–128. CSLI Pub., Stanford (1996)
7. Penberthy, J., Weld, D.S.: UCPOP: A Sound, Complete, Partial Order Planner for ADL. In: Proc. of the 3rd. Int. Conf. on Principles of Knowledge Representation and Resoning, pp. 113–124 (1992)
8. Pollock, J.: Cognitive Carpentry: A Blueprint for How to Build a Person. MIT Press, Cambridge (1995)
9. Pollock, J.: Defeasible Planning. In: Bergmann, R., Kott, A. (Cochairs) Integrating Planning, Scheduling, and Execution in Dynamic and Uncertain Environments AIPS Workshop (1998)
10. Prakken, H., Vreeswijk, G.: Logical systems for defeasible argumentation. In: Gabbay, D. (ed.) Handbook of Philosophical Logic, 2nd edn. Kluwer Academic Pub., Dordrecht (2000)
11. Rahwan, I., Amgoud, L.: An argumentation-based approach for practical reasoning. In: Proc. AAMAS, pp. 347–354 (2006)

12. Simari, G.R., García, A.J.: Actions and arguments: Preliminaries and examples. In: Proc. VII Congreso Argentino en Ciencias de la Computación, Argentina, pp. 273–283 (October 2001)
13. Simari, G.R., García, A.J., Capobianco, M.: Actions, Planning and Defeasible Reasoning. In: NMR 2004. Proceedings of the 10th International Workshop on Non-Monotonic Reasoning, pp. 377–384 (2004), ISBN: 92-990021-0-X
14. Simari, G.R., Loui, R.P.: A Mathematical Treatment of Defeasible Reasoning and its Implementation. Artificial Intelligence 53, 125–157 (1992)
15. Stolzenburg, F., García, A., Chesñevar, C.I., Simari, G.R.: Computing Generalized Specificity. Journal of Non-Classical Logics 13(1), 87–113 (2003)