Relaxing non-interference requirements in parallel plans

MIQUEL BOFILL*, *Departament d'Informàtica, Matemàtica Aplicada i Estadística Universitat de Girona, E-17003 Girona, Spain.*

JOAN ESPASA**, *School of Computer Science, University of St Andrews, St Andrews KY16 9SX, UK.*

MATEU VILLARET†, *Departament d'Informàtica, Matemàtica Aplicada i Estadística Universitat de Girona, E-17003 Girona, Spain.*

Abstract

The aim of being able to reason about quantities, time or space has been the main objective of the many efforts on the integration of propositional planning with extensions to handle different theories. Planning modulo theories (PMTs) are an approximation inspired by satisfiability modulo theories (SMTs) that generalize the integration of arbitrary theories with propositional planning. Parallel plans are crucial to reduce plan lengths and hence the time needed to reach a feasible plan in many approaches. Parallelization of actions relies on the notion of (non-)interference, which is usually determined syntactically at compile time. In this paper we define a semantic notion of interference between actions in PMT. Apart from being strictly stronger than any syntactic notion of interference, we show how semantic interference can be easily and efficiently checked by calling an off-the-shelf SMT solver at compile time, constituting a technique orthogonal to the solving method.

Keywords: Planning, planning modulo theories, SMT, planning as satisfiability, parallel plans.

1 Introduction

The problem of planning, in its most basic form, consists in finding a sequence of actions that allows one to reach a goal state from a given initial state. In its classical approach, state variables are propositional, but in many real world problems we need to reason about concepts such as time, space, capacities, etc., that are impractical to represent using only propositional variables. Many planners have been built to be able to deal with such problems. These solvers glue together a classical planner with a specific theory solver that exclusively handles the non-propositional part. Planning modulo theories (PMTs) (8) are a modular framework inspired in the architecture of lazy satisfiability modulo theory (SMT), which is the natural extension of Boolean Satisfiability (SAT) when propositional formulas need to be combined with other theories.

The possibility of several actions being planned at the same time step, i.e. the notion of parallel plans, can be considered a key idea to reduce plan lengths and hence the time needed to reach a feasible plan in many approaches. Parallel plans increase the efficiency not only because they allow

*E-mail: mbofill@imae.udg.edu
**E-mail: jea20@st-andrews.ac.uk
†E-mail: villaret@imae.udg.edu

doi:10.1093/jigpal/jzz026

to reduce the time horizon, but also because it is unnecessary to consider all total orderings of the actions that are performed in parallel when seeking for a plan.

A problem that arises when considering parallel plans is that of interference between actions. It is commonly accepted that it should be possible to serialize every parallel plan while preserving its semantics. Hence, it is usually required that effects of actions planned at the same time commute and that there is no interaction between preconditions and effects of different actions. A set of actions planned at the same time is called a *happening* (6). Existing works on numeric planning use syntactic or limited semantic approaches to determine interference between actions, in a fairly restrictive way (6; 7; 11). In this work we aim at relaxing unnecessarily strong conditions for non-interference. The main contributions are

- a new relaxed notion of *happening execution* and
- a semantic notion of interference in the context of PMT.

We also prove their correctness, motivate their usefulness with some examples and show how they can be easily implemented. This paper is an extension of the work (2) presented at the ICAPS 2016 conference. We extend it with a precise description of how interference detection is implemented and by adding detailed experiments.

The rest of the paper proceeds as follows. In Section 2 we recall PMT. In Section 3 we introduce our new semantic notion of interference between actions. In Section 4 we describe how interference can be checked by using an SMT solver at compile time. In Section 5 we propose some encodings for parallel plans in a planning as SMT approach. Section 6 is devoted to experimental evaluation. Section 7 concludes. An appendix is included with a Planning Domain Definition Language (PDDL) model of an introduced planning domain, detailed experiments and proofs of auxiliary lemmas.

2 Planning modulo theories

We follow the concepts and notation defined in (8) for PMTs. The key to extending classical planning into PMT is to support first-order sentences modulo theories in the preconditions of actions.

A *state* is a valuation over a finite set of variables X, i.e. an assignment function, mapping each variable $x \in X$ to a value in its domain, D_x. The expression $s[x]$ denotes the value state s assigns to variable x, and $s[x \mapsto v]$ is the state identical to s except that it assigns the value v to variable x. A *state space* for a set of variables X is the set of all valuations over X. By $var(S)$ we denote the state variables of a state space S.

A *first-order sentence over a state space S modulo T* is a first-order sentence over the variables of the state space, constant symbols, function symbols and predicate symbols, where T is a theory defining the domains of the state space variables and interpretations for the constants, functions and predicates.[1]

A *state space modulo T* is a state space ranging over the domains defined in T. A *term* over S modulo T is, similarly, an expression constructed using the symbols defined by S and T. A formula ϕ is T-satisfiable if $\phi \wedge T$ is satisfiable in the first-order sense. By $eval^s_T(\phi)$ we denote the value of ϕ under the assignment s, according to the interpretation defined by theory T.

A *substitution* is a partial mapping from variables to terms. It can be represented explicitly as a function by a set of bindings of variables to terms. That is if $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, then $\sigma(x_i) = t_i$ for all i in $1..n$, and $\sigma(x) = x$ for every other variable.

[1] In some other contexts, such as mathematical logic, a theory is understood as being just a set of sentences.

Substitutions are extended homomorphically to a total mapping from terms to terms. We use the postfix notation $t\sigma$ for the image of a term t under a substitution σ. This is defined inductively on the structure of terms as follows:

$$t\sigma = \begin{cases} \sigma(t) & \text{if } t \text{ is a variable} \\ f(t_1\sigma, \ldots, t_m\sigma) & \text{if } t \text{ is of the form } f(t_1, \ldots, t_m). \end{cases}$$

In the second case of this definition, $m = 0$ is allowed: in this case, f is a constant symbol and $f\sigma$ is f. Thus $t\sigma$ is t with all variables replaced by terms as specified by σ. The image of a formula under a substitution is defined similarly.

The composition of two substitutions σ_1 and σ_2, denoted by juxtaposition, is defined as the composition of two functions, i.e. $t\sigma_1\sigma_2 = (t\sigma_1)\sigma_2$.

DEFINITION 2.1 (Action).
An *action* a, for a state space S modulo T, is a state transition function, comprising the following:

- A first-order sentence over S modulo T, Pre_a (the *precondition* of a).
- A set Eff_a (the *effects* of a), of assignments to a subset of the state variables in S, each setting a distinct variable to a value defined by a term over S modulo T.

An action a, for a state space S modulo T, is *applicable* (or *executable*) in a state $s \in S$ if $T, s \models Pre_a$ (i.e., the theory together with the valuation s satisfies the precondition of a).

We represent actions a as pairs $\langle Pre_a, Eff_a \rangle$, with the effects Eff_a often written as a substitution $\sigma_a = \{x_1 \mapsto exp_1, \ldots, x_n \mapsto exp_n\}$, where exp_i is an expression that defines the value of variable x_i in the resulting state, for each i in $1..n$ (e.g. $x \mapsto x + k$, for increasing a numeric variable x by k). We use $\top$ and $\bot$ to denote the Boolean *true* and *false* values, respectively. Making abuse of notation, we will talk of a substitution as an assignment.

Following application of a, the state is updated by the assignments in Eff_a to the variables that they affect, leaving all other variables unchanged. We denote the unique state resulting from applying action a, in a state s in which is applicable, by $app_a(s)$. Formally, the resulting state s' is the mapping where, for each variable $x \in var(S)$, $s'(x) = eval^s_T(x\sigma_a)$, where σ_a is the substitution representing the effects of a.

For sequences $a_1; a_2; \ldots; a_n$ of actions we define

$$app_{a_1;a_2;\ldots;a_n}(s) = app_{a_n}(\cdots app_{a_2}(app_{a_1}(s)) \cdots).$$

DEFINITION 2.2 (Planning modulo theory).
A *Planning Modulo T* problem, for a theory T, is a tuple $\pi = \langle S, A, I, G \rangle$ where

- S is a state space in which all of the variable domains are defined in T,
- A is a set of actions for S modulo T,
- I is a valuation in S (the *initial state*) and
- G is a first-order sentence over S modulo T (the *goal*).

A (sequential) *plan* for π is a sequence of actions $a_1; \ldots; a_n$ such that, for all i in $1..n$, a_i is applicable in state s_{i-1} and s_i is the result of applying a_i to s_{i-1}, where $s_0 = I$ and $T, s_n \models G$.

As it is usual in SMT, we assume T is a first-order theory with equality, which means that the equality symbol $=$ is a predefined predicate, interpreted as the identity on the underlying domain.

Sometimes we say that a sequence of actions is a plan starting from an initial state I, without specifying the goal. In this case we mean that the plan is executable starting from I.

The ILP-PLAN framework (11), based on integer optimization of linear integer programs, is a particular case of this, taking linear inequalities as preconditions, and limiting effects to increasing, decreasing or setting the value of a variable. Numeric planning, as defined in (9) or in (7) is also a particular case, using a very limited fragment of first-order logic in the preconditions of actions, and taking T as the theory of rational functions, i.e. fractions between polynomials. The proposal in (14) raises preconditions to general Boolean formulae but does not consider numeric variables.

3 A semantic notion of interference

In the following, we consider plans as sequences of *sets of actions*. A set of actions planned at the same time is commonly called a *happening* (6). Two actions can be concurrently planned if, roughly, they do not interfere. It is commonly accepted that two actions are non-interfering only if the composition of their effects is commutative, and there is no interaction between effects and preconditions. In (6; 7) the state resulting from executing a happening is defined as the one obtained after applying the composition of effects of the actions in the happening.

EXAMPLE 3.1
Let $a = \langle \top, \{x \mapsto x + y + z\}\rangle$ and $b = \langle \top, \{x \mapsto x + 1, y \mapsto y + 1, z \mapsto z - 1\}\rangle$. These actions do not interfere, as their preconditions are true (and hence cannot interact with effects) and their effects commute: executing first a and then b, as well as executing first b and then a, produces the same effect, which is that of an action of the form $\langle \top, \{x \mapsto x+y+z+1, y \mapsto y+1, z \mapsto z-1\}\rangle$, defining the state transition function of their simultaneous execution.

