Conditional Planning with External Functions

Davy Van Nieuwenborgh[1,⋆], Thomas Eiter[2,⋆⋆], and Dirk Vermeir[1]

[1] Vrije Universiteit Brussel, VUB
Dept. of Computer Science
Pleinlaan 2, B-1050 Brussels, Belgium
{dvnieuwe,dvermeir}@vub.ac.be
[2] Institute of Information Systems,
Vienna University of Technology, Austria
eiter@kr.tuwien.ac.at

Abstract. We introduce the logic-based planning language $\mathcal{K}^c$ as an extension of $\mathcal{K}$ [5]. $\mathcal{K}^c$ has two advantages upon $\mathcal{K}$. First, the introduction of external function calls in the rules of a planning description allows the knowledge engineer to describe certain planning domains, e.g. involving complex action effects, in a more intuitive fashion then is possible in $\mathcal{K}$. Secondly, in contrast to the conformant planning framework $\mathcal{K}$, $\mathcal{K}^c$ is formalized as a conditional planning system, which enables $\mathcal{K}^c$ to solve planning problems that are impossible to express in $\mathcal{K}$, e.g. involving sensing actions. A prototype implementation of conditional planning with $\mathcal{K}^c$ is build on top of the DLV$^{\mathcal{K}}$system, and we illustrate its use by some small examples.

1 Introduction

In general, the task of a planning system consists of finding, dependent on the initial state, a sequence of actions such that a certain goal will be reached if, starting from that initial state, the actions in the sequence are executed in the correct order. In the context of logic-based languages, a number of frameworks have been proposed in the literature to logically describe and reason about such planning problems, e.g. [8,9,3,13,11,4,15,5,16].

In [4,5], the planning language $\mathcal{K}$ was introduced as a system for planning under incomplete knowledge, i.e. rather than describing transitions between states of the world (complete knowledge), one can describe in $\mathcal{K}$ transitions between states of (incomplete) knowledge. $\mathcal{K}$'s ability to deal with incomplete knowledge comes from the fact that its semantics is close in spirit to the answer set semantics [7], thus allowing negation as failure (naf) to be used in the causation rules of the planning description. As for answer sets, the semantics of $\mathcal{K}$ is defined in a two step process. First, the semantics is defined for planning descriptions without naf. Next, for arbitrary planning descriptions, a reduction is introduced, similar to the GL-reduct [7], which removes naf from the planning descriptions w.r.t. a candidate state transition. Finally, if the state transition

⋆ Supported by the Flemish Fund for Scientific Research (FWO-Vlaanderen).
⋆⋆ Supported by the Austrian Science Fund project FWF P16536-N04.

C. Baral, G. Brewka, and J. Schlipf (Eds.): LPNMR 2007, LNAI 4483, pp. 214–227, 2007.

is valid w.r.t. the reduct of the planning description, the state transition is valid for the planning description.

Although $\mathcal{K}$ is an expressive framework, it suffers from two shortcomings. The first problem encountered is a direct consequence of the fact that $\mathcal{K}$ is a conformant planning system. Consider e.g. the problem (taken from [15]) of defusing a bomb. To defuse the bomb, a special lock has to be placed in the locked position. If one defuses the bomb while the lock is unlocked, the bomb explodes and the person defusing the bomb is killed. The person defusing the bomb can determine if the bomb is locked or unlocked by looking at it, and she can switch the lock from the locked to the unlocked position and vice-versa. Obviously, no action can be undertaken once the bomb has exploded. A possible encoding of this problem in $\mathcal{K}$ is depicted below.

```
fluents: exploded. locked. unlocked. disarmed. dead.
actions: disarm. turn. look.
always: caused exploded after disarm, unlocked.
        caused disarmed after disarm, locked.
        caused unlocked after turn, locked.
        caused locked after turn, unlocked.
        caused dead if exploded.
        caused locked if not unlocked after look.
        caused unlocked if not locked after look.
        executable disarm if not exploded, not dead.
        nonexecutable disarm if not locked, not unlocked.
        executable turn if not exploded, not dead.
        executable look if not exploded, not dead.
        inertial dead.
        noConcurrency.
goal: disarmed?(3)
```

One can check that no conformant plan[1] for the above problem exists. Indeed, if the action *look* is performed we know that the bomb is either locked or unlocked, but both cases need a different, but incompatible, strategy for defusing the bomb. In such cases, one needs to switch from conformant to conditional planning. A conditional plan for the above problem could be

$$look; case \left\{ \begin{array}{c} \{locked\} \rightarrow disarm \\ \{unlocked\} \rightarrow turn; disarm \end{array} \right. \quad (1)$$

Thus, the person defusing the bomb first looks at the bomb. If it is locked, she disarms it; but if it is unlocked, she first turns the switch and then disarms the bomb.

A second problem with $\mathcal{K}$ is encountered when one needs to describe effects of actions that are not easily captured by logic programming rules. Consider the following variant of the Bubble Breaker game. We have a grid of a certain height and certain width and on each position in the grid we have a colored bubble. We can tap on any position in the grid, but if we tap a position, the biggest connected (left/right/up/down directions) region of bubbles with the same color and including the tapped position, is removed from the board. In case the biggest connected region only contains the tapped position itself, nothing is removed. After the bubbles are removed, the remaining bubbles in each column fall down, such that there are no holes between the bubbles in the same column. An illustration of the course of the game is provided in Figure 1. The goal of the game is to tap certain positions in such a way that all the bubbles are removed from the board.

[1] A conformant (or secure) plan is a plan that always leads to a goal state when it is executed.

