J Sched
DOI 10.1007/s10951-008-0083-7

A constraint programming formulation for planning: from plan scheduling to plan generation

Antonio Garrido · Marlene Arangu · Eva Onaindia

Received: 15 May 2007 / Accepted: 8 July 2008

Abstract Planning research is recently concerned with the resolution of more realistic problems as evidenced in the many works and new extensions to the Planning Domain Definition Language (PDDL) to better approximate real problems. Researchers' works to push planning algorithms and capture more complex domains share an essential ingredient, namely the incorporation of new types of constraints. Adding constraints seems to be the way of approximating real problems: these constraints represent the duration of tasks, temporal and resource constraints, deadlines, soft constraints, etc., i.e. features that have been traditionally associated to the area of scheduling. This desired expressiveness can be achieved by augmenting the planning reasoning capabilities, at the cost of slightly deviating the planning process from its traditional implicit purpose, that is finding the causal structure of the plan. However, the resolution of complex domains with a great variety of different constraints may involve as much planning effort as scheduling effort (and perhaps the latter being more prominent in many problems). For this reason, in this paper we present a general approach to model those problems under a constraint programming formulation which allows us to represent and handle a wide range of constraints. Our work is based on the original model of CPT, an optimal temporal planner, and it extends the CPT's formulation to deal with more expressive constraints. We will show that our general formulation can be used for planning and/or scheduling, from scheduling a given complete plan to generating the whole plan from scratch. However, our contribution is not a new planner but a constraint programming formulation for representing highly-constrained planning + scheduling problems.

Keywords Constraint programming · Planning · Scheduling · Constraints · Integration

1 Introduction

Planning, within the field of AI, is concerned with selecting a set of actions whose execution will lead from a particular initial state to a state in which a goal condition is satisfied. Scheduling, on the other hand, is aimed at determining which resources to allocate to actions and when to allocate them while satisfying the problem constraints. Broadly speaking, planning decides *which* actions to apply while scheduling decides *when* and *how* execute such actions. Planning research has been almost exclusively concerned with the logical (causal) structure of the relationship between the actions in the plan, rather than with the metric temporal structure, the dynamics of resource usage or other aspects beyond the classical assumptions. That is, planning decides the orderings of actions in such a way that the logical executability of the plan is guaranteed (Long and Fox 2003). However, the planning community has evidenced a need to push planning algorithms to capture more complex domains which closely approximate real problems (Fox and Long 2003; Gerevini and Long 2006; Hoffmann et al. 2004). This means to incorporate metric, temporal and resource constraints, numeric expressions, deadlines, restrictions on the objectives, soft constraints, persistences, temporal windows, quality metrics, etc. The purpose of this paper is to

A. Garrido (✉) · M. Arangu · E. Onaindia
Universidad Politécnica de Valencia, Camino de Vera s/n,
46071 Valencia, Spain
e-mail: agarridot@dsic.upv.es

M. Arangu
e-mail: marangu@dsic.upv.es

E. Onaindia
e-mail: onaindia@dsic.upv.es

present a mixed formulation, based on constraint programming, to handle planning and scheduling features. Firstly, we will present some essential background on planning, POCL planning and constraint programming to provide the reader with a general overview on the problem of planning and scheduling in AI.

1.1 Background

Planning aims at finding a set of actions which allows an executive agent to transform an initial state into another state that satisfies some goals. A classical planning problem is usually defined as a tuple $\langle \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$, where $\mathcal{I}$ and $\mathcal{G}$ represent the initial state and the goals to be satisfied, respectively, and $\mathcal{A}$ the available actions that can be applied. This definition of planning problem is inherited from the definition introduced some decades ago in STRIPS (Fikes and Nilsson 1971), where a state is a conjunction of positive literals that represent the properties' values of the problem objects. Literals in first-order state descriptions must be ground and function-free. Propositions are defined as positive literals and, consequently, a state informs about which propositions currently hold; if a proposition p holds in a given state then p is true in that state, and false otherwise. Additionally, each action specifies which propositions must belong to the state in order for the action to be applicable (preconditions) and which propositions the action will add or remove (positive and negative effects, respectively) to create a new state. This representation based on a finite set of propositions and a closed world assumption is usually called propositional representation (Ghallab et al. 2004).

Classical planning has usually adopted a propositional representation and makes a number of simplifying assumptions, among others: (i) the number of states is finite as the number of propositions remains invariable; (ii) each state is fully observable and the application of actions is deterministic, i.e. the same action a applied on the same state S always results in the same state S' (effects are totally predictable); (iii) actions have instantaneous duration and time is implicit (state transitions are immediate); and (iv) actions can be directly executed in the environment and the task of planning is to devise a plan that achieves the goals before any part of the plan is executed, also known as off-line planning. Under these assumptions, a classical plan consists of a collection of actions in a total or partial order, i.e. a sequential or parallel plan, where the only measure of quality is the number of actions, the number of plan steps or a linear combination of both.

There have been many algorithmic approaches to classical planning, such as state-space, plan-space or graph-based planning (Ghallab et al. 2004; Weld 1999). In this paper we mainly focus on plan-space planning, as it offers a natural and very appropriate way to handle time and complex constraints in a planning framework. Plan-space planning also provides efficient mechanisms to cope with multiple interferences among subgoals and to establish ordering relations among the actions in the plan. The flexibility provided by the least-commitment strategy allows a planner not to commit to a particular, fixed order until the whole plan structure is determined and fully detailed. This way, a plan-space search approach must take decisions about how the selection of actions affect the resolution of the rest of the problem. This type of approach is known as POCL (Partial-Order Causal-Link) and combines a representation in the form of partial-order planning and least-commitment with reasoning on causal links and threat resolution (Ghallab et al. 2004). POCL planners generate plans with actions in parallel and only establish ordering relations (precedences) when absolutely necessary. This is part of the least-commitment principle: execution times of actions are left unbound until the plan constraints determine their values. On the other hand, POCL reasoning basically lies on the concept of causal links, which represent the *cause–effect* relation between an action a (the producer action) that supports a proposition p required by action b (the consumer action). Clearly, a causal link implies an ordering constraint between the producer action and the consumer action. Moreover, no other action that deletes a proposition p can be executed between the producer and the consumer, i.e. threats to the causal link are not allowed. In order to solve a threat, the threatening action must be executed either before the producer action (mechanism known as promotion) or after the consumer action (known as demotion). A similar situation happens when two actions are mutually exclusive (mutex), i.e. when they cannot be executed at the same time. For instance, if two actions have contradictory effects because they add and delete, respectively, the same proposition one of the two actions must be promoted/demoted to solve the mutex. These powerful mechanisms of POCL planning, along with its expressive power and formal properties, such as soundness and completeness (Penberthy and Weld 1992), make it an appealing framework to incorporate time and other types of constraints.