EXAMPLE 3.2
Let $c = \langle \top, \{x \mapsto x + y + z\}\rangle$ and $d = \langle \top, \{x \mapsto x + 1, y \mapsto y + 2, z \mapsto z - 1\}\rangle$. These actions interfere, since their effects do not commute. Executing first c and then d is equivalent to executing $\langle \top, \{x \mapsto x + y + z + 1, y \mapsto y + 2, z \mapsto z - 1\}$, whereas executing first d and then c is equivalent to executing $\langle \top, \{x \mapsto x + y + z + 2, y \mapsto y + 2, z \mapsto z - 1\}$. Then they would not be allowed to be planned in parallel.

Thanks to the commutativity requirement, effects of non-interfering actions can be composed in any order, allowing parallel plans to be serialized in any order, while preserving their semantics. This adheres to the $\forall$-step semantics of (14), but it does not lift to the $\exists$-step semantics (introduced in the same work for the Boolean case), where it is only necessary that actions can be executed it at least one order, making it possible to increase the number of parallel actions.

In the rest of this section we introduce a new semantics of happening execution, define a new notion of interference and prove that their combination is valid for both $\forall$-step and $\exists$-step semantics.

DEFINITION 3.3 (Commuting assignments).
Two assignments $\{x \mapsto exp_1\}$ and $\{x \mapsto exp_2\}$ *commute*, for a variable x and two expressions (terms) exp_1 and exp_2 over a state space S modulo T, if $T \models (exp_2\{x \mapsto exp_1\} = exp_1\{x \mapsto exp_2\})$.

EXAMPLE 3.4
If T is the theory of the reals, then $\{x \mapsto x+1\}$ and $\{x \mapsto x-2\}$ commute, since $T \models ((x-2)+1 = (x+1)-2)$, whereas $\{x \mapsto x+1\}$ and $\{x \mapsto x*2\}$ do not commute, since $T \not\models ((x+1)*2 = (x*2)+1)$.

DEFINITION 3.5 (Simply commuting actions).
We will refer to a set $A = \{a_1, \ldots, a_n\}$ of actions as *simply commuting*, for a state space S modulo T, if for every variable $x \in var(S)$ and every pair of assignments $\{x \mapsto exp_1\}$ and $\{x \mapsto exp_2\}$ in the effects of actions in A, $\{x \mapsto exp_1\}$ and $\{x \mapsto exp_2\}$ commute.

DEFINITION 3.6 (Happening action).
Let $A = \{a_1, \ldots, a_n\}$ be a set of simply commuting actions. We define the *happening action* for A as an action $h(A) = \langle Pre_{h(A)}, \sigma_{h(A)} \rangle$ with

$$Pre_{h(A)} = \bigwedge_{a \in A} Pre_a$$

and

$$\sigma_{h(A)} = \bigcup_{x \in var(S)} \{\sigma_{x,1} \circ \cdots \circ \sigma_{x,n}\},$$

where $\sigma_{x,i}$, for i in $1..n$, is the mapping of variable x in the effects of action a_i, and $\circ$ denotes the composition of functions.

Note that the effects on each variable can be composed in any order, because of the commutation requirement. Therefore, $h(A)$ is well defined.

DEFINITION 3.7 (Happening execution).
Let $A = \{a_1, \ldots, a_n\}$ be a set of simply commuting actions. Then, the state resulting from the execution of the happening A in state s, denoted $app_A(s)$, is defined as $app_{h(A)}(s)$, where $h(A)$ is the happening action corresponding to A.

Note that if some action in A is not applicable in state s then $app_A(s)$ is undefined.

EXAMPLE 3.8
Let $a = \langle \top, \{x \mapsto x+1, y \mapsto y+1\} \rangle$ and $b = \langle \top, \{y \mapsto y+x\} \rangle$. Then $app_{\{a,b\}}(s)$, for a state s, is $app_{h(\{a,b\})}(s)$, with $h(\{a,b\}) = \langle \top, \{x \mapsto x+1, y \mapsto (y+x)+1\} \rangle$.

A key difference with the transition functions for happenings defined in (6) and (7) is that, instead of considering the *composition of functions* (i.e. the composition of effects of actions, seen as functions on all variables), we are considering the *function of compositions* (i.e. the function defined by the composition of assignments to each single variable across all actions). We consider the possibility of composing the effects on each variable in any order, as a minimal requirement to be able to serialize plans in some order (see definitions and proofs below). The proposed definition of happening execution allows us to increase the number of parallel actions in the context of $\exists$-step plans, where parallel semantics and interference notions of existing approaches to numeric planning are too restrictive. Following this line we define a relaxed semantic notion of interference and prove that it is compatible with both $\forall$-step and $\exists$-step semantics.

DEFINITION 3.9 (Affecting action).
Given two actions $a = \langle Pre_a, \sigma_a \rangle$ and $b = \langle Pre_b, \sigma_b \rangle$, for a state space S modulo T, we consider a to *affect* b if

1. $Pre_a \wedge Pre_b \wedge \neg(Pre_b\sigma_a)$ is T-satisfiable or
2. either a and b are not simply commuting, or $Pre_a \wedge Pre_b \wedge \neg(x\sigma_{h(\{a,b\})} = x\sigma_b\sigma_a)$ is T-satisfiable for some variable $x \in var(S)$, where $h(\{a, b\})$ denotes the happening action for a and b,

i.e. a can impede the execution of b, or they are not simply commuting, or they are simply commuting but executing first a and then b has a different effect than that of the happening $\{a, b\}$.

Recall that $h(\{a, b\})$ is defined only for simply commuting actions.

EXAMPLE 3.10
Following Example 3.8, where actions a and b are simply commuting, we have that a affects b since $y\sigma_{h(\{a,b\})} = (y + x) + 1$, while $y\sigma_b\sigma_a = (y + x)\sigma_a = (y + 1) + (x + 1)$, and thus $Pre_a \wedge Pre_b \wedge \neg(y\sigma_{h(\{a,b\})} = y\sigma_b\sigma_a)$ is T-satisfiable. On the contrary, b does not affect a, since the preconditions of both actions are true, $x\sigma_{h(\{a,b\})} = x + 1 = x\sigma_b\sigma_a$, and $y\sigma_a\sigma_b = (y + 1)\sigma_b = (y + x) + 1$. This is to say that the effect of the happening $\{a, b\}$ is the same as executing first b and then a, but not first a and then b. In fact, in this example we have $app_{\{a,b\}}(s) = app_{b;a}(s) \neq app_{a;b}(s)$ for all s.

DEFINITION 3.11 (Interference).
Given two actions a and b, we consider a and b to *interfere* if a affects b or b affects a.

3.1 ∀-Step plans

Lack of interference guarantees that actions in a happening can be executed sequentially in any total order and that the final state is independent of the ordering (see Theorem 3.17). The notion of ∀-step plan, defined in (14), can be generalized to the setting of PMT as follows.

DEFINITION 3.12 (∀-Step plan).
Given a set of actions A and an initial state I, for a state space S modulo T, a ∀-*step plan* for A and I is a sequence $P = \langle A_0, \ldots, A_{l-1} \rangle$ of sets of actions for some $l \geq 0$, such that there is a sequence of states $s_0, \ldots, s_l$ (the execution of P) such that

1. $s_0 = I$, and
2. for all $i \in \{0, \ldots, l - 1\}$ and every total ordering $a_1 < \cdots < a_n$ of A_i, $app_{a_1;\ldots;a_n}(s_i)$ is defined and equals s_{i+1}.

LEMMA 3.13
Let a and b be two simply commuting actions, for a state space S modulo T, such that a does not affect b, and let $s \in S$ be a state such that a and b are applicable in s. Then $app_{\{a,b\}}(s) = app_{a;b}(s)$.

LEMMA 3.14
Let A be a set of non-interfering actions, for a state space S modulo T, and let $s \in S$ be a state such that all actions in A are applicable in s. Then $app_{a_1;\ldots;a_n}(s)$ is the same state for every total ordering $a_1 < \cdots < a_n$ of A.

LEMMA 3.15
Let A be a set of simply commuting actions, for a state space S modulo T, such that $|A| \geq 2$. Then, for every action $a \in A$, we have that a and $h(A\backslash\{a\})$ are simply commuting, and $h(\{a, h(A\backslash\{a\})\}) = h(A)$.

LEMMA 3.16
Let A be a set of actions, and a an action, for a state space S modulo T, such that the actions in $A \cup \{a\}$ are simply commuting. If a affects none of the actions in A, then a does not affect the happening action $h(A)$.

The reader is referred to the Appendix A for the proofs and extra lemmas.

THEOREM 3.17
Let A be a set of non-interfering actions, for a state space S modulo T, and $s \in S$ a state such that $app_A(s)$ is defined. Then $app_A(s) = app_{a_1;\ldots;a_n}(s)$ for any total ordering $a_1 < \cdots < a_n$ of A.

PROOF. By induction on the number of actions n in A. If $n = 1$ then we are trivially done. If $n \geq 2$, then let $A = \{a\} \cup A'$. Since actions in A are non-interfering, then they are simply commuting and a affects none of the actions in A'. Then, by Lemma 3.16, we have that a does not affect the happening action $h(A')$. Now observe that, since $app_A(s)$ is defined and $Pre_{h(A)} = \bigwedge_{a \in A} Pre_a$, both a and $h(A')$ are applicable in state s. Then, by Lemma 3.13, we have that $app_{\{a,h(A')\}}(s) = app_{a;h(A')}(s)$. We conclude by showing that $app_A(s) = app_{\{a,h(A')\}}(s)$ and $app_{a;h(A')}(s) = app_{a_1;\ldots;a_n}(s)$ for any total ordering $a_1 < \cdots < a_n$ of A.

Equality $app_A(s) = app_{\{a,h(A')\}}(s)$ holds by Lemma 3.15. For equality $app_{a;h(A')}(s) = app_{a_1;\ldots;a_n}(s)$, observe that $app_{a;h(A')}(s) = app_{A'}(app_a(s))$. Since actions in A' are non-interfering and $app_{A'}(app_a(s))$ is defined, by the induction hypothesis we have that $app_{A'}(app_a(s)) = app_{a_1;\ldots;a_{n-1}}(app_a(s))$ for any total ordering $a_1 < \cdots < a_{n-1}$ of A', i.e. $app_{a;h(A')}(s) = app_{a;a_1;\ldots;a_{n-1}}(s)$ for any total ordering $a_1 < \cdots < a_{n-1}$ of A'. Finally, since actions in A are non-interfering and all of them are applicable in state s, by Lemma 3.14 we have that $app_{a;h(A')}(s) = app_{a_1;\ldots;a_n}(s)$ for any total ordering $a_1 < \cdots < a_n$ of A □

3.2 ∃-Step plans

Here we generalize the notion of ∃-step plan, proposed in (4) and further developed in (14), to the setting of PMTs. Under the ∃-step semantics, it is not necessary that all actions are non-interfering as long as they can be executed it at least one order, which makes it possible increase the number of parallel actions still further.