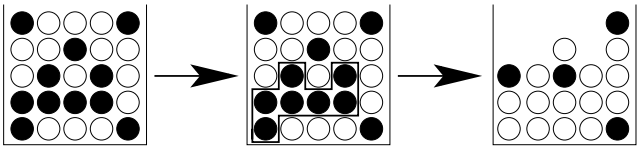

Fig. 1. Left: the initial configuration of the board. Middle: the region that is selected after tapping position $(1, 1)$, i.e. the lower left corner. Right: the configuration after the selected block is removed.

Clearly, it is not so easy to describe the effects of the *tap* action using causation rules. First, one has to compute the biggest connected region, afterwards that region has to be removed and finally the blocks on top of the removed region have to fall down. One solution is to add an additional action *update* and some extra fluents to take care of this complicated process of computing the effects of the *tap* action. The complete encoding in $\mathcal{K}$ can be found in [17], but we will briefly describe parts of it here.

First, some (non)executability statements are needed for the tap and update action, i.e. when there are holes between bubbles in the same column, the update action is executable and the tap action is not.

```
executable update if pos(X,Y,C), not placed(X,Yn), succ(Yn,Y).
nonexecutable tap(X,Y) if pos(A,B,C), not placed(A,Bn), succ(Bn,B).
```

Next, to compute the region that has to be removed from the board and to actually remove that region, one can use the following set of causation rules.

```
caused remove(X,Y,C) after tap(X,Y), pos(X,Y,C).
caused remove(Xn,Y,C) if remove(X,Y,C) after pos(Xn,Y,C), succ(X,Xn).
caused remove(Xn,Y,C) if remove(X,Y,C) after pos(Xn,Y,C), succ(Xn,X).
caused remove(X,Yn,C) if remove(X,Y,C) after pos(X,Yn,C), succ(Y,Yn).
caused remove(X,Yn,C) if remove(X,Y,C) after pos(X,Yn,C), succ(Yn,Y).
caused pos(X,Y,C) if not remove(X,Y,C) after pos(X,Y,C), tap(A,B).
```

Finally, to encode that the bubbles have to fall down, the update action will be executed a number of times and each time all the bubbles with a free position below them are moved one position lower.

```
caused pos(X,Yn,C) after update, pos(X,Y,C), not placed(X,Yn), succ(Yn,Y).
caused pos(X,1,C) after update, pos(X,1,C).
caused pos(X,Y,C) after update, pos(X,Y,C), pos(X,Yn,Cn), succ(Yn,Y).
```

Note that a plan for this problem using the encoding in $\mathcal{K}$ will always be of the form

$$tap(x, y); update; \ldots, update; tap(v, w); update; \ldots; tap(k, l); \ldots; update; tap(a, b).$$

To solve problems like the above, we propose to extend the action language $\mathcal{K}$ by allowing external functions to be called[2] in the causation rules of an action description. Intuitively, an external function call allows a knowledge engineer to import fluent information from an external source in response to an action that is performed. These external function calls are inspired by the DLVHEX system [6], i.e. a system for answer

[2] We assume that the external functions have no side effects when they are called.

set programming with higher-order atoms and external evaluations. For this reason, our external functions will use the same notation as the ones in DLVHEX.

Reconsider e.g. the Bubble Breaker game and the following causation rule involving an external function $\&nbc$, which stands for $new\_board\_configuration$

$$caused\ pos(Xn,\ Yn,\ Cn)\ after\ \&nbc[X,\ Y](Xn,\ Yn,\ Cn), tap(X,\ Y)\ .$$

When the action tap is executed for a certain position (X, Y), the external function $\&nbc$ will return a set of tuples of the form (Xn, Yn, Cn), with (Xn, Yn) a position and Cn a color, that correspond to the new configuration of the grid when a tap action is executed on the given position. The above rule imports this output into the fluent pos, yielding that the above rule can be intuitively read as "executing the action tap on position X and Y of the grid causes the fluents $pos(Xn, Yn, Cn)$ to become true if the tuple (Xn, Yn, Cn) is an output of the external function call $\&nbc$".

The external function in the previous rule is deterministic, i.e. for a given configuration of the grid and a position (X, Y), the external function always produces the same output tuples. However, when we reconsider the defusing a bomb problem with external functions $\&look\_effect$ and $\&disarm\_effect$ computing the effects of the *look* (resp. *disarm*) action, we get non-deterministic outcomes. E.g., consider the following causation rules

$$caused\ X\ after\ \&look\_effect()[X], look.$$
$$caused\ X\ after\ \&disarm\_effect()[X], disarm.$$

Intuitively, the external functions will return, when their corresponding action gets executed, some fluents as output which are imported into the planning process by the above causation rules. Clearly, the outcomes of these functions are non-deterministic. E.g., disarming the bomb when you don't know if it is locked or not can either yield *disarmed* or *exploded*. Further, this example on non-deterministic external functions demonstrates that the proposed extension to $\mathcal{K}$ can be used to simulate sensing actions [12,10,14,15,16]. By combining an ordinary action together with an external function that materializes the sensed values of executing the "sensing" action, we have very flexible means to encode sensing, e.g. sensing more than one fluent, or a combination of fluents at the same time; or sensing that depends on what is known (or not known) in the current (incomplete) state, ...

The rest of the paper is organized as follows. In Section 2 we introduce the syntax of $\mathcal{K}^c$, while its semantics is defined in Section 3. Before concluding in Section 5, we discuss our prototype implementation in section 4.

2 Syntax of $\mathcal{K}^c$

A *signature* of a planning domain with external actions PD is a tuple $PD_{sig} = (\sigma^{act}, \sigma^{fl}, \sigma^{ec}, \sigma^{typ}, \sigma^{con}, \sigma^{var})$, where σ^{act}, σ^{fl}, σ^{ec} and σ^{typ} are mutually disjoint sets of respectively action, fluent, external function and type names. The names in σ^{act}, σ^{fl} and σ^{typ} actually correspond to predicate symbols, so we associate with each of them an arity $n \geq 0$. On the other hand, the names in σ^{ec} correspond to external predicate symbols,

with whom we associate both an input arity i and an output arity o ($i, o \geq 0$)[3]. Further, σ^{con} and σ^{var} are mutually disjoint sets of respectively constants and variable symbols[4].