Despite the success of algorithms in classical planning, scheduling capabilities have not traditionally been included in planning settings and they have been treated separately. A scheduling problem now expands the tuple that defines a planning problem to $\langle \mathcal{I}, \mathcal{G}, \mathcal{A}, \mathcal{R}, \mathcal{C}, \mathcal{M} \rangle$, where $\mathcal{I}, \mathcal{G}, \mathcal{A}$ are the elements for planning, and $\mathcal{R}$, $\mathcal{C}$, and $\mathcal{M}$ are the available resources to be used by actions, the constraints to be satisfied and the metric (objective function to measure quality) to be optimised in the problem, respectively. A major difficulty in this type of problems is that planning and scheduling features are strongly interrelated and separation is not always possible; planning an action may have a serious impact in the resource usage and affect negatively the problem metric, while satisfying a problem constraint may entail to discard an action that supports some problem goals. Consequently, research in planning and scheduling tends to combine features from both perspectives in a mixed way. An immediate consequence of this mixed approach is that a purely propositional representation is not enough to express scheduling features like actions with different duration and temporal constraints, continuous resources managed as numeric variables (i.e. fuel level, energy, profit, etc.), complex constraints in the form of inequalities on the variables (for instance, fuel ≥ 100), and multi-criteria metric optimisation. In order to be able to express these types of constraints, the propositional representation needs to be augmented to deal with numeric features. Intuitively, we can view a planning + scheduling (P&S) problem as a two-task problem, solving the propositional part and the numeric part. The propositional part comprises the problem structure, i.e. the actions which will form part of the plan along with the causal relationships to support the goals. The numeric part is mainly devoted to satisfy the time and resource constraints that involve the actions in the plan. One approach to tackle this type of problems is by extending the capabilities of a planner to perform reasoning on time and resources. Under this framework, the planner must work with scheduling features while doing planning, which entails significant changes in the planning algorithms and makes the solving process more complex. A different resolution scheme is to use a Constraint Satisfaction Problem (CSP) approach to solve the propositional part plus the numeric part of the problem. Under this framework, the CSP not only solves the scheduling features but also the causal relationships and goal supports, which are managed as constraint satisfaction too.

Due to its expressive power, POCL planning remains an appealing framework for dealing with scheduling features thanks to their ability to reason about supports, precedences and causal links in a least-commitment paradigm. Particularly, POCL is independent of the assumption that actions must be instantaneous or have the same duration, because the partial order and subgoal preserving mechanisms allow POCL approaches to define actions of arbitrary duration as long as the conditions under which actions interfere are well defined (Smith et al. 2000). Additionally, the great advantage of POCL-based frameworks is that they offer a high degree of execution flexibility, particularly in a temporal planning setting, in comparison to other planning systems. Finally, POCL planners traditionally aim at solving problems requiring a much richer expressiveness for time, resources, constraints, and actions.

1.2 POCL and constraint programming

POCL planning approaches represent a natural and very appropriate way to include and handle time in a planning framework. POCL's flexibility facilitates the incorporation of more real features and, therefore, it has been extensively used in real applications that require features from both planning and scheduling. The natural extension of POCL planning to handle time has been undertaken in many POCL planners such as HSTS (Muscettola 1994), ZENO (Penberthy and Weld 1994), or IxTeT (Ghallab and Laruelle 1994). All of them support metric–temporal features. HSTS includes quantitative duration constraints, ZENO provides reasoning about deadline goals, continuous change and exogenous events, and IxTeT includes timed preconditions, access to time points within the interval of an action and goals situated in time. These earlier planners state the flexibility of POCL planning to incorporate any type of temporal aspects in a planning framework. Moreover, IxTeT is capable of dealing with resources or concurrency and RAX (Jónsson et al. 2000), an AI-based planner/scheduler from the NASA, generates plans that account for on-board activities having different duration and requiring resources. IxTeT and RAX are two good samples of POCL planners + schedulers where planning and scheduling features are strongly interrelated.

A common framework that appears in these POCL planners is that they all use an interval representation and rely on constraint satisfaction techniques to represent and manage the relationship between intervals. This is referred as Constraint-Based Interval (CBI) approaches (Smith et al. 2000). Planners that use this interval representation maintain a constraint network to represent the temporal relations and keep track of the constraints in a plan. This approach offers a unified view of the temporal planning system, where the constraint network provides a single, uniform mechanism for doing temporal and numeric resource reasoning and handling the planning conflicts. In this case, the constraint network is a compact representation of all aspects that define a plan. The most common constraint network for CBI approaches is the Simple Temporal Network (STN) (Dechter et al. 1991). A STN can be represented by using a completely connected distance graph (*d-graph*) where each edge is labelled by the shortest temporal distance between two time points. In a POCL temporal framework each plan generated during the search process is represented as a STN. The advantage of using a STN is that it allows for rapid response to temporal queries as there exist well-known algorithms, such as arc-consistency, that can be applied for inference and constraint checking in the network. Planners like VHPOP (Younes and Simmons 2003) or IxTeT (Ghallab and Laruelle 1994) use a STN to record temporal constraints.

All the above mentioned planners are often referred to as performing constraint posting. In these approaches, constraint satisfaction techniques are added as an adjunct to the planning process, but the planning process itself is not formulated as a CSP (Van Beek and Chen 1999). In this *co-operative framework* between a POCL planner and a CSP solver, the underlying idea is to use a CSP solver within the POCL planning framework, leaving the logical structure of the plan to the planner (propositional planning problem) and the temporal features and remaining complex constraints to the CSP solver, which basically performs the plan scheduling. Consequently, selecting the actions that will form part of the plan is the task of the POCL algorithm whilst the CSP is in charge of allocating each plan action a time slot and checking the constraints consistency.

On the other hand, some works contemplate a constraint programming (CP) approach to planning, like GP-CSP (Do and Kambhampati 2001b), CPlan (Van Beek and Chen 1999), and CPT (Vidal and Geffner 2004, 2006) that solve (part of) the planning problem by encoding it as a CSP. The underlying idea of these approaches is to come up with a *CP formulation*, with all its variables and constraints, and solve it by applying CSP techniques. Hence, the CSP engine solves the overall problem, i.e. the logical structure of the problem as well as all the constraints defined in the problem. The CP formulation also encodes reasoning mechanisms to manage causal links, mutexes, orderings and threats, i.e. the POCL mechanisms to support preconditions and subgoal preserving, as well as all the metric, temporal, and resource constraints. This general formulation through constraint programming has the ability to solve a planning problem by reasoning about supports, mutex relations, and causal links in very elaborate models of actions. A CP formulation can be generated from the domain + problem definition (Garrido et al. 2006; Van Beek and Chen 1999; Vidal and Geffner 2004, 2006) or compiled from an intermediate structure, such as a planning graph (Do and Kambhampati 2001b). Additionally, we can find hand-coded formulations (Van Beek and Chen 1999) or automated formulations (Do and Kambhampati 2001b; Garrido et al. 2006; Vidal and Geffner 2004, 2006). The latter offer the advantage that they do not require a user to provide specific hand-coded domain knowledge.

From the standpoint of constraint checking, the CP formulation subsumes the cooperative framework because the CSP solver will not only carry out the constraint-based scheduling but also the reasoning about precondition supports and inconsistencies. Thus, we can say that the CSP solver is acting as a *scheduler* in the cooperative approach and as a *planner* + *scheduler* when being used in a CP formulation. It is important to highlight that when the CSP solver is only acting as a scheduler its task is to check whether there exists a correct variable assignment (time and resources) that supports the current causal structure of the given plan, that is, a correct schedule for the plan. On the contrary, when the CSP solver is acting as a planner + scheduler its task is to select the actions that support the problem (sub)goals and satisfy the time, resource and complex constraints, which makes the solving process much more difficult; after all, P&S subsumes scheduling.

In a CP model, we might also use the CSP for *pseudo-planning*, thus leaving open some planning supports and fixing others. In this case, we let the CSP solver take the decision on the supporting actions for the remaining open preconditions. When the CSP solver is performing pseudo-planning, the initial plan is already partially known because it has been partially solved or because some of its actions are initially imposed. In such a case, the CSP has, besides checking the temporal/resource constraints, to find the appropriate supports for the remaining open preconditions. Whatever the use of the CSP, the key issue is that the CP formulation is not altered and the only difference lies in which part of the formulation to activate or deactivate as typically happens in conditional/dynamic CSPs (Mittal and Falkenhainer 1990).