DEFINITION 3.18 (∃-Step plan).
Given a set of actions A and an initial state I, for a state space S modulo T, a ∃-*step plan* for A and I is a sequence $P = \langle A_0, \ldots, A_{l-1} \rangle$ of sets of actions together with a sequence of states $s_0, \ldots, s_l$ (the execution of P), for some $l \geq 0$, such that

1. $s_0 = I$, and
2. for all $i \in \{0, \ldots, l-1\}$ there is a total ordering $a_1 < \cdots < a_n$ of A_i, such that $app_{a_1;\ldots;a_n}(s_i)$ is defined and equals s_{i+1}.

Instead of requiring that each group A_i of actions can be ordered to any total order, as in ∀-step semantics, in ∃-step semantics it is sufficient that there is one order that maps state s_i to s_{i+1}. Note that under this semantics the successor s_{i+1} of s_i is not uniquely determined solely by A_i, as the successor depends on the implicit ordering of A_i and, hence, the definition has to make the execution $s_0, \ldots, s_l$ explicit.

THEOREM 3.19
Let A be a set of simply commuting actions, for a state space S modulo T, such that, for some total ordering $a_1 < \cdots < a_n$ of A, if $a_i < a_j$ then a_i does not affect a_j, and let $s \in S$ be a state such that $app_A(s)$ is defined. Then $app_A(s) = app_{a_1;\ldots;a_n}(s)$.

PROOF. The proof is analogous to the proof of Theorem 3.17, but without using Lemma 3.14. We proceed by induction on the number of actions n in A. If $n = 1$ then we are trivially done. If $n \geq 2$, then let $A' = \{a_2, \ldots, a_n\}$. We have that actions in A are simply commuting and a_1 affects none of the actions in A'. Then, by Lemma 3.16, we have that a_1 does not affect the happening action $h(A')$. Now observe that, since $app_A(s)$ is defined and $Pre_{h(A)} = \bigwedge_{a \in A} Pre_a$, both a_1 and $h(A')$ are applicable in state s. Then, by Lemma 3.13, we have that $app_{\{a_1,h(A')\}}(s) = app_{a_1;h(A')}(s)$. We conclude by showing that $app_A(s) = app_{\{a_1,h(A')\}}(s)$ and $app_{a_1;h(A')}(s) = app_{a_1;\ldots;a_n}(s)$.

Equality $app_A(s) = app_{\{a_1,h(A')\}}(s)$ holds by Lemma 3.15. For equality $app_{a_1;h(A')}(s) = app_{a_1;\ldots;a_n}(s)$, observe that $app_{a_1;h(A')}(s) = app_{A'}(app_{a_1}(s))$. Since actions in A' are simply commuting and $app_{A'}(app_{a_1}(s))$ is defined, by the induction hypothesis we have $app_{A'}(app_{a_1}(s)) = app_{a_2;\ldots;a_n}(app_{a_1}(s))$ according to the given ordering, and hence $app_{a_1;h(A')}(s) = app_{a_1;\ldots;a_n}(s)$. □

4 Checking interference with SMT

We can exactly check the proposed notion of interference, according to Definitions 3.3, 3.5 and 3.9, by means of checking the satisfiability of some SMT formulas at compile time. As far as we know, previous approaches used syntactic or limited semantic approaches (6; 7; 11).

The following example is taken from the *Planes* domain, described in Section 6. The problem consists in transporting people between several cities using planes, with a limited number of seats. The considered actions are `board` and `fly`. Boarding is limited by seat availability, and a plane can only fly if it is transporting somebody.

If we consider action a as

$$\begin{aligned}&\text{board\_person1\_plane1\_city1} = \\ &\quad \langle \text{seats\_plane1} > \text{onboard\_plane1} \wedge \\ &\qquad\quad \text{at\_person1\_city1} \wedge \text{at\_plane1\_city1}, \\ &\quad \{\text{at\_person1\_city1} \mapsto \bot,\ \text{in\_person1\_plane1} \mapsto \top, \\ &\qquad\quad \text{onboard\_plane1} \mapsto \text{onboard\_plane1} + 1\}\rangle\end{aligned}$$

and action b as

$$\begin{aligned}&\text{fly\_plane1\_city1\_city2} = \\ &\quad \langle \text{onboard\_plane1} > 0 \wedge \text{at\_plane1\_city1}, \\ &\quad \{\text{at\_plane1\_city1} \mapsto \bot,\ \text{at\_plane1\_city2} \mapsto \top\}\rangle\end{aligned}$$

then most planners, checking interference syntactically, would determine interference, since a modifies the `onboard_plane1` variable and b uses this variable in its precondition. However, according to Definition 3.9, it can be seen that a does not affect b, since

1. $Pre_a \wedge Pre_b \wedge \neg(Pre_b\sigma_a)$ is T-unsatisfiable, because the precondition of b, onboard_plane1 $>$ $0 \wedge$ at_plane1_city1, cannot be falsified by the effects of a, namely {at_person1_city1 $\mapsto$ $\bot$, in_person1_plane1 $\mapsto \top$, onboard_plane1 $\mapsto$ onboard_plane1 + 1},
2. a and b are simply commuting and
3. $Pre_a \wedge Pre_b \wedge \neg(x\sigma_{h(\{a,b\})} = x\sigma_b\sigma_a)$ is T-unsatisfiable for all variables x.

The first check can be modeled in the SMT-LIB language (1) as follows:

```
;; declaration of problem variables.
(declare-fun at_person1_city1 () Bool)
(declare-fun at_plane1_city1 () Bool)
(declare-fun seats_plane1 () Int)
(declare-fun onboard_plane1 () Int)

;; preconditions of actions "board" and "fly"
(assert (and (> seats_plane1 onboard_plane1)
             at_person1_city1 at_plane1_city1))

(assert (and (> onboard_plane1 0)
             at_plane1_city1))

;; negated precondition of fly after board
(assert (not (and (> (+ onboard_plane1 1) 0)
                  at_plane1_city1)))
(check-sat)
```

Note that in the negated precondition of `fly`, we are replacing each variable by the term which represents its value after the execution of `board`, i.e. we replace onboard_plane1 by onboard_plane1 + 1. A negative answer should be obtained from the SMT solver.

The check of simply commutativity would consist in checking for all variables commonly modified by the two actions, if the effects can be commuted. In the example, there are no common variables modified by both actions. Hence, suppose we are checking whether two arbitrary assignments $\{x \mapsto exp_1\}$ and $\{x \mapsto exp_2\}$ commute. According to Definition 3.3, this would consist in checking whether $\neg(exp_2\{x \mapsto exp_1\} = exp_1\{x \mapsto exp_2\})$ is T-satisfiable. A negative answer would imply T-unsatisfiability of this negation and, hence, commutativity of the assignments. The third check can be implemented analogously by means of satisfiability checks.

An important difference with the purely syntactic definition of interference of (13) is that we include preconditions of the checked actions in our checks. More precisely, the reason for adding the preconditions in all satisfiability checks is that we require that the two actions for which we check potential interference can occur in parallel. This way, we are able to avoid many 'false positive' interference relationships, which would make the final formula grow unnecessarily. It can also be seen as a combination of a interference and reachability check, all in one. All in all, we obtain a much more fine-grained notion of interference, that will help to increase the parallelization of actions. Note that the interference relationships determined semantically will always be a subset of the interference relationships determined syntactically. Interestingly, we will be using a SMT solver both at compile time, as an oracle to predict interference relationships, and at solving time.

4.1 Ungrounded checking

Although the time needed for performing the proposed interference checks by calling a modern SMT solver is negligible, in big problems the number of calls can grow considerably. Here we propose an

improvement to be able to check interference between actions without the need of grounding them first.

The main idea is to model the interference queries as before, but substituting the action parameters appearing in the preconditions and effects with variables instead of concrete values. Then, incorporate to the formula the disequality relationships between variables of different types and ask the solver for a model. If the first action can affect the second, a query to the SMT solver would give a concrete set of values that explain why the first action can interfere with the second. Note however that what we really need is not one but all models of the formula. This is because we need all the concrete grounded actions to be able to add the necessary mutual exclusion clauses to the formula.

The #SMT problem is the problem of counting the number of satisfying assignments of a given SMT formula. In our case, we do not need to count them, but to enumerate them. Unfortunately, no efficient implementation of an SMT solver enumerating models has been found. One alternative could be to successively call the SMT solver by adding a clause forbidding the model found, until no model exists. But this approximation would need at least as many queries to the SMT solver as concrete interference cases exist. For this reason we propose an alternative.

Let us consider the original lifted actions in the PDDL model of the previous example:

```
(:action board
 :parameters (?p - person ?a1 - aircraft ?c1 - city)

 :precondition (and (at ?p ?c1)
                    (at ?a1 ?c1)
                    (> (seats ?a1) (onboard ?a1)))

 :effect (and (not (at ?p ?c1))
              (in ?p ?a1)
              (increase (onboard ?a1) 1)))

(:action fly
 :parameters (?a2 - aircraft ?c2 ?c3 - city)

 :precondition (and (at ?a2 ?c2)
                    (> (onboard ?a2) 0))

 :effect (and (not (at ?a2 ?c2))
              (at ?a ?c3)))
```

Suppose we have three planes in the problem: A320-1, A320-2 and A320-3. If we assign A320-1 to parameters `?a1` and `?a2`, one should find the same interferences than if we assign A320-2 to parameters `?a1` and `?a2`. The same should happen if we assign A320-1 and A320-2 or A320-2 and A320-3 to `?a1` and `?a2`, respectively. So, to reason about interference between actions `board` and `fly`, we will need to determine, e.g. if the actions interfere in the case that `?a1` and `?a2` are the same aircraft. Or in the case that cities `?c1` and `?c2` are the same, etc. That is interference is determined depending on the equality relationship between parameters.