For a given signature PD_{sig}, an *action atom* is defined as $p(t_1, \ldots, t_n)$, where $p \in \sigma^{act}$, n is the arity associated with p and $t_1, \ldots, t_n \in \sigma^{con} \cup \sigma^{var} \cup \sigma^{fl}$. We define *fluent atoms* and *type atoms* similarly by substituting $p \in \sigma^{act}$ by $p \in \sigma^{fl}$ or $p \in \sigma^{typ}$ respectively. An *external function call* is defined as $\&p[i_1, \ldots, i_m](o_1, \ldots, o_n)$, where $p \in \sigma^{ec}$, m, n are the input and output arities associated with p and $i_1, \ldots, i_m, o_1, \ldots, o_n \in \sigma^{con} \cup \sigma^{var} \cup \sigma^{fl}$. A *variable atom* is defined as $X(t_1, \ldots, t_n)$, $n \geq 0$, where $X \in \sigma^{var}$ and $t_1, \ldots, t_n \in \sigma^{con} \cup \sigma^{var} \cup \sigma^{fl}$.

An action (resp. fluent, external function call, type, variable) literal is an action (resp. fluent, external function call, type, variable) atom a or its classical negation $\neg a$. For a set of literals X we use $\neg X$ to denote the set $\{\neg p \mid p \in X\}$, where $\neg(\neg a) \equiv a$ for an atom a. Furthermore, we use X^+ (resp. X^-) to denote the set of positive (resp. negative) literals in X. To denote the set of all action (resp. fluent, external function call, type, variable) literals that can be formed using the signature, we use $\mathcal{L}_{act}$ (resp. $\mathcal{L}_{fl}, \mathcal{L}_{ec}, \mathcal{L}_{typ}, \mathcal{L}_{vat}$). In addition, we use $\mathcal{L}_{fl,typ} = \mathcal{L}_{fl} \cup \mathcal{L}_{typ}$, $\mathcal{L}_{dyn} = \mathcal{L}_{fl} \cup {\mathcal{L}_{act}}^+ \cup {\mathcal{L}_{ec}}^+$ and $\mathcal{L} = \mathcal{L}_{fl,typ} \cup {\mathcal{L}_{act}}^+ \cup {\mathcal{L}_{ec}}^+$.

All actions, fluents, and external function calls that can be used in a planning description have to be declared using the following declaration rules.

Definition 1. *An **action** (resp. **fluent**) **declaration** is an expression of the form*

$$p(X_1, \ldots, X_m) \textit{ requires } t_1, \ldots, t_n$$

where either $p \in \sigma^{act}$ (resp. $p \in \sigma^{fl}$) or $p \in \sigma^{var}$, i.e. either $p(X_1, \ldots, X_m) \in {\mathcal{L}_{act}}^+$ (resp. $p(X_1, \ldots, X_m) \in {\mathcal{L}_{fl}}^+$) or $p(X_1, \ldots, X_m) \in {\mathcal{L}_{vat}}^+$, and $X_1, \ldots, X_m \in \sigma^{var}$. Further, $t_1, \ldots, t_n \in \mathcal{L}_{typ}$, $n \geq 0$, and every X_i (and p if $p \in \sigma^{var}$) occurs in $t_1, \ldots, t_n$. Whenever $n = 0$, the keyword requires may be omitted.

*An **external function call declaration** is an expression of the form*

$$\&p[I_1, \ldots, I_m](O_1, \ldots, O_n) \textit{ requires } t_1, \ldots, t_k \textit{ ranges } r_1, \ldots, r_l$$

where $\&p[I_1, \ldots, I_m](O_1, \ldots, O_n) \in {\mathcal{L}_{ec}}^+$ and $I_1, \ldots, I_m, O_1, \ldots, O_n \in \sigma^{var}$. Further, $t_1, \ldots, t_k, r_1, \ldots, r_l \in \mathcal{L}_{typ}$, $k, l \geq 0$, and every I_i occurs in $t_1, \ldots, t_k$ and every O_i occurs in $r_1, \ldots, r_l$.

To describe the static and dynamic dependencies of fluents on other fluents, external functions, and actions, we introduce causation rules, while initial state constraints are used to describe the initial state of a planning problem.

Definition 2. *A **causation rule** (**rule**, for short) is an expression of the form*

$$\textit{caused } f \textit{ if } b_1, \ldots, b_k, \textit{not } b_{k+1}, \ldots, \textit{not } b_l \textit{ after } a_1, \ldots, a_m, \textit{not } a_{m+1}, \ldots, \textit{not } a_n$$

where $f \in \mathcal{L}_{fl} \cup \mathcal{L}_{vat} \cup \{false\}$, $b_1, \ldots, b_l \in \mathcal{L}_{fl,typ} \cup {\mathcal{L}_{ec}}^+$, $a_1, \ldots, a_n \in \mathcal{L}$, $l \geq k \geq 0$, and $n \geq m \geq 0$. Whenever $l = 0$ (resp. $n = 0$), the keyword if (resp. after) may be

[3] Note that predicate symbols with the same name, but different arities, are not allowed.

[4] As usual constants begin with a lower case letter, while variables start with an upper case letter.