There are two general issues related to CP encodings that are worth mentioning:

– The conversion of a planning problem into a CP formulation may result in a large CSP model, particularly when the planning domain involves many actions. First, the number of variables and their domain size grows with the number of actions. Second, the number of constraints increases with the number of interferences between actions. This potential blow-up in the size of the model gives rise to a potentially exponential resolution process rather than a potentially exponential number of polynomial CSPs as it would be in the case of a cooperative POCL-CSP framework.
– In some approaches the CSP model can be simplified by starting the encoding from an intermediate structure rather than from the whole planning domain. For instance, GP-CSP solves a planning graph by automatically compiling it into a CSP and then applying a standard CSP algorithm, which replaces the Graphplan's backward search (Blum and Furst 1997). A planning graph does not comprise a planning domain but only the part of the problem reachable up to a certain level, which is an indication of the plan's length. Additionally, a planning graph provides action-distance estimates which can be very helpful to generate a more efficient CP formulation in terms of reducing the number of constraints and the size of the variables' domains. CPT makes an extensive use of lower bounds, in the form of admissible heuristics, on the starting times of preconditions and actions to identify structural mutexes and define distances between actions (Vidal and Geffner 2006).

The contribution of this paper is a CP formulation for POCL planning with complex constraints. Our formulation extends the original model of CPT by introducing a set of extensions in the model of actions, temporal and numeric resource constrains, and user-specified quality metrics. The next section presents an overview of the main characteristics of CPT.

1.3 Overview of CPT

CPT is an optimal temporal POCL planner based on constraint programming (Vidal and Geffner 2004, 2006). CPT provides a domain-independent formulation of temporal planning by combining a POCL branching scheme with sound pruning rules. The main contributions of CPT are:

- A domain-independent formulation of temporal planning based on constraint programming. The formulation encodes all actions in the problem domain, not only the actions in the plan. Variables are associated to each action, its preconditions and starting times. Constraints represent disjunctions, rules, temporal constraints, or their combinations.
- Distance-based heuristic estimates as search bounds. A preprocessing stage calculates: (i) lower bounds on the times to achieve the preconditions and set their earliest possible starting time; (ii) distances between actions, as the cost of the shortest-path connecting two actions; (iii) permanent mutex information, when the distance between two preconditions or actions is infinite; and (iv) an upper bound on the plan's length, which is iteratively increased until a solution plan is found, thus guaranteeing optimality *w.r.t.* makespan. The information computed in this preprocessing stage permits us to reduce the variables' domain and generate more restrictive constraints.
- A specific POCL-based search engine that uses the previous formulation and heuristics to perform branching by iteratively selecting and fixing flaws, and backtracking upon inconsistencies in a state space.

All in all, CPT is a planner that performs heuristic branching to select the next flaw to be fixed, as in a POCL framework, while using a CP formulation that allows the planner to propagate constraints and apply pruning mechanisms.[1]

1.4 Our contributions

In this paper we focus on a general CP formalism for POCL planning with complex constraints, following the line presented in Garrido et al. (2006). The formulation is based on the original model proposed by CPT and *extended* with:

- A more elaborate and expressive model of actions. Unlike CPT that uses a conservative model of actions (Smith and Weld 1999), we work with a model which subsumes the conservative model and the one proposed in PDDL2.1[2] (Fox and Long 2003). Our model allows the user to specify actions whose duration can vary within an interval and conditions/effects which can be required/generated at any time point.
- Numeric capabilities to express the use of continuous resources in actions. We do not explicitly support continuous functions to model continuous effects, but we handle discrete changes of resources to happen at certain times.
- More expressive and complex constraints, such as quantitative temporal constraints, persistences, deadlines, temporal windows, metric capabilities, etc.
- Quality metrics for the definition of multi-criteria optimisation. Unlike CPT that can only find optimal plans *w.r.t.* makespan, we can use any combination of variables as an optimisation expression.
- A flexible and automated CP formulation to encode P&S problems. The formulation admits different levels of completeness in the initial input plan. This way the solver will generate a plan from scratch if the plan structure is not provided in the formulation (plan generation from an initial empty plan); or the solver will work as a pseudo-planner if some part of the plan structure is represented in the formulation; or the solver will act as a scheduler if a complete plan is encoded in the formulation (plan scheduling).
- The CP formulation is a purely declarative representation of the planning problem and it is independent of the used CSP solver. A solver-independent formulation has the *advantage* that any systematic or local search-based engine can interpret and handle the model. However, this also means a *shortcoming* in terms of search performance as the resolution of our model entirely relies on the techniques used by the CSP solver.

1.5 Paper outline

This paper is organised as follows. So far we have introduced the basic background on planning, POCL planning and constraint programming. We have also presented an overview on CPT and summarised our main contributions. Section 2 outlines the expressiveness supported by our CP formulation. In Sect. 3 we describe in detail all the necessary elements for our formulation and make a comparison with PDDL3.0, the latest version of PDDL. Section 4 focuses on the model resolution and its properties. A complete application example is presented in Sect. 5 to illustrate the model adequacy for representing different scenarios of planning and scheduling. Section 6 presents some experimental results which show the scalability of our model when solved with a particular CSP solver. Finally, the conclusions of the paper along with our future work are discussed in Sect. 7.

[1]Other particular details of CPT will be gradually included in the paper as soon as they are needed.

[2]PDDL (Planning Domain Definition Language) is a standard language used in the AI planning community to define all the elements of a planning problem.

2 Expressiveness for complex domains

The model of actions we present in this paper allows the user to represent constraints between actions (persistences, quantitative precedences, temporal windows, etc.) and numeric capabilities to manage continuous resources. The CP formulation facilitates the specification of additional conditions and effects of actions as well as sophisticated constraints. In particular, we use a model of actions that subsumes the currently used model in the planning community and supports a higher level of expressiveness than the latest versions of PDDL[3] (Edelkamp and Hoffmann 2004; Fox and Long 2003; Gerevini and Long 2006). Firstly, the duration of an action can be specified with a value within a known interval to model uncertain durations or durations that are only known approximately. Moreover, the model offers possibilities for the specification of propositional/numeric conditions and effects that go beyond the conservative (Smith and Weld 1999) and non-conservative (Fox and Long 2003) model of actions used in modern planning. Conditions can be required in *any* interval of time which can range totally, partially or even out of the interval of the action execution. Thus, we can model conditions that do not fall within the interval of the action as, for instance, real *pre*conditions to be satisfied some time before the action starts or even conditions that are required *after* the execution of the action. For instance, in a logistics domain, a $fly(?plane, ?origin, ?destination)$ action usually requires the $?plane$ to be in the $?origin$ some time before the action *fly* starts, rather than just at its starting time. Similarly, a numeric condition like $fuel(?plane) \geq 1000$ can also be required during some time after the end of the *fly* action. On the other hand, effects can also be generated at any time (within or outside the action interval) and persist along some time. Let us consider the example of the *fly* action again. The effect of *flying* is generated when *fly* starts and only persists until it ends. The effect of having the $?plane$ at the $?destination$ happens when the action *fly* ends and has an infinite persistence. In contrast, the effect of having the $?plane$ at a given terminal of the $?destination$ might be generated some time after *fly* finishes because it is necessary that the $?plane$ traverses the airport segments to arrive at the terminal.