Since equality or disequality between parameters of different types has no sense, the first thing that is needed is to group the parameters of the same type in sets, by its most general declared type.

Then, one should need to consider all different possible equality and disequality relationships between the parameters of the same type, to find out in which cases one action can interfere with another action. If we consider all the possible partitions of the set, they map directly to all the possible equalities and disequalities between elements of the set.

EXAMPLE 4.1
Consider the set $\{A, B, C\}$. All the partitions of this set are as follows: $\{\{A\}, \{B\}, \{C\}\}$, $\{\{A, B\}, \{C\}\}$, $\{\{A, C\}, \{B\}\}$, $\{\{A\}, \{B, C\}\}$ and finally $\{\{A, B, C\}\}$. When two elements appear in the same set, we consider them to be equal, and when they appear on different sets, we consider them to be different. So, on partition $\{\{A, C\}, \{B\}\}$ we should consider that $A = C, A \neq B$ and $C \neq B$.

Once the set partitions have been generated for each set of parameters, we need to compute the Cartesian product between all the sets in order to obtain the combination of equality and disequality relations between the parameters of the two actions.

We propose to model interference as shown previously, and do one query for each possible combination of equality and disequality between parameters of the two actions. Instead of using variables or concrete values, for convenience when we intend for two parameters to be equal we substitute them for the same integer, and by different integers when we want them to be different.

This approach results in much fewer queries to the SMT solver. Considering the same example with a total of 2 planes, 4 persons and 6 cities, a grounded checking would result in $4 \times 2 \times 6 = 48$ grounded fly actions and $2 \times 6 \times 6 = 72$ grounded board actions. This would result in $48 \times 72 = 3456$ grounded checks.

The total number of partitions of an n-element set is the Bell number B_n. If we now consider the proposed ungrounded checking method, the sets of parameters will be the following: for planes $S_{planes} = \{a1, a2\}$ and for cities $S_{cities} = \{c1, c2, c3\}$. Bell numbers for these sets are $B_2 = 2$ and $B_3 = 5$, so we will have a total of $2 \times 5 = 10$ ungrounded checks. As it can be seen, the number of checks needed using this technique is much lower than using the grounded checking, and thus scales much better with large problems.

Algorithm 1 Mutex Generation

Require: A : a set of PDDL actions
Ensure: M : a set of actions pairs $\langle a, b \rangle$ where a interferes with b

1: $A_p \leftarrow$GeneratePairs(A)
2: $M \leftarrow \emptyset$
3: **for all** $\{a_1, a_2\} \in A_p$ **do**
4: $M \leftarrow M \cup$ InterferenceChecking(a_1,a_2)
5: **end for**
6: **return** M

Algorithm 1 discovers, for a given set of ungrounded actions, the minimum set of interferences between them. It starts by generating all the possible pairs of actions. Then, for each generated pair, it calls Algorithm 2 to discover the minimum set of grounded interferences between the two actions.

Algorithm 2 Interference Checking

Require: a_1 : first action, a_2 : second action
Ensure: M : the set of ground interferences between a_1 and a_2

1: $[p_1, p_2, \ldots, p_n] \leftarrow$ groupByType(parameters(a_1) $\cup$ parameters(a_2))
2: $L \leftarrow$ setPartitions(p_1) $\times$ setPartitions(p_2) $\times \cdots \times$ setPartitions(p_n)
3: **for** $l \in L$ **do**
4: com $\leftarrow$pairWithIntegers(l)
5: **if** check1(com,a_1,a_2) $\vee$ check2(com,a_1,a_2) **then**
6: $M \leftarrow$ generateInterferences(a_1,a_2,com)
7: **end if**
8: **end for**
9: **return** M

Algorithm 2 receives two ungrounded actions and returns the minimum set of grounded interferences between them. It uses the following functions:

parameters given an action, it returns a set with all the parameters of that action.

groupByType receives a set of parameters and returns a list of sets. Each set groups the original parameters by its most general type.

setPartitions receives a set and efficiently (12) generates all the possible partitions of the original set.

pairWithIntegers receives an n-tuple of sets of sets of parameters (i.e. one of the possible partitions above), where each component corresponds to a type, whose elements (sets) denote parameters with the same value (and different to the parameters in the other sets). Then, it returns the same n-tuple but with each parameter paired to an integer. This integer will be equal to integers on the same set and different to others. For example given the n-tuple $(\{\{A, B\}, \{C\}\}, \{\{D\}, \{E\}\})$, as A and B belong to the same set, they will have the same integer associated. The other elements belong to different sets, so they will have different integers associated. Given this example, a possible return value would be

$$\{\{\{\langle A, 1\rangle, \langle B, 1\rangle\}, \{\langle C, 2\rangle\}\}, \{\{\langle D, 3\rangle\}, \{\langle E, 4\rangle\}\}\}$$

check1 This function encodes the first condition of interference explained in Definition 3.9 to SMT: substitutes each parameter of the action by the integer paired with it and finally checks and returns the satisfiability of the resulting formula.

check2 Does the same as the former, but with the second condition in Definition 3.9.

generateInterferences generates all the ground instances of the pair of actions (a_1, a_2) that correspond to the equalities and disequalities induced by *com*.

For example consider a problem with persons `p1` and `p2` and cities `Barcelona` and `London`. The two considered actions are a_1 = `board` with parameters `(?p - person ?c - city)` and

action $a_2 =$ `fly` with parameters `(?c1 -city ?c2 - city)`. If we consider the n-tuple *com* to be $\{\{\langle c, 1\rangle, \langle c1, 1\rangle, \langle c2, 1\rangle\}, \{\langle p, 2\rangle\}\}$ (all city parameters need to be equal), the generated interferences would be

$\langle$board_p1_barcelona, fly_barcelona_barcelona$\rangle$

$\langle$board_p1_london, fly_london_london$\rangle$

$\langle$board_p2_barcelona, fly_barcelona_barcelona$\rangle$

$\langle$board_p2_london, fly_london_london$\rangle$

5 Encodings

In this section we propose two different encodings for *planning as SMT*, as a particular case of PMT. We generalize Rintanen's (13) encoding for *planning as SAT* to include non-Boolean variables, with the idea of using an off-the-shelf SMT solver to solve the problem.

The proposed encodings are valid for any theory T under a quantifier-free first-order logic with equality. In particular, for numeric planning we could take T as the theory of the integers (or the reals) and use quantifier free linear integer (or real) arithmetic formulae. In the SMT-LIB standard (1), QF_LIA stands for the logic of Quantifier-Free Boolean formulas, with Linear Integer Arithmetic constraints, and similarly QF_LRA for the case of reals. These logics have a good compromise between expressiveness and performance, and therefore are a natural choice for *numeric planning as SMT*.

5.1 SMT encoding

Let $\pi = \langle S, A, I, G\rangle$ be planning problem modulo T, for a theory T under a quantifier-free first-order logic with equality. For each variable x in $var(S)$ and each time step t, a new variable x^t of the corresponding type is introduced, denoting the value of x at step t. Moreover, for each action a and each time step t, a Boolean variable a^t is introduced, denoting whether a is executed at step t.

Given a term s, by s^t we denote same term s, where all variables x in $var(S)$ have been replaced by x^t, and analogously for formulas. For example $(x + y)^t = x^t + y^t$, and $(p \wedge x > 0)^t = p^t \wedge x^t > 0$. For the case of effects, we define

$$\{x \mapsto \top\}^t \stackrel{def}{=} x^{t+1}$$

$$\{x \mapsto \bot\}^t \stackrel{def}{=} \neg x^{t+1}$$

$$\{x \mapsto s\}^t \stackrel{def}{=} (x^{t+1} = s^t),$$

where s denotes a non-Boolean term belonging to theory T. For example for an assignment $\{x \mapsto x + k\}$, where k is a constant, we have $\{x \mapsto x + k\}^t = (x^{t+1} = x^t + k)$. For sets of assignments, i.e. action effects, we define

$$(\{x \mapsto s\} \cup Eff)^t \stackrel{def}{=} \{x \mapsto s\}^t \wedge Eff^t$$

$$\emptyset^t \stackrel{def}{=} \top,$$

where s is a term (either Boolean or not) and Eff is a set of assignments.

The constraints are as follows. For each time step t, execution of an action implies that its precondition is met:

$$a^t \rightarrow Pre_a^t \qquad \forall a = \langle Pre_a, Eff_a \rangle \in A. \tag{1}$$

If the action is executed, each of its effects will hold at the next time step:

$$a^t \rightarrow Eff_a^t \qquad \forall a = \langle Pre_a, Eff_a \rangle \in A. \tag{2}$$

Finally, we need explanatory axioms to express the reason of a change in state variables. For each variable x in $var(S)$,

$$x^t \neq x^{t+1} \rightarrow \bigvee_{\substack{\forall a = \langle Pre_a, Eff_a \rangle \in A \\ \text{such that } \exists \{x \mapsto s\} \in Eff_a}} a^t . \tag{3}$$

That is a change in the value of x implies the execution of at least one action that has an assignment to x among its effects.

The previous constraints are complemented as follows depending on the type of parallelism we wish:

Sequential plans We can achieve a sequential plan by imposing an *exactly one* constraint on the action variables at each time step.

∀-Step plans According to Definition 3.12, in ∀-step plans we require the possibility of ordering the set of actions planned at each time step to any total order. Therefore, for each time step t, we simply add a mutex clause for every pair of interfering actions a_i and a_j:

$$\neg(a_i^t \wedge a_j^t) \text{ if } a_i \text{ affects } a_j \text{ or } a_j \text{ affects } a_i \tag{4}$$

∃-Step plans According to Definition 3.18, in ∃-step plans there must only exist a total ordering of parallel actions resulting in a valid sequential plan. A basic form of guaranteeing this is to take an arbitrary total ordering $<$ on the actions and forbid the parallel execution of two actions a_i and a_j such that a_i affects a_j only if $a_i < a_j$:

$$\neg(a_i^t \wedge a_j^t) \text{ if } a_i \text{ affects } a_j \text{ and } a_i < a_j. \tag{5}$$

It is not difficult to see that the encoding presented up to this point is correct for sequential plans, but it does not adhere to the parallel plan semantics derived from Definition 3.7 (happening execution). If two actions planned at the same time modify a same variable, two different situations can arise. On the one hand, if the assignments are not equivalent, then the SMT formula encoding the planning problem will become unsatisfiable. Although this is right for the Boolean case, it is more subtle for other theories, where effects can be cumulative. For example two assignments $\{x \mapsto x+1\}$ and $\{x \mapsto x+2\}$ would result into subformulas $x^{t+1} = x^t + 1$ and $x^{t+1} = x^t + 2$ which, together, are unsatisfiable. This, in practice, would rule out many parallel plans. On the other hand, if assignments were equivalent, then all but one would become redundant in the SMT formula. Then, the formula would possibly be satisfiable but, in this case, solutions would not adhere to the semantics of happening execution, where the effects of actions planned at the same time are composed for each

variable. A simple way of overcoming this problem could be to forbid the parallel execution of actions modifying a same non-Boolean variable, but this would rule out the parallelization of actions with cumulative effects. For this reason, a finer approach is described in the next section.