*omitted. When both $l = n = 0$, the keyword caused may also be dropped. Rules where $n = 0$ are called **static rules**, while all other rules are called **dynamic rules**. A static rule preceded by the keyword initially, is called an **initial state constraints**.*

To access the different parts of a causation rule r (or an initial state constraint), we define $h(r) = f$, $post^+(r) = \{b_1, \ldots, b_k\}$, $post^-(r) = \{not\ b_{k+1}, \ldots, not\ b_l\}$, $pre^+(r) = \{a_1, \ldots, a_m\}$, $pre^-(r) = \{not\ a_{m+1}, \ldots, not\ a_n\}$, and $lit(r) = \{f, b_1, \ldots, b_l, a_1, \ldots, a_n\}$.

To allow conditional execution of actions, we define executability conditions.

Definition 3. *An **executability condition** is an expression of the form*

$$executable\ a\ if\ b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_l$$

where $a \in \mathcal{L}_{act}{}^+ \cup \mathcal{L}_{vat}$, $b_1, \ldots, b_l \in \mathcal{L}$ and $l \geq k \geq 0$. Whenever $l = 0$, i.e. the execution is unconditional, the keyword if may be omitted.

To access the different parts of an executability condition e, we define $h(e) = a$, $post^+(e) = post^-(e) = \emptyset$, $pre^+(e) = \{b_1, \ldots, b_k\}$, $pre^-(e) = \{not\ b_{k+1}, \ldots, not\ b_l\}$, and $lit(e) = \{a, b_1, \ldots, b_l\}$.

Furthermore, we define, for any rule, initial state constraint, or executability condition r, that $post(r) = post^+(r) \cup post^-(r)$ and $pre(r) = pre^+(r) \cup pre^-(r)$.

From $\mathcal{K}$, we also adopt the *safety restriction* notion, i.e. all rules (including initial state constraints) and executability conditions have to satisfy the syntactic restriction that all variables in a naf type literal must also occur in some literal which is not a naf type literal.

Definition 4. *An **action description** is a pair (D, R) where D is a finite set of action, fluent and external function declarations and R is a finite set of safe executability conditions, safe causation rules, and safe initial state constraints.*

*A **planning domain** is a pair $PD = (\Pi, AD)$, where Π is a logic program over the literals of $\mathcal{L}_{typ}$ admitting exactly one answer set, and AD is an action description.*

*A **query** is of the form*

$$g_1, \ldots, g_m, not\ g_{m+1}, \ldots, not\ g_n?(i)$$

where $g_1, \ldots, g_n \in \mathcal{L}_{fl}$ are variable-free, and $i, j \geq 0, n \geq m \geq 0$.

*A **planning problem** is a pair (PD, q), where PD is a planning domain and q is a query.*

In appendix B and C of [17], one can find the $\mathcal{K}^c$ encodings of the Bubble Breaker game and the defusing a bomb problem respectively. Especially note the difference in readability between the encoding, of the same problem, in $\mathcal{K}$ from Appendix A of [17] and the one in $\mathcal{K}^c$ from Appendix B of [17].

3 Semantics of $\mathcal{K}^c$

3.1 Instantation

Similar to the grounding of a logic program, we instantiate a planning problem such that the semantics can be defined more easily. The main difference with classical grounding

is that we only allow correctly typed action, fluent, and external function call literals to be generated.

A substitution is any function $\theta : \sigma^{var} \mapsto \sigma^{con} \cup \sigma^{fl}$, i.e. a function assigning constants or fluent names to variables. The application of a substitution θ can be extended to any syntactic object x by defining $\theta(x)$ as the object x' obtained from x by replacing every $X \in \sigma^{var}$ that occurs in x, and that is defined by θ, with $\theta(x)$.

First, we define the valid instantiations of the fluents, actions, and external functions.

Definition 5. *Let $PD = (\Pi, (D, R))$ be a planning domain, and let M be the unique answer set of Π.*

*For a fluent (resp. action) declaration $d \in D$ and a substitution θ that is at least defined over $X_1, \ldots, X_m$ (and p if $p \in \sigma^{var}$), we say that $\theta(p(X_1, \ldots, X_m))$ is a **legal fluent** (resp. **action**) **instantiation** if $\{\theta(t_1), \ldots, \theta(t_n)\} \subseteq M$ and $\theta(p) \in \sigma^{fl}$ (resp. $\theta(p) \in \sigma^{act}$) if $p \in \sigma^{var}$.*

*For an external function declaration $d \in D$ and a substitution θ that is at least defined over $I_1, \ldots, I_m, O_1, \ldots, O_n$, $\theta(\&p[I_1, \ldots, I_m](O_1, \ldots, O_n))$ is a **legal external function call instantiation** if $\{\theta(I_1), \ldots, \theta(I_m), \theta(O_1), \ldots, \theta(O_n)\} \subseteq M$.*

To denote the set of all legal fluent (resp. action, external function call) instantiations of PD and their classical negations, we will use $\mathcal{L}^{fl}_{PD}$ (resp. $\mathcal{L}^{act}_{PD}, \mathcal{L}^{ec}_{PD}$). In addition, we use $\mathcal{L}_{PD} = \mathcal{L}^{fl}_{PD} \cup \mathcal{L}^{act}_{PD}{}^{+} \cup \mathcal{L}^{ec}_{PD}{}^{+}$.

Using the above, we can define the instantiation of a planning domain.

Definition 6. *Let $PD = (\Pi, (D, R))$ be a planning domain. The instantiation of PD, denoted $PD\downarrow$, is defined as $PD\downarrow = (\Pi\downarrow, (D, R\downarrow))$, where $\Pi\downarrow$ is the grounding of Π over σ^{con} and $R\downarrow = \{\theta(r) \mid r \in R, \theta \in \Theta_r\}$, where Θ_r is the set of all substitutions θ that define all the variables in r, such that*

- *$lit(\theta(r)) \cap \mathcal{L}_{dyn} \subseteq \mathcal{L}_{PD}$;*
- *$(post^{+}(\theta(r)) \cup pre^{+}(\theta(r))) \cap \mathcal{L}_{typ} \subseteq M$;*
- *$h(r) \in \mathcal{L}^{fl}_{PD} \cup \{false\}$ if r is a causation rule or an initial state constraint; and*
- *$h(r) \in \mathcal{L}^{act}_{PD}{}^{+}$ if r is an executability condition.*

Intuitively, the above ensures that in the instantiation of PD all actions, fluents and external function calls agree with their declarations, that positive type literals agree with the background knowledge, that causation rules or initial state constraints should cause a fluent literal and that executability conditions should have an action in their head. A planning domain PD is said to be *ground* if PD and $PD\downarrow$ coincide.