In addition to the expressiveness supported by this model of actions, a CP formulation also permits to specify some extra P&S features such as:

– Qualitative and quantitative precedence constraints between actions. We can model *explicit* ordering and synchronisation constraints between actions through any combination of their start and end times, such as start–start, start–end, etc.
– Temporal constraints between propositions, numeric variables, actions, and their combinations. Similarly to precedence constraints, our model considers *quantitative* temporal constraints between the generation times of propositions or numeric variables, the start/end times of actions and any particular combination.
– Deadlines in actions and goals. We can specify actions and goals to be required before a particular deadline. Deadlines are not only used for top-level goals but also for subgoals to be satisfied at any time during the execution of the plan. This is also particularly interesting to manage *intermediate goals*, i.e. goals that must hold at some specific point in the plan but they do not have to persist until the end of the plan.
– Temporal windows (like timed initial literals of PDDL2.2, Edelkamp and Hoffmann 2004) either for propositions, numeric variables, or actions. External constraints can impose limitations on the availability of some propositions (e.g. sunlight availability is limited), or time intervals when actions must be executed (e.g. actions that must occur only during opening times, or allocated within a particular schedule).
– Capabilities for numeric expressions to manage continuous resources, such as energy, profit, etc. We can express constraints (inequalities) on the numeric variables (e.g. $energy \geq 150$) and assign values (e.g. $profit += 50$). Additionally, the model supports the definition of *multi-criteria optimisation* which combines logical (propositional) and numeric expressions.

As can be seen from the previous features, the expressiveness of our model is centred around the concept of action rather than state as in PDDL3.0 (Gerevini and Long 2006). From our point of view, expressing constraints in terms of action instances is, in general, a more flexible representation as it allows access to any time point in the execution of a plan. In Sect. 3.3, we present a comparison between the expressiveness of our model and PDDL3.0.

3 A CP formulation for extended POCL planning

In this section we present our CP formulation for POCL planning with a special emphasis on the features presented in Sect. 2. The underlying idea (see Fig. 1) is to encode a P&S problem, that is, the causal structure of the plan (actions, causal links, supports, threats and mutexes) and the scheduling constraints (numeric resources, temporal constraints, etc.), as a CP model that can then be solved by any

[3]Although PDDL has evolved and now supports actions with duration, numeric features, multi-criteria optimisation, and trajectory constraints, it does not yet allow users to easily model complex scheduling features, such as quantitative precedences that involve several actions, complex temporal constraints, conditions/effects out of the action execution, persistences, etc.

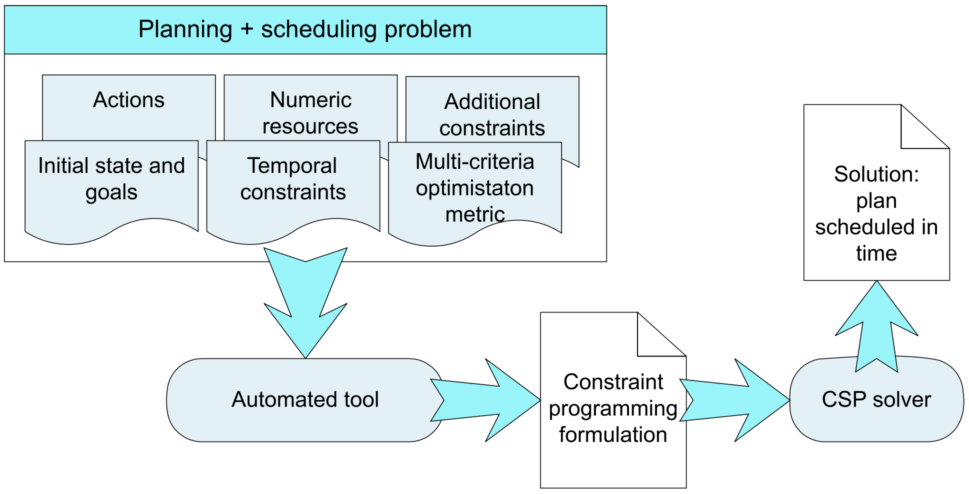

Planning + scheduling problem
Actions
Numeric resources
Additional constraints
Initial state and goals
Temporal constraints
Multi-criteria optimistaton metric
Solution: plan scheduled in time
Automated tool
Constraint programming formulation
CSP solver

Fig. 1 General schema of the process to encode a P&S problem into a CP formulation

CSP solver in order to find a solution plan. Obviously, by *any* CSP solver we mean a solver that supports the expressiveness of our constraint model which includes binary and non-binary constraints. An important feature of this model is that, depending on whether or not the causal structure of the plan is encoded in the formulation, the CSP solver will simply have to perform the plan scheduling (scheduler), or otherwise act as a planner + scheduler. In the middle point, the solver would work as a pseudo-planner when the causal structure of the plan is partially provided in the formulation.

In a CP setting, a problem is represented as a set of variables, a domain of values for each variable and a set of constraints among the variables. Since our formulation is highly influenced by the model of CPT, we first present the original CPT's formulation of variables and constraints, and then we extend it with the necessary components for our model.

3.1 Original CPT's formulation

Variables are used in CPT to represent actions and their preconditions. The formulation considers all the actions of a planning domain plus two dummy actions *Start* and *End*; *Start* supports the propositions of the initial state, whereas *End* requires the problem goals. CPT defines four variables:[4]

- $S(a) \in [0, \infty[$ stands for the starting time of a.
- $Sup(p, a) \in \{b_i\} \mid b_i$ adds p, that is, b_i is a support for action a.
- $Time(p, a) \in [0, \infty[$ stands for the starting time of $Sup(p, a)$ (time is discrete).
- $InPlan(a) \in [0, 1]$ indicates the presence of a in the plan. Both $InPlan(Start)$ and $InPlan(End)$ are set equal to 1 (*true*).

On the other hand, CPT defines five basic types of constraints:

- Bounds, *w.r.t. Start* and *End*:

  $S(Start) \leq S(a)$,
  $S(a) \leq S(End)$.

- Preconditions (b_i a supporting action of a):

  $Time(p, a) \leq S(a)$,
  $S(b_i) \leq S(a)$.

- Causal links and threat resolution introduced by b_i:

  $S(b_i) + duration(b_i) \leq Time(p, a) \vee S(a) \leq S(b_i)$.

- Mutex (effect-interfering between two actions):

  $S(a) \leq S(b_i) \vee S(b_i) \leq S(a)$.

- Support constraints:

  If $Sup(p, a) = b_i$ then $Time(p, a) = S(b_i)$.

CPT also includes distance-based estimates in the constraints to make them more informative, thus increasing pruning. Additionally, it introduces other redundant constraints which are not needed for soundness or completeness but for performance reasons.

3.2 Our model: enriching the expressiveness

3.2.1 *Variables and domains*

As in CPT, variables are basically used to define actions and conditions,[5] both propositional and numeric, supporting actions and the time when the conditions are achieved (in our

[4]In CPT, the actual names for the three first variables are $T(a)$, $S(p, a)$, and $T(p, a)$, respectively. As our model uses many more variables, we adopt more representative names and keep this new notation throughout the paper, even when referring to CPT, to avoid confusion.

[5]Note that in our formulation we use the term *condition* instead of *precondition* since conditions can be required at any time point (some time before, some time during, some time after, etc.) *w.r.t.* the execution of the action. We will present some examples for this in Sect. 5.