5.2 Chained SMT encoding

Here we present an encoding which builds onto the former in order to add support for cumulative effects in parallel plans.

Let N be the set of non-Boolean variables from $var(S)$. For each action $a = \langle Pre_a, Eff_a \rangle$ and each variable $n \in N$, let $Eff_{a,n}$ be the assignment $\{n \mapsto exp\} \in Eff_a$, or the empty set if there is no such assignment. For each $n \in N$, let $A_n = \{a \mid a \in A \wedge Eff_{a,n} \neq \emptyset\}$, i.e. the set of actions that modify variable n.

Constraints (2) are split and rewritten as follows, in order to take into account a possible 'chain of assignments' on each variable $n \in N$. First of all, we remove the effects on variables $n \in N$ such that $|A_n| > 1$, i.e. those that are modified by more than one action.

$$a^t \rightarrow (Eff_a \setminus \cup_{n \in N, |A_n|>1} \{Eff_{a,n}\})^t \; \forall a = \langle Pre_a, Eff_a \rangle \in A \tag{6}$$

Then, for each variable $n \in N$ such that $|A_n| > 1$, and for each time step t, the following constraints are introduced, using additional variables $\zeta^t_{n,0}, \ldots, \zeta^t_{n,|A_n|}$ of the type of n, and considering an enumeration $a_1, \ldots, a_{|A_n|}$ of the actions in A_n:

$$n^t = \zeta^t_{n,0} \tag{7}$$

$$\begin{aligned} a^t_i &\rightarrow Eff^t_{a_i,n}\{n^{t+1} \mapsto \zeta^t_{n,i}, n^t \mapsto \zeta^t_{n,i-1}\} \\ \neg a^t_i &\rightarrow \zeta^t_{n,i} = \zeta^t_{n,i-1} \qquad\qquad \forall a_i \in a_1, \ldots, a_{|A_n|} \end{aligned} \tag{8}$$

$$n^{t+1} = \zeta^t_{n,|A_n|}. \tag{9}$$

EXAMPLE 5.1

Let $A = \{a_1, a_2\}$, with actions $a_1 = \langle \top, \{x \mapsto x + 1, y \mapsto 0\} \rangle$ and $a_2 = \langle \top, \{x \mapsto x + 2\} \rangle$. Then $A_x = \{a_1, a_2\}$ and $A_y = \{a_1\}$. For time step t, we would add variables $\zeta^t_{x,0}$, $\zeta^t_{x,1}$, $\zeta^t_{x,2}$ and the following constraints:

$$\begin{aligned} &a^t_1 \rightarrow y^{t+1} = 0 && a^t_2 \rightarrow \top \text{(tautology)} \\ &x^t = \zeta^t_{x,0} \\ &a^t_1 \rightarrow \zeta^t_{x,1} = \zeta^t_{x,0} + 1 && \neg a^t_1 \rightarrow \zeta^t_{x,1} = \zeta^t_{x,0} \\ &a^t_2 \rightarrow \zeta^t_{x,2} = \zeta^t_{x,1} + 2 && \neg a^t_2 \rightarrow \zeta^t_{x,2} = \zeta^t_{x,1} \\ &x^{t+1} = \zeta^t_{x,2}. \end{aligned}$$

TABLE 1. Total number of instances of each domain solved by each approach. For each domain, the approach solving more instances is marked in bold. In case of draw, the faster is marked.

Domain	NR1	NR2	SYN	SEM	SEM+C
Depots	**13**	13	5	5	5
Petrobras	3	3	5	6	**7**
Planes	5	8	8	8	**8**
ZenoTravel	13	14	13	13	**15**
DriverLog	**18**	12	14	14	15
Total	**52**	50	45	46	50

TABLE 2. Sum of the number of time steps of the plans found, restricted to commonly solved instances. The first column shows, in parenthesis, the number of instances solved by all approaches. NumReach uses the same parallelism approach when using different background solvers, so only one column is included. The winning approach is shown in bold.

Domain	NR	SYN	SEM	SEM+C
Depots (5)	54	52	52	**51**
Petrobras (3)	19	12	12	**11**
Planes (4)	90	82	62	**45**
Zenotravel (13)	97	69	69	**42**
DriverLog (12)	104	85	84	**63**
Total (37)	364	300	279	**212**

6 Empirical evaluation

In this section we evaluate the impact of the proposed notion of interference on the length of parallel plans, using ∃-step semantics. Experiments have been performed using both syntactic and semantic checks of interference at compile time. Three different approaches have been considered:

- In the first approach, noted as SYN, we use the encoding of Section 5.1 with the ∃-step semantics, i.e. equations (1), (2), (3) and (5). Interference between actions is determined syntactically by checking concurrent assignment or assignment and inspection to the same variable.
- In the second approach, noted as SEM, we use the same encoding as in SYN, but inteference is checked semantically with the method explained in Section 4. However, as explained in Section 5.1, the proposed basic encoding does not support the concurrent modification of non-Boolean variables. For this reason, although interference is checked semantically in the first place, concurrent modification of non-Boolean variables is ruled out by an additional (syntactic) check afterwards.
- In the third approach, noted as SEM+C, interference is checked semantically as in SEM, but the chained encoding described in Section 5.2 is used instead, i.e. we use equations (1), (6), (7), (8), (9), (3) and (5).

Experiments have been implemented in the RanTanPlan system (3) and executed on an 8 GB Intel® Xeon® E3-1220v2 machine at 3.10 GHz, using Yices (5) v2.3.0 as back-end SMT solver with the QF_LIA logic (1), and a 2-hour timeout. For the sake of completeness, we compare the

TABLE 3. Reduction of interferences thanks to the semantic notion of interference.

Domain	SYN	SEM+C	Difference	% Removed
Depots (5)	5.35e6	1.09e6	4.26e6	79%
Petrobras (5)	1.90e7	3.10e5	1.87e7	96%
Planes (8)	8.60e4	7.92e3	7.80e4	98%
Zenotravel (13)	1.17e6	5.26e4	1.12e6	95%
DriverLog (14)	2.95e6	6.34e5	2.26e6	78%

TABLE 4. The two columns show, on average, the sizes of the SEM+C encoding with respect to the Syntactic one on the commonly solved instances.

Domain	Variables	Constraints
Depots (5)	399%	55%
Petrobras (5)	461%	86%
Planes (8)	233%	19%
Zenotravel (13)	291%	35%
DriverLog (14)	233%	101%

performance of our implementation with the numeric planner NumReach/SAT (10) using MiniSAT 2.2.0 (column NR1) and NumReach/SMT using Yices v2.3.0 (column NR2).

Five domains have been considered: the numeric versions of *ZenoTravel*, *DriverLog* and *Depots*, the real-life challenging *Petrobras* domain and a crafted domain called *Planes*. *ZenoTravel* and *DriverLog* are some of the domains in the literature with a higher numeric interaction between actions. Domains like *Rovers* or *Settlers* have been excluded because they are too big to show meaningful results with the encoding at hand and the chosen timeout. The *Petrobras* domain models a real-life problem of resource-efficient transportation of goods from ports to petroleum platforms. It was proposed as a challenge problem at the International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS 2012). Instances were modeled from (15).

Due to the limited numeric interactions between actions in the domains found in the literature, we additionally propose a new domain called *Planes* which is created from *ZenoTravel*, by adding some plausible numeric constraints, in order to help us demonstrate the benefits of checking interference between actions semantically. The full model can be found in Appendix A. Moreover, although in this section we just highlight the most relevant aspects of the experiments, details of each particular execution per instance and solver can also be found in Appendix A.

Table 1 shows the number of instances solved by each approach. Checking interference semantically and using the chained SMT encoding is best in *Petrobras*, *ZenoTravel* and *Planes*, while NumReach/SAT is best in *Depots* and *DriverLog*. The big gap in the number of solved instances in *Depots* is twofold: lack of intrinsic parallelism in the domain and being the perfect scenario for the reachability approach of NumReach.

Table 2 shows the sum of the number of time steps of the plans found, for commonly solved instances. Note that the domains where our implementation solves more instances are also the ones which exhibit more gains in parallelism. Note also the significant reduction in time steps from the syntactic approach to the semantic approach with the chained SMT encoding, especially in the *Planes* domain.

The importance of the semantic notion of interference and its checking through the SMT solver, is that it generates the minimum set of a-priori interferences between actions. Table 3 compares the SYN and SEM+C approaches on the commonly solved instances. For each family, it shows the sum of interferences, the difference between the two approaches, and the percentage of interferences that could be avoided thanks to the semantic notion of interference.

Table 4 compares the sizes of the first satisfiable formulas resulting from the SEM+C encoding and the SYN encoding. Generally the number of variables in the problems gets multiplied by a factor of three or four. Although this might seem a lot, in terms of the SYN encoding the number of variables ranges between one and fifty thousand. On the other hand, the number of constraints gets reduced. This reduction is significant, as the number of constraints in the SYN encoding can reach tens of milions in the biggest instances. Note that these size comparisons are dependant on the number of steps needed to solve the instance.

7 Conclusion

The main contribution of this paper is the formalization of an elegant and easy to implement solution to the problem of determining interference between actions in parallel plans. We have introduced a relaxed notion of interference in the context of PMTs, which can be checked by calling an SMT solver at compile time. The technique is therefore orthogonal to any solving method.

We have argued why the presented proposal is better than syntactic based ones and provided empirical evidence of its usefulness by showing a significant improvement in parallelism in some domains. We leave as further work a deeper study on the relation between the proposed semantics of happening execution and the one defined in (6) and (7).