From now on, we will assume that we are working with the grounded version of a given planning domain PD, i.e. we will implicitly replace PD by $PD\downarrow$.

3.2 Conditional Planning

A plan in $\mathcal{K}$ is a sequence of sets of actions. However, this approach is not feasible in the context of conditional planning, where one wants to cope with non-deterministic effects of actions. For this reason, we will introduce the concept of a conditional plan, i.e. a plan that allows to branch depending on the effects that are caused by executing an

action. Our notion of a conditional plan is inspired by the one from [16], and is limited to conditional plans with only the case-endcase construct[5].

In what follows, we use $\mathcal{P}(X)$ to denote the powerset of X.

Definition 7. *Let PD be a planning domain. A **conditional plan** for PD is defined inductively as follows:*

1. *A sequence of sets of actions $A_1; \ldots; A_k$, with $A_i \subseteq \mathcal{L}_{PD}^{act}{}^{+}$, is a conditional plan.*
2. *If we have a sequence of sets of actions $A_1; \ldots; A_k \in \mathcal{L}_{PD}^{act}{}^{+}$ and conditional plans $c_1, \ldots, c_l$, with $1 \leq l \leq |\mathcal{P}(\mathcal{L}_{PD}^{fl})|$, then*

$$A_1; \ldots; A_k; \mathit{case} \begin{cases} o_1 \ \rightarrow \ c_1 \\ \quad \ldots \\ o_l \ \rightarrow \ c_l \end{cases} \tag{2}$$

   is a conditional plan, where $o_i \in \mathcal{P}(\mathcal{L}_{PD}^{fl})$ and $o_i \neq o_j$ whenever $i \neq j$, i.e. each element of $\mathcal{P}(\mathcal{L}_{PD}^{fl})$ is associated to at most one c_i.
3. *Nothing else is a conditional plan.*

Intuitively, a conditional plan of the form[6] (2) in the above definition has to be read as "execute the actions in A_1, then the ones in A_2, ..., then the ones in A_k; and depending on which set of fluent literals that are true after executing these actions, execute the corresponding plan c_i".

The plan we introduced in the introduction, i.e. (1) on page 215, is a conditional plan for our running defusing a bomb example. Although in general, a conditional planner should consider all possible sets of fluent literals in a case construct, in practice this is not always necessary as certain sets cannot occur, given the current state and the actions performed. E.g., the external function *look_effect* will never return both *locked* and *unlocked* at the same time.

Definition 8. *For a planning domain PD, a **state** is any consistent subset $s \subseteq \mathcal{L}_{PD}^{fl}$.*

For each external function $p \in \sigma^{ec}$, we will use, with $t_1, \ldots, t_m \in \sigma^{con} \cup \sigma^{fl}$,

$$out(\&p[t_1, \ldots, t_m]) = \{(o_1, \ldots, o_n) \mid \&p[t_1, \ldots, t_m](o_1, \ldots, o_n) \in \mathcal{L}_{PD}^{ec}{}^{+}\} \ ,$$

i.e. the set containing all possible output tuples for a given input tuple. As not all input tuples are valid for the legal instantations of p, we will use

$$vit(\&p) = \{(t_1, \ldots, t_m) \mid \&p[t_1, \ldots, t_m](o_1, \ldots, o_n) \in \mathcal{L}_{PD}^{ec}{}^{+}\} \ .$$

Further, we associate with p an $(m+1)$-ary function $f_{\&p}$ that associates with each tuple $(s, t_1, \ldots, t_m)$ an element of $\mathcal{P}(\mathcal{P}(out(\&p[t_1, \ldots, t_m])))$, where s is a state and

[5] Although e.g. [12,14] introduce constructs as if-then-else or while-do in conditional plans, the former can be easily transformed to case-endcase statements, while the same holds for the latter in case one is interested in plans of bounded length.

[6] For practical purposes, multiple o_i (having the same c_i) might be compactly represented by a Boolean combination F of fluent literals using the connectives $\wedge$, $\vee$ and *not* , which is evaluated on a state s in the obvious way. A state s would correspond to the conjunction $\bigwedge_{l \in s} l \wedge \bigwedge_{l \in \mathcal{L}_{PD}^{fl} \setminus s}$ *not* l.

$(t_1, \ldots, t_m) \in vit(\&p)$. Intuitively, the function $f_{\&p}$ returns, for an external function $p \in \sigma^{ec}$ and an input tuple $(t_1, \ldots, t_m)$ w.r.t. a state s, the combinations of output tuples that are possible as a return value when the function is executed. Clearly, when $|f_{\&p}(s, t_1, \ldots, t_m)| = 1$, the external function is deterministic, otherwise it is non-deterministic.

To handle the external functions correctly in a state transition, we need to take care that each external function is evaluated exactly once for each possible input tuple. If not, we would have undesired results in case of non-deterministic functions, e.g. having two different rules with the same external function evaluating to different sets of output tuples. For this reason, we define an *external function evaluation* w.r.t. a state s as a function g_s, such that for each $p \in \sigma^{ec}$ and for each $(t_1, \ldots, t_m) \in vit(\&p)$, $g_s(\&p[t_1, \ldots, t_m]) = o$, where $o \in f_{\&p}(s, t_1, \ldots, t_m)$. Thus, each external function evaluates in g_s to exactly one set of output tuples for each possible input w.r.t. a state s.