Table 1 Formulation of variables and their domains

Variable	Domain	Description
Block 1. Variables necessary for an action		
$S(a), E(a)$	$[0, \infty[$	Extension to CPT. Start and end time, respectively, of action a. $S(Start) = E(Start) = 0$ and $S(End) = E(End)$
$dur(a)$	$[dur_{\min}(a), dur_{\max}(a)]$	Duration of action a within two positive bounds. $dur(Start) = dur(End) = 0$
$InPlan(a)$	$[0, 1]$	Same meaning as in CPT. $InPlan(Start) = InPlan(End) = 1$
Block 2. Variables necessary for the propositional part of an action		
$Sup(p, a)$	$\{b_i\} \mid b_i$ adds p	Same meaning as in CPT. Symbolic variable to denote the potential supporters b_i; it represents the causal link $b_i \xrightarrow{p} a$
$Time(p, a)$	$[0, \infty[$	Same meaning as in CPT. Time when the supporter b_i assigned to variable $Sup(p, a)$ supports p (time is continuous)
$Persist(p, a)$	$[0, \infty[$	Persistence of condition p for action a; it represents the duration of the effect p, after which p disappears
$Req_{start}(p, a)$, $Req_{end}(p, a)$	$[0, \infty[$	Interval $[Req_{start}(p, a), Req_{end}(p, a)]$ at which action a requires p
Block 3. Variables for the numeric part of an action		
$Sup(\phi, a)$	$\{b_i\} \mid b_i$ supports ϕ for a	Analogous to $Sup(p, a)$ but for numeric condition ϕ
$Time(\phi, a)$	$[0, \infty[$	Analogous to $Time(p, a)$: time when the supporter b_i assigned to variable $Sup(\phi, a)$ updates ϕ
$Req_{start}(\phi, a)$, $Req_{end}(\phi, a)$	$[0, \infty[$	Interval $[Req_{start}(\phi, a), Req_{end}(\phi, a)]$ at the numeric condition $Cond(\phi, a)$ must be satisfied for action a (see block 3 in Table 2)
$V_{actual}(\phi, a)$	$]-\infty, \infty[$	Actual value of fluent ϕ at time $Req_{start}(\phi, a)$ if a requires $Cond(\phi, a)$. Otherwise, $V_{actual}(\phi, a)$ encodes the actual value of ϕ at time $S(a)$
$V_{updated}(\phi, a)$	$]-\infty, \infty[$	Updated value of ϕ by action a. This variable is only necessary if a modifies the value of ϕ

formalism, time is modelled in $\mathbb{R}$). We will use p to denote a propositional condition/effect, and ϕ for numeric variables used in the conditions/effects of actions. We will also refer to numeric variables as *fluents* to avoid confusion with the variables of the CP model itself. Variables are associated to each action encoded in the formulation, being these all actions of the planning domain (after grounding all the operators), or a smaller subset in case of scheduling or pseudo-planning. Variables, their initial domains and their descriptions are organised into three blocks and presented in Table 1.

Variables in Block 1 are self-explanatory, but variables in Block 2 require a deeper explanation. Variable $Sup(p, a)$ initially contains all possible supporting actions for p and its final assignment will set the value of $Time(p, a)$. Variable $Persist(p, a)$ allows us to model persistences in a very flexible way, to simulate different states in propositions, based on the action b_i which supports p and the action a for which p is supported. Particularly, if $Sup(p, a) = b_i$, the value of $Persist(p, a)$ is given by b_i, whereas if $Sup(p, a) = b_j$ such a value is given by b_j. Furthermore, if $Sup(p, a') = b_i$, i.e. action b_i also supports p for action a', the value of $Persist(p, a')$ can be different to $Persist(p, a)$. Therefore, persistences may depend on two factors: (i) the action b_i which supports p; and (ii) the action a which requires p. Intuitively, a value $Persist(p, a) = \infty$ represents the infinite persistence used in classical planning: once an effect is generated it only disappears when it is explicitly deleted by another action. On the contrary, a value $Persist(p, a) = 5$ means that p will be automatically deleted 5 time units after its achievement. A more detailed example on the use of this variable is given in Sect. 3.3.2. Variables $Req_{start}(p, a)$, $Req_{end}(p, a)$ ($Req_{start}(p, a) \le Req_{end}(p, a)$) provide a high expressiveness for dealing with many types of conditions. Usually, $Req_{start}(p, a) = S(a)$ and $Req_{end}(p, a) = E(a)$, but with this two-variable formulation we can represent a wide range of conditions, from punctual conditions when $Req_{start}(p, a) = Req_{end}(p, a)$, to conditions that are required during a longer interval, for instance, $Req_{start}(p, a) = S(a) - 1$ and $Req_{end}(p, a) = E(a) + 1$, to model a condition that is required beyond the duration interval of action a.

Variables $Sup(\phi, a)$, $Time(\phi, a)$, $Req_{start}(\phi, a)$, and $Req_{end}(\phi, a)$ in Block 3 are similar to those in Block 2, but now related to fluents instead of propositions. The calculation of supporters for a fluent ϕ from a proposition p and their propagation is importantly different, however. When dealing with fluents the term *support* is not fully appropriate since there is no particular action that supports a fluent, and many actions $\{b_i\}$ can update it. Therefore, the variable $Sup(\phi, a)$ will contain the last action b_i that updated the fluent ϕ before executing a, thus propagating its value to a. Additionally, during the propagation, the variable $V_{actual}(\phi, a)$ is twofold necessary. First, the value of ϕ stored in $V_{actual}(\phi, a)$ must be checked if there exists a numeric condition on ϕ in action a (e.g. $Cond(\phi, a) = V_{actual}(\phi, a) > 100$). Second, if a updates the value of ϕ, the new value is calculated from the initial value of $V_{actual}(\phi, a)$ (e.g. $V_{updated}(\phi, a) = V_{actual}(\phi, a) - 50.0$). Consequently, the value assigned to the variable $V_{updated}(\phi, a)$ will be a modification over the value of $V_{actual}(\phi, a)$ or, otherwise, it will be assigned an absolute value.

In our formulation, it is important to note that when an action a is not yet in the plan ($InPlan(a)$ is not bound yet) all the variables that action a involves—like supporters, causal links and times—are considered as conditional, as those variable are only meaningful when $InPlan(a) = 1$. Hence, when $InPlan(a) = 1$ all variables and constraints that action a involves are activated, and deactivated otherwise. On the other hand, in our statically-generated model, actions are encoded at most once in the formulation, since the number of times an action will be executed in the plan is unknown a priori. This limitation, known as canonicity restriction (Vidal and Geffner 2006), prevents a same action from being executed more than once in the plan. Although this situation is very common in scheduling, where each action occurrence happens only once, in planning it is possible to execute the same action at different times (actually, this means having different occurrences of the same action). This restriction can be relaxed by establishing a distinction between actions, as a generic type, and their occurrences, as done in Vidal and Geffner (2006). Basically, when an action a needs to be included in the plan it is automatically *cloned*, along with all its variables, initial domains and constraints, in order to have two independent action occurrences a and a'; one of them is included in the plan ($InPlan(a) = 1$) and the other remains unbound ($InPlan(a') = \{0, 1\}$) until it becomes necessary, time when it will be cloned again to have a third action occurrence.