Funding

This work has been partially supported by grant MPCUdG2016/055 (UdG), grant TIN2015-66293-R (MINECO/FEDER, UE), grant RTI2018-095609-B-I00 (MICINN/FEDER, UE) and grant EP/P015638/1 (EPSRC, UK).

References

[1] C. Barrett, A. Stump and C. Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org, 2010.

[2] M. Bofill, J. Espasa and M. Villaret. A semantic notion of interference for planning modulo theories. *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016*, pp. 56–64. AAAI Press, 2016.

[3] M. Bofill, J. Espasa and M. Villaret. The RanTanPlan planner: system description. *The Knowledge Engineering Review*, **31**, 452–464, 2016.

[4] Y. Dimopoulos, B. Nebel and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In *Recent Advances in AI Planning, Fourth European Conference on Planning (ECP'97)*, pp. 169–181. Vol. 1348 of LNCS, Springer, 1997.

[5] B. Dutertre and L. De Moura. The Yices SMT Solver. *Technical Report*. Computer Science Laboratory, SRI International, 2006. Available at http://yices.csl.sri.com.

[6] M. Fox and D. Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, **20**, 61–124, 2003.

[7] A. E. Gerevini, A. Saetti and I. Serina. An approach to efficient planning with numerical fluents and multi-criteria plan quality. *Artificial Intelligence*, **172**, 899–944, 2008.
[8] P. Gregory, D. Long, M. Fox and J. C. Beck. Planning modulo theories: extending the planning paradigm. *Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*. AAAI, 2012.
[9] M. Helmert. Decidability and undecidability results for planning with numerical state variable. In *Sixth International Conference on Artificial Intelligence, Planning and Scheduling (AIPS 2002)*, pp. 303–312. AAAI, 2002.
[10] J. Hoffmann, C. P. Gomes, B. Selman and H. A. Kautz. SAT encodings of state-space reachability problems in numeric domains. In *20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pp. 1918–1923, 2007.
[11] H. Kautz and J. P. Walser. State-space planning by integer optimization. In *AAAI/IAAI*, pp. 526–533. AAAI/The MIT Press, 1999.
[12] S. Kawano and S. Nakano. Constant time generation of set partitions. *IEICE Transactions*, **88-A**, 930–934, 2005.
[13] J. Rintanen. Planning and SAT. In *Handbook of Satisfiability*. Vol. 185 of Frontiers in Artificial Intelligence and Applications, pp. 483–504. IOS Press, 2009.
[14] J. Rintanen, K. Heljanko and I. Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, **170**, 1031–1080, 2006.
[15] D. Toropila, F. Dvorak, O. Trunda, M. Hanes and R. Barták. Three approaches to solve the Petrobras challenge: exploiting planning techniques for solving real-life logistics problems. In *IEEE 24th International Conference on Tools with Artificial Intelligence (ICTAI 2012)*, pp. 191–198. IEEE Computer Society, 2012.

Appendix A

A.1 Planes domain PDDL model

A.2 Detailed experiments

In the following tables, Syntactic, Semantic and Sem+chain columns denote the three approximations explained in Section 6, while the NR/SAT and NT/SMT columns show the results for the NumReach solver (10) with the SAT and SMT approximations, respectively.

Tables A5, A7, A9, A11 and A13 show the detailed results for the solved instances of each domain. The *sec.* columns contain the total time for each problem instance in seconds, with TO denoting a time out, and MO a memory out; *ts* columns contain the number of time steps of the plan; *aff.* columns contain the number of computed affecting relations between the problem actions.

Tables A6, A8, A10, A12 and A14 break down the total time into generation and solving time. *Gen. time* columns contain the time needed to ingest the problem files and generate the SMT formula. This includes essentially the time needed for the syntactic or semantic interference checks, being the rest of the tasks trivial as they always take less than 0.1 second. *Sol. time* columns contain the time needed from the point of having the first SMT formula, to having a valid plan. *Total time* columns contain the total time for each problem instance in seconds.

A.3 Missing proofs and extra lemmas

LEMMA A.1

Let a and b be two simply commuting actions, for a state space S modulo T, such that a does not affect b, and let $s \in S$ be a state such that a and b are applicable in s. Then $app_{\{a,b\}}(s) = app_{a;b}(s)$.

TABLE A5. Detailed results for the *Depots* domain.

n	NR/SAT		NR/SMT		Syntactic			Semantic			Sem+chain		
	sec.	ts	sec.	ts	sec.	ts	aff.	sec.	ts	aff.	sec.	ts	aff.
1	0.0	6	1.4	6	3.9	6	5e4	1.7	6	2e4	1.7	6	1e4
2	0.4	9	8.3	9	32.4	9	3e5	13.2	9	1e5	14.9	8	7e4
3	5.5	13	42.9	13	165.3	13	1e6	82.1	13	3e5	307.0	13	2e5
4	9.7	15	134.3	15	484.8	14	1e6	288.7	14	8e5	4292.0	14	5e5
5	TO	-	5187.9	21	TO	-	-	TO	-	-	TO	-	-
7	2.4	11	37.4	11	241.3	10		117.9	10	6e5	1142.8	10	3e5
8	15.2	15	403.0	15	MO	-	-	TO	-	-	TO	-	-
10	4.4	11	101.5	11	TO	-	-	MO	-	-	TO	-	-
11	43.3	18	TO	-	TO	-	-	TO	-	-	MO	-	-
13	2.6	10	84.3	10	TO	-	-	TO	-	-	TO	-	-
14	12.4	13	1314.0	13	TO	-	-	TO	-	-	TO	-	-
16	2.0	9	142.4	9	TO	-	-	TO	-	-	TO	-	-
17	6.8	8	395.6	8	TO	-	-	TO	-	-	TO	-	-
19	17.5	11	853.6	11	TO	-	-	TO	-	-	TO	-	-

TABLE A6. Detailed generation and solving times in seconds for each solved instance in the *Depots* domain.

n	Syntactic			Sem+chain		
	Gen. time	Sol. time	Total time	Gen. time	Sol. time	Total time
1	0.1	1.6	1.7	1.6	0.1	1.7
2	1.0	12.2	13.2	11.5	3.4	14.9
3	1.8	80.3	82.1	54.4	252.6	307.0
4	3.4	285.3	288.7	180.7	4111.3	4292.0
7	3.5	114.4	117.9	50.0	1092.8	1142.8

TABLE A7. Detailed results for the *Petrobras* domain.

n	NR/SAT		NR/SMT		Syntactic			Semantic			Sem+chain		
	sec.	ts	sec.	ts	sec.	ts	aff.	sec.	ts	aff.	sec.	ts	aff.
1	9.0	6	329.7	6	153.7	3	3e6	151.2	3	3e6	1209.2	3	4e4
2	17.5	6	357.7	6	197.9	4	4e6	193.4	4	3e6	1590.5	4	5e4
3	98.6	7	958.9	7	282.5	5	4e6	260.5	5	3e6	1335.5	4	6e4
4	TO	-	TO	-	467.6	6	4e6	391.4	6	3e6	1911.9	4	7e4
5	TO	-	TO	-	1435.6	7	4e6	1398.7	7	3e6	2667.8	4	8e4
6	TO	-	TO	-	TO	-		3373.0	8	3e6	1939.7	4	9e4
7	TO	-	TO	-	TO	-		TO	-		2666.7	4	1e5

TABLE A8. Detailed generation and solving times in seconds for each solved instance in the *Petrobras* domain.

n	Syntactic			Sem+chain		
	Gen. time	Sol. time	Total time	Gen. time	Sol. time	Total time
1	58.2	95.5	153.7	88.5	1120.7	1209.2
2	69.2	128.7	197.9	122.5	1468.0	1590.5
3	83.9	198.6	282.5	165.3	1170.2	1335.5
4	41.9	425.7	467.6	173.0	1738.9	1911.9
5	32.5	1403.1	1435.6	130.9	2536.9	2667.8
6	TO	TO	TO	124.4	1815.3	1939.7
7	TO	TO	TO	107.0	2559.7	2666.7

TABLE A9. Detailed results for the *Planes* domain.

n	NR/SAT		NR/SMT		Syntactic			Semantic			Sem+chain		
	sec.	ts	sec.	ts	sec.	ts	aff.	sec.	ts	aff.	sec.	ts	aff.
1	TO	-	36.4	15	0.9	13	4e3	0.2	10	6e2	1.9	9	4e2
2	3.3	18	37.7	18	6.5	16	4e3	1.1	12	6e2	3.9	10	4e2
3	TO	-	228.0	20	42.2	18	1e4	6.3	13	1e3	78.7	10	1e3
4	4.4	23	633.2	23	401.3	21	1e4	78.8	15	1e3	307.9	11	1e3
5	TO	-	763.9	22	179.9	20	1e4	47.5	15	2e3	46.5	11	1e3
6	5.3	25	1153.0	25	1971.5	23	1e4	585.6	18	2e3	331.8	13	1e3
7	TO	-	1238.4	23	374.5	21	1e4	54.4	16	2e3	41.8	11	1e3
8	5.0	24	1247.7	24	1508.6	22	1e4	119.4	17	2e3	66.5	11	1e3
12	15.5	21	TO	-	TO	-	-	TO	-	-	TO	-	-

TABLE A10. Detailed generation and solving times in seconds for each solved instance in the *Planes* domain.

n	Syntactic			Sem+chain		
	Gen. time	Sol. time	Total time	Gen. time	Sol. time	Total time
1	0.0	0.9	0.9	0.3	1.6	1.9
2	0.0	6.5	6.5	0.8	3.1	3.9
3	0.1	42.1	42.2	7.5	71.2	78.7
4	0.2	401.1	401.3	7.6	300.3	307.9
5	0.5	179.4	179.9	4.5	42.0	46.5
6	0.1	1971.4	1971.5	0.5	331.3	331.8
7	1.9	372.6	374.5	6.0	35.8	41.8
8	4.5	1504.1	1508.6	5.5	61.0	66.5

TABLE A11. Detailed results for the *Zenotravel* domain.