Definition 9. *Let PD be a planning domain. A **state transition** is a tuple*

$$t = \langle s, g_s, A, s', g_{s'} \rangle \ ,$$

where s, s' are states, $g_s, g_{s'}$ are external function evaluations w.r.t. s (resp. s') and A is a set of action atoms.

Similar to the answer set semantics[7], we define our semantics first for positive planning domains, i.e. planning domains that are free from negation as failure. Afterwards, we will define a reduction from a general planning domain to a positive one. In what follows, we consider a ground planning domain $PD = (\Pi, (D, R))$, where M the unique answer set of Π.

For a set of ground literals $X \subseteq \mathcal{L}_{fl,typ} \cup \mathcal{L}_{act}{}^{+}$, a ground literal $l \in \mathcal{L}_{fl,typ} \cup \mathcal{L}_{act}{}^{+}$ and an external function evaluation g_s, with $s = X \cap \mathcal{L}_{PD}^{fl}$, we use

- $X \models_{g_s} l$, when $l \in X$;
- $X \models_{g_s}$ *not* l, when $l \notin X$;
- $X \models_{g_s} \&p[t_1, \ldots, t_m](o_1, \ldots, o_m)$, when $(o_1, \ldots, o_m) \in g_s(\&p[t_1, \ldots, t_n])$; and
- $X \models_{g_s}$ *not* $\&p[t_1, \ldots, t_m](o_1, \ldots, o_m)$, when $(o_1, \ldots, o_m) \notin g_s(\&p[t_1, \ldots, t_n])$.

Finally, for a set of ground literals $Y \subseteq \mathcal{L}$, we use $X \models_{g_s} Y$ iff $X \models_{g_s} y$ for each $y \in Y$. As usual, we have $X \not\models_{g_s} Y$ if we have not $X \models_{g_s} Y$.

Definition 10. *For a positive PD, a state s_0 and an external function evaluation g_{s_0}, we call s_0 a **legal initial state** if s_0 is the smallest (w.r.t. subset inclusion) set such that $h(r) \in s_0$ whenever $s_0 \cup M \models_{g_{s_0}} post(r)$ for all initial state constraints and static rules $r \in R$.*

For a positive PD, a state s and an external function evaluation g_s, a set $A \subseteq \mathcal{L}_{PD}^{act}{}^{+}$ is called an *executable action set* w.r.t. s and g_s, if for each $a \in A$ there exists an executability condition $e \in R$ such that $h(e) = a$ and $s \cup M \cup A \models_{g_s} pre(e)$.[7]

[7] Note that we allow dependent actions, i.e. actions that depend on the execution of other actions.

Definition 11. *Let PD be a positive planning domain, let $t = \langle s, g_s, A, s', g_{s'}\rangle$ be a state transition and let $r \in R$ be a causation rule. We say that r is **satisfied** by s' w.r.t. t iff either $h(r) \subseteq s' \setminus \{false\}$ or we do not have both $s' \cup M \models_{g_{s'}} post(r)$ and $s \cup M \cup A \models_{g_s} pre(r)$.*

*A state transition $t = \langle s, g_s, A, s', g_{s'}\rangle$ is called a **legal state transition** if A is an executable action set w.r.t. s and g_s, and s' is a minimal (w.r.t. subset inclusion) consistent set that satisfies all causation rules in R, except initial state constraints, w.r.t. t.*

Next, we generalize the above to arbitrary ground planning domains PD, i.e. planning domains containing negation as failure in the rules. This is done by defining a reduction to positive planning domains, similar to the GL-reduct for the answer set semantics [7].

Definition 12. *Let PD be an arbitrary planning domain and consider a state transition $t = \langle s, g_s, A, s', g_{s'}\rangle$. The **reduction** of PD w.r.t. t, denoted PD^t, is defined by $PD^t = (\Pi, (D, R^t))$, where R^t is obtained from R by removing:*

- *every $r \in R$ for which either $s' \cup M \not\models_{g_{s'}} post^-(r)$ or $s \cup A \cup M \not\models_{g_s} pre^-(r)$;*
- *all literals not l, with $l \in \mathcal{L}$, from the remaining rules.*

Clearly, PD^t is a positive planning domain. Now we can define the concepts of legal initial states, executable action sets and legal state transitions in the case of arbitrary planning domains.

Definition 13. *Let PD be a planning domain. A state s_0 is a **legal initial state** if s_0 is a legal initial state for PD^t, where $t = \langle \emptyset, g_\emptyset, \emptyset, s_0, g_{s_0}\rangle$. A set A is an **executable action set** w.r.t. a state s, if A is an executable action set w.r.t. s in PD^t, where $t = \langle s, g_s, A, \emptyset, g_\emptyset\rangle$. A state transition $t = \langle s, g_s, A, s', g_{s'}\rangle$ is a **legal state transition** if it is a legal state transition for PD^t.*

Before we can define optimistic conditional plans, we need some additional notions. For a planning domain PD and two states s_0 and s_n, with $n \geq 0$, a sequence of state transitions

$$T = \langle\langle s_0, g_{s_0}, A_1, s_1, g_{s_1}\rangle, \langle s_1, g'_{s_1}, A_2, s_2, g_{s_2}\rangle, \ldots, \langle s_{n-1}, g'_{s_{n-1}}, A_n, s_n, g_{s_n}\rangle\rangle$$

is called a *trajectory* from s_0 to s_n for PD if all state transitions in T are legal. Further, we use $\mathcal{T}(s_0, s_n)$ to denote the set of all trajectories from s_0 to s_n; and for a sequence of sets of actions $A_1; \ldots; A_k$ and a set of states S, we use

$$PD(A_1; \ldots; A_k, S) = \{s_k \mid \langle\langle s_0, g_{s_0}, A_1, s_1, g_{s_1}\rangle, \langle s_1, g'_{s_1}, A_2, s_2, g_{s_2}\rangle, \ldots, \langle s_{k-1}, g'_{s_{k-1}}, A_k, s_k, g_{s_k}\rangle\rangle \in \mathcal{T}(s_0, s_k) \wedge s_0 \in S\} \ .$$

Now, we have all necessary means to define optimistic conditional plans.