If all variables *InPlan* are initially set to either 0 or 1 and the domains of the variables $Sup(p, a)$, $Sup(\phi, a)$ contain a single value, then the CSP solver will be acting as a scheduler. When all the actions and condition supporters in the plan are already solved, the solver only has to find a feasible assignment for the remaining variables (time and numeric resource assignment). On the contrary, if no variable *InPlan* is initially bound, the CSP solver will have to select the appropriate actions to build up the causal structure of the plan, besides solving the scheduling constraints. Finally, the CSP solver will perform pseudo-planning activities when (i) *some* of the *InPlan* variables are set to 0 or 1; or (ii) *all* variables *InPlan* are assigned a value but variables $Sup(p_i, a)/Sup(\phi_i, a)$ are left open. This functionality is particularly flexible when we have obligatory actions in a plan but the supporters are unknown, or when some parts of the plan must be fully committed. In essence, under this pseudo-planning setting we can represent several levels of commitment, depending on the number of variables that are initially known.

3.2.2 Constraints

Constraints establish relationships among the model's variables. Constraints in our formulation are organised into four blocks, as shown in Table 2.

Constraints in Block 1 provide the insights of an action within a plan and establish the precedence constraints with respect to *Start* and *End*. Blocks 2 and 3 include similar constraints for propositions and fluents to satisfy conditions and numeric conditions, respectively. The propositional part of the model also includes the optional initialisation of $Persist(p, a)$, whereas the numeric part shows a constraint which binds $V_{actual}(\phi, a)$ to a possible value of $V_{updated}(\phi, b_i)$; $Cond(\phi, a)$ represents the numeric condition that $V_{actual}(\phi, a)$ must satisfy.

Finally, Block 4 includes additional constraints considered as complex constraints, and consequently rarely managed in traditional planning, in the form of extra features, such as precedences, quantitative temporal constraints, persistences, deadlines, temporal windows, etc. For instance, the constraint $S(Start) + 100 \geq E(End)$ restricts the makespan to 100; $V_{actual}(\phi, End) > 50.5$ is a top-level goal where the value of fluent ϕ must be greater than 50.5, and $Time(\phi, End) < S(b)$ denotes that the final value of ϕ must be achieved before action b starts. Moreover, the formulation also allows us to model numeric variables to express the use of metric resources in the plan, by including the necessary numeric variables in the conditions and effects of the actions. This way, the model can include a multi-criteria optimisation function, such as: minimise $3 \cdot V_{actual}(\phi_i, End) - 2 \cdot V_{actual}(\phi_j, End) + 4 \cdot E(End)$.

Although we make an extensive use of these expressive constraints in our formulation, it is important to remark again that only the constraints involved by actions which are in the plan ($InPlan() = 1$) are considered in the resolution process.

Table 2 Formulation of constraints

Constraint	**Description**
Block 1. Constraints necessary for an action	
$S(a) + dur(a) = E(a)$	Start and end times of an action *a*
$E(Start) \leq S(a)$	Equivalent meaning to CPT. *a* must start after *Start*
$E(a) \leq S(End)$	Equivalent meaning to CPT. *a* must end before *End*
Block 2. Constraints necessary for the propositional part of the action	
$Time(p,a) \leq Req_{start}(p,a)$	Extension to CPT. Condition *p* for action *a* is supported before it is required
if $Sup(p,a) = b_i$ then	
$InPlan(b_i) = 1$	
$Time(p,a) =$ time when b_i adds p	Extension to CPT. Time of the supporter for variable $Sup(p,a)$ and the
$Persist(p,a) =$ persistence given by b_i	(optional) persistence of *p* for *a*
Block 3. Constraints necessary for the numeric part of an action	
$Time(\phi,a) \leq Req_{start}(\phi,a)$	Analogous to block 2
if $Sup(\phi,a) = b_i$ then	
$InPlan(b_i) = 1$	
$Time(\phi,a) =$ time when b_i updates ϕ	Time of the supporter for variable $Sup(\phi,a)$, and propagation of $V_{updated}(\phi,b_i)$
$V_{actual}(\phi,a) = V_{updated}(\phi,b_i)$	
$Cond(\phi,a) = V_{actual}(\phi,a)$ `comp-op` *Exp*	Numeric condition that ϕ must accomplish for *a* in $[Req_{start}(\phi,a), Req_{end}(\phi,a)]$. `comp-op` $\in \{<,\leq,=,\geq,>,\neq\}$ and *Exp* is any combination of variables and/or values that is evaluated in $\mathbb{R}$
Block 4. Additional complex planning constraints	
$Req_{end}(p,a) \leq Persist(p,a)$	Persistence constraint: the upper bound of the interval of a condition requirement never exceeds the value for the persistence of such a condition
$\min(tw(p)) \leq Req_{start}(p,a) \leq Req_{end}(p,a) \leq \max(tw(p))$, or $\min(tw(a)) \leq S(a) \leq E(a) \leq \max(tw(a))$	Temporal windows (in the form of external constraints) that propositions (or even fluents) and actions must hold, where $tw(p)$ and $tw(a)$ are the temporal windows for *p* and *a*, respectively
Var_i `comp-op` $Var_j + x$	Any type of binary constraint. `comp-op` $\in \{<,\leq,=,\geq,>,\neq\}$ and $x \in \mathbb{R}$
$Constraint(Var_i, Var_j, \ldots, Var_n)$	A general customised n-ary constraint that involves constraints among several variables of the model and may depend on each particular problem

3.3 Comparison between PDDL3.0 and our model

In this section we compare the expressiveness power of PDDL3.0 (Gerevini and Long 2006) and our model by analysing the type of constraints that can be modelled with each language. PDDL3.0 was the language of the fifth International Planning Competition (IPC–5[6]), an extension to PDDL that allows the user to express strong and soft constraints about the structure (state trajectory) of the plans and strong and soft problem goals. State trajectory constraints are conditions that must be met by the entire sequence of states visited during the execution of the plan. PDDL3.0 introduces several temporal modal operators to specify the temporal frequency of the properties of the states visited during the execution of a plan. Hence, plan trajectories assert conditions over the states reached by the plan rather than over the particular action instances in the plan. In Gerevini and Long (2006), authors recognise that "there would be value in also allowing propositions asserting the occurrence of action instances in a plan, rather than simply describing properties of the states", but they restrict this extension of the language to state constraints.

Constraints between the occurrence of action instances in a plan is exactly what our model allows to express. Temporal goal state conditions or deadlines can be represented in our model by specifying constraints between variables. Variables represent actions, conditions required by the ac-

[6]More information about IPC can be found at http://www.icaps-conference.org/.

tions and the occurrence times of those conditions. Hence, the specification of time constraints in our model is centred around the notion of *action* rather than the notion of *state*. In the next sections, we will see how state trajectory constraints can be expressed in terms of action constraints and vice versa.

3.3.1 State trajectory constraints into action constraints

A state trajectory constraint can be translated into a constraint between actions as long as the formulation of the action constraint reproduces the same plan trajectory. Given a particular state constraint, we have to identify the actions which generate the goal descriptors and use the appropriate mechanisms in our formulation to translate the PDDL3.0 temporal modal operators.

Table 3 shows three examples of PDDL3.0 constraints and their equivalent representation in our model. As can be observed in that table, the three examples of PDDL3.0 constraints can be easily encoded in our formulation. The first row in Table 3 shows a PDDL3.0 constraint that uses the basic modal operator `at-most-once`. In our model we post the following constraint: the sum of the *InPlan* variables of all the actions that generate $holding(?b)$ must be ≤ 1. The `within` constraints are formulated through a simple temporal constraint with respect to the time of the dummy action *End*. The modal operator `sometime-before` requires a little bit more of encoding as we have to retrieve all the actions which generate $at(?t, city1)$ and $at(?t,\ city2)$ and establish the corresponding temporal relation between the end points of each combination of actions.