n	NR/SAT		NR/SMT		Syntactic			Semantic			Sem+chain		
	sec.	ts	sec.	ts	sec.	ts	aff.	sec.	ts	aff.	sec.	ts	aff.
1	0.0	2	0.1	2	0.0	1	5e2	0.0	1	1e2	0.0	1	1e2
2	0.0	7	1.5	7	0.0	3	8e2	0.0	3	2e2	0.0	3	1e2
3	0.0	6	3.6	6	0.1	3	3e3	0.1	3	1e3	0.1	3	6e2
4	0.0	6	2.3	6	0.2	4	4e3	0.1	4	1e3	0.1	3	7e2
5	0.0	7	6.8	7	0.4	4	7e3	0.2	4	3e3	0.1	3	1e3
6	0.0	7	4.1	7	0.8	6	9e3	0.4	6	3e3	0.2	3	1e3
7	0.0	8	9.0	8	0.7	5	1e4	0.4	5	4e3	0.2	3	2e3
8	0.3	7	7.7	7	2.5	5	4e4	1.6	5	2e4	0.5	3	5e3
9	0.3	9	18.1	9	24.8	8	5e4	20.9	8	2e4	1.3	4	6e3
10	0.6	9	24.4	9	70.0	8	5e4	42.1	8	2e4	1.8	4	6e3
11	3.3	8	18.4	8	8.1	6	8e4	5.7	6	3e4	3.2	4	8e3
12	3.6	10	99.0	10	73.6	7	9e4	76.8	7	3e4	4.4	4	1e4
13	22.0	11	565.3	11	1324.9	9	1e4	1270.2	9	4e4	3.3	4	1e4
14	TO	-	540.1	9	TO	-	-	TO	-	-	779.7	4	8e4
15	TO	-	TO	-	TO	-	-	TO	-	-	2850.6	4	2e5

TABLE A12. Detailed generation and solving times in seconds for each solved instance in the *Zenotravel* domain.

n	Syntactic			Sem+chain		
	Gen. time	Sol. time	Total time	Gen. time	Sol. time	Total time
1	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.1	0.1	0.1	0.0	0.1
4	0.0	0.2	0.2	0.1	0.0	0.1
5	0.0	0.4	0.4	0.1	0.0	0.1
6	0.0	0.8	0.8	0.2	0.0	0.2
7	0.0	0.7	0.7	0.2	0.0	0.2
8	0.1	2.4	2.5	0.5	0.0	0.5
9	1.0	23.8	24.8	1.2	0.1	1.3
10	2.8	67.2	70.0	1.6	0.2	1.8
11	0.6	7.5	8.1	2.9	0.3	3.2
12	5.1	68.5	73.6	3.8	0.6	4.4
13	100.1	1224.8	1324.9	2.7	0.6	3.3
14	TO	TO	TO	159.7	620.0	779.7
15	TO	TO	TO	321.4	2529.2	2850.6

TABLE A13. Detailed results for the *Driverlog* domain.

n	NR/SAT		NR/SMT		Syntactic			Semantic			Sem+chain		
	sec	ts	sec	ts	sec	ts	aff.	sec	ts	aff.	sec	ts	aff.
1	0.0	7	0.4	7	0.4	5	5e3	0.3	5	2e3	0.2	5	1e3
2	0.0	10	5.5	10	1.1	8	9e3	0.6	8	3e3	0.6	7	2e3
3	0.0	8	3.1	8	0.8	6	8e3	0.4	6	3e3	0.4	4	2e3
4	0.0	8	4.0	8	1.3	6	1e4	0.7	6	4e3	0.5	4	3e3
5	0.0	9	5.5	9	1.5	7	1e4	0.7	6	5e3	0.4	4	4e3
6	0.0	6	2.4	6	1.2	4	2e4	0.6	4	7e3	0.6	4	6e3
7	0.0	7	3.6	7	2.0	5	3e4	1.0	5	1e4	0.8	4	8e3
8	0.0	8	5.1	8	3.4	7	3e4	2.0	7	1e4	1.1	5	9e3
9	0.0	11	10.3	11	12.1	10	8e4	5.4	10	2e4	5.1	8	2e4
10	0.0	8	6.4	8	18.0	7	2e5	7.6	7	5e4	4.4	4	4e4
11	0.0	10	12.0	10	36.6	9	2e5	18.9	9	7e4	6.6	6	5e4
12	TO	-	TO	-	620.8	16	6e5	495.7	16	2e5	296.0	12	1e5
13	TO	-	TO	-	159.6	11	1e6	87.6	11	3e5	47.9	7	2e5
14	0.0	12	208.9	12	271.9	11	8e5	453.5	11	2e5	167.8	8	1e5
15	0.2	12	TO	-	MO	-	-	TO	-	-	5459.2	8	6e5
16	0.8	12	TO	-	TO	-	-	MO	-	-	TO	-	-
17	1.1	12	TO	-	MO	-	-	TO	-	-	TO	-	-
18	2.4	13	TO	-	TO	-	-	TO	-	-	TO	-	-
19	2.5	12	TO	-	TO	-	-	TO	-	-	TO	-	-
20	4.6	10	TO	-	TO	-	-	TO	-	-	TO	-	-

TABLE A14. Detailed generation and solving times in seconds for each solved instance in the *Driverlog* domain.

n	Syntactic			Sem+chain		
	Gen. time	Sol. time	Total time	Gen. time	Sol. time	Total time
1	0.0	0.4	0.4	0.2	0.0	0.2
2	0.0	1.1	1.1	0.5	0.1	0.6
3	0.0	0.8	0.8	0.4	0.0	0.4
4	0.1	1.2	1.3	0.4	0.1	0.5
5	0.1	1.4	1.5	0.4	0.0	0.4
6	0.0	1.2	1.2	0.6	0.0	0.6
7	0.1	1.9	2.0	0.7	0.1	0.8
8	0.2	3.2	3.4	0.9	0.2	1.1
9	0.7	11.4	12.1	3.0	2.1	5.1
10	1.7	16.3	18.0	3.6	0.8	4.4
11	3.0	33.6	36.6	4.6	2.0	6.6
12	28.8	592.0	620.8	251.3	44.7	296.0
13	7.9	151.7	159.6	40.2	7.7	47.9
14	13.9	258.0	271.9	140.3	27.5	167.8
15	TO	TO	TO	4510.1	949.1	5459.2

```
(define (domain planes)
(:requirements :typing :fluents)
(:types  city locatable - object
         aircraft person - locatable)
(:functions
 (at       ?x - locatable) - city
 (in       ?p - person) - aircraft
 (fuel     ?a - aircraft) - number
 (seats    ?a - aircraft) - number
 (capacity ?a - aircraft) - number
 (onboard  ?a - aircraft) - number
 (distance ?c1 - city ?c2 - city) - number)

(:action board
 :parameters (?p - person
              ?a - aircraft
              ?c - city)
 :precondition (and (= (at ?p) ?c)
                    (= (at ?a) ?c)
                    (> (seats ?a) (onboard ?a)))
 :effect (and (assign (at ?p) undefined)
              (assign (in ?p) ?a)
              (increase (onboard ?a) 1)))

(:action debark
 :parameters (?p - person
              ?a - aircraft
              ?c - city)
 :precondition (and (= (in ?p) ?a)
                    (= (at ?a) ?c))
 :effect (and (assign (in ?p) undefined)
          (assign (at ?p) ?c)
          (decrease (onboard ?a) 1)))

(:action fly
 :parameters (?a - aircraft ?c1 ?c2 - city)
 :precondition (and (= (at ?a) ?c1)
                    (> (onboard ?a) 0)
                    (>= (fuel ?a) (distance ?c1 ?c2)))
 :effect (and (assign (at ?a) ?c2)
          (decrease (fuel ?a) (distance ?c1 ?c2)))
 )

(:action refuel
 :parameters (?a - aircraft)
 :precondition (and
                    (< (* (fuel ?a) 2) (capacity ?a))
                    (= (onboard ?a) 0))
 :effect (and (assign (fuel ?a) (capacity ?a)))))
```

FIGURE A1. PDDL model of the Planes domain

PROOF. Let $a = \langle Pre_a, \sigma_a \rangle$ and $b = \langle Pre_b, \sigma_b \rangle$. Since a and b are applicable in s, we have that $app_{\{a,b\}}(s)$ is defined. Moreover, since a does not affect b, by Lemma A.5 we have that $app_{a;b}(s)$ is defined.

We conclude by showing that $app_{\{a,b\}}(s)[x] = app_{a;b}(s)[x]$ for every variable x. Recall that $app_{\{a,b\}}(s) = app_{h(\{a,b\})}(s)$, where $h(\{a,b\})$ denotes the happening action for a and b.

Now, by definition of application, we have $app_{h(\{a,b\})}(s)[x] = eval^s_T(x\sigma_{h(\{a,b\})})$ and $app_{a;b}(s)[x] = eval^s_T(x\sigma_b\sigma_a)$, for every variable x. On the other hand, since a does not affect b, $Pre_a \wedge Pre_b \wedge \neg(x\sigma_{h(\{a,b\})} = x\sigma_b\sigma_a)$ is T-unsatisfiable. And, since a and b are both applicable in state s, we have $T, s \models Pre_a$ and $T, s \models Pre_b$. Therefore $T, s \models Pre_a \wedge Pre_b \wedge (x\sigma_{h(\{a,b\})} = x\sigma_b\sigma_a)$, and thus $eval^s_T(x\sigma_{h(\{a,b\})}) = eval^s_T(x\sigma_b\sigma_a)$, which lets us conclude. □

LEMMA A.2
Let A be a set of non-interfering actions, for a state space S modulo T, and let $s \in S$ be a state such that all actions in A are applicable in s. Then $app_{a_1;\ldots;a_n}(s)$ is the same state for every total ordering $a_1 < \cdots < a_n$ of A.