Definition 14. *Let $PP = (PD, q)$ be a planning problem, let C be a conditional plan and let S be a set of states. We define C being **optimistic** w.r.t. S inductively as*

1. *if $C = A_1; \ldots; A_k$ and $\exists s \in S \cdot \exists x \in PD(C, \{s\}) \cdot \{g_{m+1}, \ldots, g_n\} \cap x = \emptyset \wedge \{g_1, \ldots, g_m\} \subseteq x$, then C is optimistic w.r.t. S.*

2. *if* $C = A_1; \ldots; A_k;$ *case* $\left\{ \begin{array}{l} o_1 \rightarrow c_1 \\ \quad \ldots \quad \text{; and } \{o_1, \ldots o_l\} \cap PD(A_1; \ldots; A_k, S) \neq \emptyset; \\ o_l \rightarrow c_l \end{array} \right.$
and c_i *is optimistic w.r.t.* o_i *for each* $i \in [1 \ldots l]$*, then* C *is optimistic w.r.t.* S*.*

Now, C *is an* ***optimistic plan*** *for* PP *if it is optimistic w.r.t. the set of all legal initial states.*

Intuitively, condition (1) in the above definition demands that a goal state can be reached for each starting state in S, while condition (2) demands that each of the conditional states o_i can be reached, starting from the states in S, and that for each of these conditional states o_i a goal state can be reached by executing c_i.

Note that an optimistic conditional plan corresponds to an optimistic plan in $\mathcal{K}$ when the conditional plan does not contain case constructs. This implies that executing an optimistic conditional plan can yield situations where the goal is not reached. Similar to $\mathcal{K}$, we can define when a conditional plan is secure, i.e. when executing the conditional plan will always result in a goal state.

Definition 15. *Let* $PP = (PD, q)$ *be a planning problem, let* C *be a conditional plan and let* S *be a set of states. The* ***secureness*** *of* C *w.r.t.* S *is inductively defined as*

1. *if* $C = A_1; \ldots; A_k$ *and* $\forall s \in S \cdot \forall i \in [1 \ldots k] \cdot \forall s' \in PD(A_1; \ldots; A_{i-1}, \{s\}) \cdot PD(A_i, \{s'\}) \neq \emptyset$ *and* $\forall s \in S \cdot \forall x \in PD(C, \{s\}) \cdot \{g_{m+1}, \ldots, g_n\} \cap x = \emptyset \wedge \{g_1, \ldots, g_m\} \subseteq x$*, then* C *is secure w.r.t.* S*.*
2. *if* $C = A_1; \ldots; A_k;$ *case* $\left\{ \begin{array}{l} o_1 \rightarrow c_1 \\ \quad \ldots \quad \text{; and } PD(A_1; \ldots; A_k, S) \subseteq \{o_1, \ldots, o_n\} \\ o_l \rightarrow c_l \end{array} \right.$
and $\forall s \in S \cdot \forall i \in [1 \ldots k] \cdot \forall s' \in PD(A_1; \ldots; A_{i-1}, \{s\}) \cdot PD(A_i, \{s'\}) \neq \emptyset$ *and* c_i *is secure w.r.t.* o_i *for each* $i \in [1 \ldots l]$*, then* C *is secure w.r.t.* S*.*

Now, C *is a* ***secure plan*** *for* PP *if it is secure w.r.t. the set of all legal initial states.*

Intuitively, the condition $\forall s \in S \cdot \forall i \in [1 \ldots k] \cdot \forall s' \in PD(A_1; \ldots; A_{i-1}, \{s\}) \cdot PD(A_i, \{s'\}) \neq \emptyset$ in the above definition ensures that a secure plan never gets "stuck" in a state during execution.

The conditional plan (1) on page 215 is a secure plan for our defusing a bomb example. However, if we change the behavior of the external function $look\_effect$ such that it either returns $locked$, $unlocked$ or neither $locked$ nor $unlocked$, than one can check that the conditional plan is not any longer secure. Furthermore, it turns out that no secure plan exists for this modification.

On the other hand, consider the following extension of the defusing a bomb example. We can look at the bomb and if the light is on, we know that the bomb is either *locked* or *unlocked*, but if we do not have light (or we don't know if there is light or not) looking at the bomb can either yield $locked$, $unlocked$ or neither $locked$ nor $unlocked$. Further, we have an action *check_light* with a corresponding external function $check\_light\_effect$ which materializes the effect of the "sensing" action *check_light*. Finally, using the action *switch* we can switch the state from the light. Now[8] one can see that the following

[8] The encoding in $\mathcal{K}^c$ of this example can be found in Appendix D of [17].

conditional plan

$$check\_light; case \begin{cases} \{light\} \rightarrow look; case \begin{cases} \{locked\} \rightarrow disarm \\ \{unlocked\} \rightarrow turn; disarm \end{cases} \\ \{no\_light\} \rightarrow switch; look; case \begin{cases} \{locked\} \rightarrow disarm \\ \{unlocked\} \rightarrow turn; disarm \end{cases} \end{cases}$$

is secure, while the conditional plan on page 215 is only an optimistic one.