Generally speaking, it is possible to come up with a translation scheme that converts deadlines into action constraints. If a proposition p is to be achieved before a certain time t, we can encode such a constraint in two different ways:

- As a condition of an action a by expressing that $Time(p, a)$ must happen before time t, as in the example of the `within` constraint in Table 3.
- As the effect of an action a (or set of actions $\{a_i\}$) by expressing that the start or end time of a (or $\{a_i\}$)—depending on whether the effect is achieved at start or at end—must happen before t, as in the translation of the modal operator `sometime-before` in Table 3.

Representing other PDDL3.0 constraints in our model formulation is also possible but not so straightforward. However, we can make use of some simple mechanisms, like creating dummy actions, to encode those PDDL3.0 constraints. Dummy actions are a very flexible way to artificially create a desired effect during an interval of time or to express the lower/upper bound on the time when the effect should hold. For example, we can represent the constraint:

(`hold-during` 9 20 $at(truck1, city1)$)

by creating the $dummy(truck1, city1)$ action with the condition $at(truck1, city1)$ to be maintained, thus avoiding threats, within the interval [9..20]. More specifically:

$$\begin{aligned}
&InPlan(dummy(truck1, city1)) = 1\\
&Req_{start}(at(truck1, city1),\\
&\qquad dummy(truck1, city1)) = 9\\
&Req_{end}(at(truck1, city1),\\
&\qquad dummy(truck1, city1)) = 20\\
&Time(at(truck1, city1),\\
&\qquad dummy(truck1, city1)) = 9\\
&Persist(at(truck1, city1),\\
&\qquad dummy(truck1, city1)) = 11
\end{aligned}$$

Other modal operators, such as `hold-after` or `always-persist`, can be likewise translated into our model by using dummy actions.

On the other hand, other PDDL3.0 constraints need more coding work in our formulation. In our statically-generated model we cannot directly represent constraints which will be satisfied an unknown number of times. Therefore, it is necessary to write a constraint which iterates all over possible alternatives and the CSP engine will check during runtime which of the specified constraints are fulfilled. Let us take the `always-within` modal operator; we can represent the constraint:

(forall ($?t$ − *truck*)
  (`always-within` 10
    ($<$ $fuel(?t)$ 20) $at(?t, fuel\_post)$))

by checking all the actions that set the *fuel* to a value below 20 and posting, for each of these actions, a temporal constraint that requires the truck to be at the *fuel_post* within 10 time units. More specifically:

Table 3 Translation of PDDL3.0 constrains into our formulation

PDDL3.0	**Our formulation**
(forall ($?b$ − *block*) (`at-most-once` $holding(?b)$))	$\forall b_i, b_j - block\ i \neq j$ $\sum InPlan(pickup(b_i)) + \sum InPlan(unstack(b_i, b_j)) \leq 1$
(`within` 250 $at(truck1, cityA)$)	$Time(at(truck1, cityA), End) \leq 250$
(forall ($?t$ − *truck*) (`sometime-before` $at(?t, city1)$ $at(?t, city2)$))	$\forall t_i - truck, \forall c_j, c_k - city$ $E(move(t_i, c_j, city2)) < E(move(t_i, c_k, city1))$

$\forall t_i - truck, \forall c_j, c_k - city$
if $V_{updated}(fuel(t_i), move(t_i, c_j, c_k)) < 20$
then
$E(move(t_i, c_j, c_k)) + 10$
$\geq E(move(t_i, c_k, fuel\_post))$

Overall, we can affirm that it is possible to encode every PDDL3.0 constraint into our formulation although some of the constraints are more laborious than others. It must be noticed that posting constraints adds some complexity to the model, as there exist more constraints and, probably, new variables. But, on the other hand, the higher the number of constraints is, the more search space pruning is done.

Some notes on preferences The specification of preferences (soft constraints) in PDDL3.0 makes sense within the context of a metric optimisation. Without a metric that specifically considers the achievement or violation of a soft constraint, preferences do not play a significant role in the planning process. A planner might either consider all soft constraints as strong ones (and thereby forcing the fulfilment of all of them) or ignore all of them, and both strategies would return a feasible solution. Thus, soft constraints and preferences affect the quality of the plan but not its validity.

The direct impact of the fulfilment or violation of soft constraints into a metric optimisation function can also be implemented in our model under the same considerations as in PDDL3.0. If preferences are not used in the metric function then they can only be handled as long as the CSP solver allows reasoning on preferences.

3.3.2 Action constraints into state constraints

Constraints among actions cannot be converted to PDDL3.0 constraints unless the effect of the action constraint results in an identifiable set of states. For instance, a precedence constraint between two actions can be modelled in PDDL3.0 by ordering a fictitious effect `sometime-after` another fictitious effect. However, a constraint that involves several actions does not impose a particular plan trajectory but simply defines a situation that may happen along the execution of the plan at each time the constraint holds. Therefore, if the satisfaction of the action constraint does not respond to a particular state pattern, the formulation of the action constraint in PDDL3.0 may be complicated or even impossible. Next we show how to encode in PDDL3.0, when possible, some of the constraints we manage in our model.

Expressiveness in the model of actions

Variable duration: $dur(a) \in [t, t']$. This constraint specifies a non-fixed duration for action a. Although we might define an action for each possible value in the interval, this restriction reveals the difficulty of expressing a constraint in PDDL3.0 that does not result in an enumerable set of states or in a finite trajectory. Moreover, this would entail a much costly search process. In general, PDDL3.0 cannot model continuous variables like $dur(a)$.

Modelling complex conditions and/or effects: $Req_{start}(p, a) \geq Time(p, a) + t$. This constraint denotes that action a can use p only after t time units have elapsed from the time when p is produced. This constraint might be represented in PDDL3.0 by inserting a dummy action of duration t, with an *overall* precondition p, and enforcing the chaining of action a after the dummy action. The drawback appears here when the '$\geq$' becomes '$=$'; in such a case it is not possible to chain the actions to happen exactly at a precise time. Similarly, constraints, such as $Req_{start}(p, a) = S(a) - t$ or $Req_{end}(p, a) = E(a) + t$, to represent happenings outside the action interval suffer from the same drawback. This fact evidences a major difference between PDDL3.0 and our model: action constraints are appropriate for specifying particular constraints as opposed to the state-based constraints of PDDL3.0.

Persistence of effects: in our model persistence depends on two factors: the action which produces a proposition p and the action which requires it. This is a flexible way to represent different durations for the same proposition depending on the producer action. Let us take the following example:

– a_1: acquiring cheese in a mall that will last 3 hours.
– a_2: acquiring (better quality) cheese in a delicatessen shop that will last 5 hours.

We have two actions, a_1 and a_2, with the same effect $acquired(cheese)$. We know that the persistence of $acquired(cheese)$ produced by a_1 and a_2 is 3 and 5 hours, respectively. This distinction is reflected when another action needs $acquired(cheese)$ as a condition, for instance, action a_3:

if $Sup(acquired(cheese), a_3) = a_1$ then
$Persist(acquired(cheese), a_3)) = 3$,
if $Sup(acquired(cheese), a_3) = a_2$ then
$Persist(acquired(cheese), a_3)) = 5$.