PROOF. Since actions in A are non-interfering and applicable in state s, by Lemma A.5 we have that $app_{a_1;\ldots;a_n}(s)$ is defined for any total ordering $a_1 < \cdots < a_n$ of A. We conclude by showing that any two consecutive actions in the sequence $a_1; \ldots; a_n$ can be permuted, preserving the final state. Consider any two consecutive actions a_i and a_{i+1} in the sequence $a_1; \ldots; a_n$. Since $app_{a_1;\ldots;a_n}(s)$ is defined, so is $app_{a_1;\ldots;a_i}(s)$, and a_i is applicable in state $app_{a_1;\ldots;a_{i-1}}(s)$ (in case that $i = 1$, let $app_{a_1;\ldots;a_{i-1}}(s)$ denote the state s). Now, since actions in A are non-interfering, by Lemma A.5 we have that $app_{a_1;\ldots;a_{i-1};a_{i+1}}(s)$ is also defined, so a_{i+1} is also applicable in state $app_{a_1;\ldots;a_{i-1}}(s)$. Finally, by Lemma A.6, we have $app_{a_i;a_{i+1}}(app_{a_1;\ldots;a_{i-1}}(s)) = app_{a_{i+1};a_i}(app_{a_1;\ldots;a_{i-1}}(s))$ which, by definition of application, is equivalent to $app_{a_1;\ldots;a_{i-1};a_i;a_{i+1};\ldots;a_n}(s) = app_{a_1;\ldots;a_{i-1};a_{i+1};a_i;\ldots;a_n}(s)$. □

LEMMA A.3
Let A be a set of simply commuting actions, for a state space S modulo T, such that $|A| \geq 2$. Then, for every action $a \in A$, we have that a and $h(A\backslash\{a\})$ are simply commuting, and $h(\{a, h(A\backslash\{a\})\}) = h(A)$.

PROOF. Let $A = \{a_1, a_2, \ldots, a_n\}$, $a = a_1$ and $A' = A\backslash\{a\} = \{a_2, \ldots, a_n\}$. According to the definition of happening action, $\sigma_{h(A')} = \cup_{x \in var(S)}\{\sigma_{x,2} \circ \cdots \circ \sigma_{x,n}\}$, where $\sigma_{x,i}$, for i in $2..n$, is the mapping of variable x in the effects of action a_i. Now, since composition of functions is associative, we have that $\sigma_{x,1} \circ (\sigma_{x,2} \circ \cdots \circ \sigma_{x,n}) = \sigma_{x,1} \circ \sigma_{x,2} \circ \cdots \circ \sigma_{x,n}$ for every variable x, being $\sigma_{x,1}$ the mapping of variable x in the effects of action a_1. And, since actions in A are simply commuting, we have that $\sigma_{x,1} \circ \sigma_{x,2} \circ \cdots \circ \sigma_{x,n} = (\sigma_{x,2} \circ \cdots \circ \sigma_{x,n}) \circ \sigma_{x_1}$, which lets us conclude that a and $h(A')$ are simply commuting.

Now, provided that a and $h(A')$ are simply commuting, in order to prove that the happening actions $h(\{a, h(A')\})$ and $h(A)$ are equivalent, we need to show that they have equivalent preconditions and effects. For preconditions, we have $Pre_{h(\{a,h(A')\})} = Pre_a \wedge Pre_{h(A')} = \wedge_{a \in A} Pre_a = Pre_{h(A)}$. For effects, we have $\sigma_{h(\{a,h(A')\})} = \cup_{x \in var(S)}\{\sigma_{x,1} \circ (\sigma_{x,2} \circ \cdots \circ \sigma_{x,n})\}$ which, as seen before, is equivalent to $\cup_{x \in var(S)}\{\sigma_{x,1} \circ \sigma_{x,2} \circ \cdots \circ \sigma_{x,n}\}$. □

LEMMA A.4
Let A be a set of actions and a an action, for a state space S modulo T, such that the actions in $A \cup \{a\}$ are simply commuting. If a affects none of the actions in A, then a does not affect the happening action $h(A)$.

PROOF. Let $A = \{a_1, \ldots, a_n\}$. We proceed by induction on the number of actions n in A. If $n = 1$ then we are trivially done, since $h(A) = a_1$ and, by assumption, a does not affect a_1.

If $n \geq 2$, let $A' = A \setminus \{a_1\}$. Then a neither affects a_1 nor the happening action $h(A')$ (by the induction hypothesis). Moreover, since actions in $A \cup \{a\}$ are simply commuting, so are a, a' and $h(A')$. Then, by Lemma A.7, we have that a does not affect $h(\{a', h(A')\})$ and, by Lemma A.3, $h(\{a', h(A')\}) = h(A)$. □

LEMMA A.5
Let A be a set of actions, for a state space S modulo T, and let $s \in S$ be a state such that all actions in A are applicable in s. Then $app_{a_1;\ldots;a_n}(s)$ is defined for every ordering $a_1 < \cdots < a_n$ of A such that if $a_i < a_j$ then a_i does not affect a_j.

PROOF. By induction on the number of actions n in A. If $n = 1$ we are trivially done. If $n \geq 2$, consider any ordering $a_1 < \cdots < a_n$ of A such that if $a_i < a_j$ then a_i does not affect a_j. Let $a_1 = \langle Pre_{a_1}, \sigma_{a_1} \rangle$. First of all we show, by contradiction, that $app_{a_i}(app_{a_1}(s))$ is defined for every $a_i = \langle Pre_{a_i}, \sigma_{a_i} \rangle$ such that $a_1 < a_i$. Suppose that $T, app_{a_1}(s) \not\models Pre_{a_i}$, i.e. that a_i is not applicable after applying a_1 in state s. This is equivalent to say that $T, s \not\models Pre_{a_i}\sigma_{a_1}$ and, since s is an assignment, to $eval^s_T(Pre_{a_i}\sigma_{a_1}) = \bot$. Now, by assumption, we have $T, s \models Pre_{a_1}$ and $T, s \models Pre_{a_i}$, since all actions are applicable in state s. Therefore, $T, s \models Pre_{a_1} \wedge Pre_{a_i} \wedge \neg(Pre_{a_i}\sigma_{a_1})$, i.e. $Pre_{a_1} \wedge Pre_{a_i} \wedge \neg(Pre_{a_i}\sigma_{a_1})$ is T-satisfiable, contradicting that a_1 does not affect a_i. Finally, since all actions a_i such that $a_1 < a_i$ are applicable in state $app_{a_1}(s)$, by the induction hypothesis we have that $app_{a_2;\ldots;a_n}(app_{a_1}(s))$ is defined for any ordering $a_2 < \cdots < a_n$ of $A \setminus \{a_1\}$ such that if $a_i < a_j$ then a_i does not affect a_j, and hence so is $app_{a_1;a_2;\ldots;a_n}(s)$ for the ordering we have considered. □

LEMMA A.6
Let a and b be two non-interfering actions, for a state space S modulo T, and let $s \in S$ be a state such that a and b are applicable in s. Then $app_{a;b}(s) = app_{b;a}(s)$.

PROOF. Since a and b are non-interfering, then they are simply commuting, and neither a affects b nor b affects a. Then, by Lemma A.1, we have $app_{\{a,b\}}(s) = app_{a;b}(s)$, and $app_{\{a,b\}}(s) = app_{b;a}(s)$□

LEMMA A.7
Let a, b and c be three simply commuting actions, for a state space S modulo T. If a affects neither b nor c, then a does not affect the happening action $h(\{b, c\})$.

PROOF. Let $a = \langle Pre_a, \sigma_a \rangle$, $b = \langle Pre_b, \sigma_b \rangle$ and $c = \langle Pre_c, \sigma_c \rangle$. We need to prove that

1. $Pre_a \wedge Pre_{h(\{b,c\})} \wedge \neg(Pre_{h(\{b,c\})}\sigma_a)$ is T-unsatisfiable,
2. a and $h(\{b, c\})$ are simply commuting and
3. $Pre_a \wedge Pre_{h(\{b,c\})} \wedge \neg(x\sigma_{h(\{a,h(\{b,c\})\})} = x\sigma_{h(\{b,c\})}\sigma_a)$ is T-unsatisfiable for every variable $x \in var(S)$.

For condition 1, since $Pre_{h(\{b,c\})} = Pre_b \wedge Pre_c$, we have it follows that $Pre_a \wedge Pre_{h(\{b,c\})} \wedge \neg(Pre_{h(\{b,c\})}\sigma_a) = Pre_a \wedge Pre_b \wedge Pre_c \wedge \neg((Pre_b \wedge Pre_c)\sigma_a) = (Pre_a \wedge Pre_b \wedge Pre_c \wedge \neg(Pre_b\sigma_a)) \vee (Pre_a \wedge Pre_b \wedge Pre_c \wedge \neg(Pre_c\sigma_a))$. Now assume that $Pre_a \wedge Pre_b \wedge Pre_c \wedge \neg(Pre_b\sigma_a)$ is T-satisfiable (the other case is analogous). Then $Pre_a \wedge Pre_b \wedge \neg(Pre_b\sigma_a)$ would also be T-satisfiable, contradicting that a does not affect b.

Condition 2 follows directly from Lemma A.3.

For condition 3, we proceed by contradiction. Let us assume that $Pre_a \wedge Pre_{h(\{b,c\})} \wedge \neg(x\sigma_{h(\{a,h(\{b,c\})\})} = x\sigma_{h(\{b,c\})}\sigma_a)$ is T-satisfiable for some variable $x \in var(S)$. Then, by definition of happening action, we have $Pre_a \wedge Pre_b \wedge Pre_c \wedge \neg(x(\sigma_{x,b} \circ \sigma_{x,c} \circ \sigma_{x,a}) = x(\sigma_{x,b} \circ \sigma_{x,c})\sigma_a)$ is T-satisfiable, where $\sigma_{x,a}$, $\sigma_{x,b}$ and $\sigma_{x,c}$ are the mappings of variable x in the effects of actions a, b and c,

respectively. So there exists some assignment s such that $T, s \models Pre_a$, $T, s \models Pre_b$, $T, s \models Pre_c$ and $eval^s_T(x\sigma_{x,b}\sigma_{x,c}\sigma_{x,a}) \neq eval^s_T(x\sigma_{x,b}\sigma_{x,c}\sigma_a)$. This implies the existence of some variable y different from x such that $\sigma_a[y] \neq y$. Moreover, since $\sigma_{x,b}$ and $\sigma_{x,c}$ are substitutions replacing only variable x, y must be a variable in $x\sigma_{x,b}$ or in $x\sigma_{x,c}$ and, necessarily, $eval^s_T(x\sigma_{x,b}\sigma_{x,a}) \neq eval^s_T(x\sigma_{x,b}\sigma_a)$ or $eval^s_T(x\sigma_{x,c}\sigma_{x,a}) \neq eval^s_T(x\sigma_{x,c}\sigma_a)$. But this, together with $T, s \models Pre_a$, $T, s \models Pre_b$ and $T, s \models Pre_c$, contradicts a affecting neither b nor c. □

Received 10 June 2019