4 Computing Conditional Plans Using $DLV^{\mathcal{K}}$

To demonstrate our conditional planning framework, we developed a prototype implementation of a part of the semantics of $\mathcal{K}^c$ on top of DLV$^{\mathcal{K}}$. As DLV$^{\mathcal{K}}$ does not support external evaluations, our prototype currently disregards[9] this feature of $\mathcal{K}^c$. Further, we only implemented optimistic plan generation so far, but in the next step a secure checker will be added, using the built-in security checker of DLV$^{\mathcal{K}}$. When provided with a classical $\mathcal{K}$ planning problem description, DLV$^{\mathcal{K}}_c$ will generate a conditional plan in a graphical representation. E.g., feeding the defusing a bomb planning description from the introduction (page 2) to the system, will yield the conditional plan depicted in Figure 2. To generate this plan, we use DLV$^{\mathcal{K}}$'s batch mode for generating optimistic plans. Each optimistic plan received from DLV$^{\mathcal{K}}$ is first compressed by removing useless planning steps, and afterwards this compressed optimistic plan is put into the graph representing the optimistic conditional plan, i.e. each optimistic plan from DLV$^{\mathcal{K}}$ can be seen as a valid trajectory in an optimistic conditional plan.

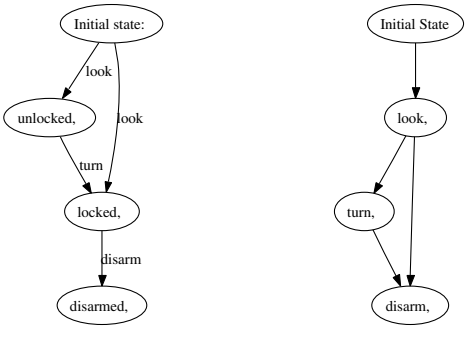

Fig. 2. The generated conditional plan for the defusing a bomb example from page 2. On the left, we have a conditional plan that also contains the states reached, while the plan on the right only contains the different actions that need to be taken to reach the goal. To increase readability, our implementation compacts the tree shape of a conditional plan into a dag whenever possible.

The prototype implementation is built using the Python programming language and can be run on any modern platform that DLV$^{\mathcal{K}}$ supports. The implementation, together

[9] One can of course, naïvely, introduce the behavior of external evaluations by adding mutually exclusive rules that introduce the possible outcomes of the external evaluations hard-coded.

with additional information and examples, can be found at `http://tinfpc2.vub.ac.be/cdlvk`.

5 Related Work and Conclusion

In this paper we presented $\mathcal{K}^c$, a conditional planning language that can use external functions to outsource the computation of certain effects when an action is executed. As $\mathcal{K}^c$ is a proper extension of the planning language $\mathcal{K}$, it relates to most other planning language in the same way, and we therefore refer to [5]. One exception here, are extensions of those languages that incorporate sensing actions to obtain non-deterministic, i.e. conditional, planning. For these extensions, e.g. [15,16], we clearly showed that external function calls are well-suited to simulate sensing actions by combining an ordinary action with an external function that materializes the effects of the "sensing" action.

In future work, we plan to extend our prototype so the generated conditional plans can be checked for secureness. We also want to incorporate external functions natively, such that the explicit introduction of such functions by using mutually exclusive rules can be dropped, improving the readability and robustness of the planning descriptions. Finally, we are currently employing our framework in the context of repairing web service workflows by planning [1], one of the topics of the ongoing WS-Diamond research project [2]. The high expressiveness and declarativity of $\mathcal{K}^c$, together with its conditional planning capabilities, turns out to be beneficial in that area of application.

References

1. Private Communications with Gerhard Friedrich, University of Klagenfurt, Austria.
2. Ws-diamond: Web-service diagnosability, monitoring & diagnosis (ist-516933). Project website at http://wsdiamond.di.unito.it.
3. Special issue on reasoning about action and change. *Journal of Logic Prog.*, 31(1-3), 1997.
4. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under incomplete knowledge. In *Computational Logic*, volume 1861 of *LNCS*, pages 807–821. Springer, 2000.
5. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *Transactions on Computational Logic*, 5(2):206–263, 2004.
6. T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 90–96, 2005.
7. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3-4):365–386, 1991.
8. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(2/3&4):301–321, 1993.
9. E. Giunchiglia, G. N. Kartha, and V. Lifschitz. Representing action: Indeterminacy and ramifications. *Artificial Intelligence*, 95(2):409–438, 1997.
10. K. Golden and D. Weld. Representing sensing actions: The middle ground revisited. In *Proc. of the 5th Intl. Conf. on Principles of KR and Reasoning*, pages 174–185, 1996.

11. L. Iocchi, D. Nardi, and R. Rosati. Planning with sensing, concurrency, and exogenous events: logical framework and implementation. In *Proc. of the 7th Intl. Conference on Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 678–689, 2000.
12. H. J. Levesque. What is planning in the presence of sensing? In *AAAI/IAAI, Vol. 2*, pages 1139–1146, 1996.
13. V. Lifschitz. *The Logic Programming Paradigm - A 25-Year Perspective*. Springer, 1999.
14. J. Lobo, S. Taylor, and G. Mendez. Adding knowledge to the action description language A. In *Proc. of the 14th National Conf. on AI (AAAI97)*, pages 454–459. AAAI Press, 1997.
15. T. C. Son and C. Baral. Formalizing sensing actions a transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, 2001.
16. T. C. Son, P. H. Tu, and C. Baral. Planning with sensing actions and incomplete information using logic programming. In *Proc. of the 7th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004)*, volume 2923 of *LNAI*, pages 261–274, 2004.
17. D. Van Nieuwenborgh, T. Eiter, and D. Vermeir. Conditional planning with external functions. Technical report, 2007, http://tinf2.vub.ac.be/˜dvnieuwe/lpnmr2007technical.ps.