In this example, the persistence only depends on the action which produces the effect so the persistence of $acquired(cheese)$ is always 3 hours if cheese is acquired in a mall or 5 hours if acquired in a delicatessen shop. However, we can also specify a variable persistence depending on the action which uses that cheese. For instance, it might be possible to have two actions requiring some cheese, a_4 and a_5 (a_4 requires cheese for a standard recipe and a_5 for a gourmet recipe). Thus, the persistence of cheese generated by a_1 will be different for a_4 and a_5:

if $\mathit{Sup}(\mathit{acquired}(\mathit{cheese}), a_4) = a_1$ then
$\mathit{Persist}(\mathit{acquired}(\mathit{cheese}), a_4)) = 3$,
if $\mathit{Sup}(\mathit{acquired}(\mathit{cheese}), a_5) = a_1$ then
$\mathit{Persist}(\mathit{acquired}(\mathit{cheese}), a_5)) = 1.5$.

In this case we can see that the persistence of *acquired*(*cheese*) in a mall is shorter if it is later required by a_5 than by a_4. Intuitively, this variable persistence provides the modeller with a measure of the *quality* of the effect depending on both the producer and the consumer. Again, this persistence would be hardly modelled in PDDL3.0 as it would be necessary to include a huge number of combinations of dummy actions to artificially *spoil* the *acquired*(*cheese*).

Temporal constraints

Synchronisation: constraints in the form $S(a_i) = S(a_j)$, $E(a_i) = E(a_j) + x$, $\mathit{Time}(p, a_i) = S(a_j)$, etc. These constraints allow the user to express that two or more actions must start/end at the same time, or that some events must happen simultaneously. In PDDL3.0 one could be tempted to use timed initial literals with fictitious synchronisation conditions, but then we would have to state beforehand the temporal window when actions must be synchronised. To our knowledge it is not possible to force an exact synchronisation in PDDL3.0 at planning time because the exact time when actions will be executed is unpredictable.

Conditional, causal constraints

Causal consequence: if $\mathit{Sup}(p, a_i) = a_j$ then $\mathit{Sup}(q, a_j) = a_k$ or, alternatively, if $\mathit{Sup}(p, a_i) = a_j$ then $\mathit{InPlan}(a_k) = 1$. This type of constraint represents a causal consequence between actions in the form *if p is supported by a_j then q needs to be supported by a_k/a_k must be in the plan*. This type of causal chaining constraint is very helpful to represent external or ad hoc domain-dependent information. This conditional constraint cannot be represented in PDDL3.0 as the language does not support neither propositions asserting the occurrence of action instances nor mechanisms to access the elements of an action.

A variant of PDDL, PDDL+ (Fox and Long 2001, 2006), was designed to allow for the modelling of durative actions where properties of the objects involved are accessible at any point during the execution of an action. Continuous durative actions resulted in level 4 and a final level, level 5, comprised additional components to support the modelling of spontaneous events and physical processes. Although our formulation allows for the modelling of continuous resources, the solver cannot access the values of these continuous quantities at arbitrary points on the time-line of the plan. It is possible to model discrete changes to happen at certain times but we cannot model continuous change within the durative action, i.e. it is not possible to refer to instants between happenings on the time-line.

PDDL level 5 handles three constructs: actions, processes and events. Actions are the entities that agents must initiate to achieve planning goals. Duration is managed in PDDL level 5 by the introduction of the notion of a process that is automatically triggered when the domain situation matches some precondition, and events are automatically triggered as a result of the numeric changes brought about by domain processes. Even though our model does not support these functionalities, synchronisation constraints introduce a powerful mechanism to work in this direction.

4 Resolution of the CP formulation

For an efficient resolution of the model, we need to incorporate in the formulation a branching scheme to expand alternative solutions and pruning mechanisms to avoid parts of the search space. Analogously to classical POCL planning, branching is used to generate the search space of different partial solutions that appear when supporting a condition or fluent, or when solving a mutex/threat between actions. Particularly, in the former case a partial plan is created for each possible supporter (action b_i). In the latter case, a distinct constraint is posted to solve the mutex, thus preventing two actions from modifying the same proposition or fluent simultaneously (effects' interference). In a conservative model of actions like CPT's this operation avoids any kind of overlapping. However, in our model this does not prevent actions from partially overlapping, thus giving rise to a more permissive solution than the one used in classical planning. Instead of forcing two interfering actions to be executed sequentially, we simply prevent the simultaneous modification, but the two actions can still overlap.[7] As for threats, two disjunctive partial plans are created, one for solving the conflict by promotion and another for demotion.

4.1 Original branching in CPT

Branching in CPT proceeds by iteratively selecting and fixing flaws in non-terminal states and backtracking upon inconsistencies (Vidal and Geffner 2006). Flaws in CPT are equivalent to flaws in POCL planning, that is, (i) support threats; (ii) open conditions; and (iii) mutex threats (effects' interfering). When selecting a flaw in a state, CPT introduces

[7]Note that preventing two actions that modify the same fluent from overlapping in any way can also be easily formulated by placing the two actions sequentially.

Table 4 Formulation of branching situations

Constraint	Description
Block 1. Branching for the propositional part of the action	
$Sup(p,a) = b_i \wedge Sup(p,a) \neq b_j \mid \forall b_i, b_j \ (b_i \neq b_j)$ that supports p for a	Same meaning as in CPT. All the possibilities to support p while $\|Sup(p,a)\| > 1$, one for each b_i (encoded as the domain of $Sup(p,a)$)
$\forall b_i, b_j$ with effect-interference (mutex) $\mathcal{T}(b_i, p) \leftarrow$ time when b_i generates p $\mathcal{T}(b_j, \neg p) \leftarrow$ time when b_j deletes p $\mathcal{T}(b_i, p) \neq \mathcal{T}(b_j, \neg p)$	Extension to CPT. Mutex resolution: b_i and b_j cannot modify (generate and delete, respectively) p at the same time
$\forall b_i$ that threats causal link $Sup(p,a)$ $\mathcal{T}(b_i, \neg p) \leftarrow$ time when b_i deletes p $\mathcal{T}(b_i, \neg p) < Time(p,a) \vee$ $Req_{end}(p,a) < \mathcal{T}(b_i, \neg p)$	Extension to CPT. Threat resolution: promotion or demotion
Block 2. Branching for the numeric part of the action	
$Sup(\phi,a) = b_i \wedge Sup(\phi,a) \neq b_j \mid \forall b_i, b_j \ (b_i \neq b_j)$ that modifies ϕ for a	Analogous to $Sup(p,a)$ but for numeric condition ϕ
$\forall b_i, b_j \ (b_i \neq b_j)$ with effect-interference (mutex) $\mathcal{T}(b_i, \phi) \leftarrow$ time when b_i modifies ϕ $\mathcal{T}(b_j, \phi) \leftarrow$ time when b_j modifies ϕ $\mathcal{T}(b_i, \phi) \neq \mathcal{T}(b_j, \phi)$	Analogous to Block 1: prevents two actions from modifying ϕ simultaneously
$\forall b_i$ that threats causal link $Sup(\phi,a)$ $\mathcal{T}(b_i, \phi) \leftarrow$ time when b_i modifies ϕ $\mathcal{T}(b_i, \phi) < Time(\phi,a) \vee$ $Req_{end}(\phi,a) < \mathcal{T}(b_i, \phi)$	Analogous to Block 1: solves the threat

a binary constraint that splits the initial state into two disjunct child states that represent different alternatives to fix the flaw. This branching scheme is sound and complete.

Branching heuristics

CPT uses a POCL-inspired search engine for selecting flaws: first, support a threat; next, an open condition; and, finally, a mutex threat. As there may be several threats, open conditions or mutexes to select, CPT uses the following heuristics: (i) the support threat with minimum slack first; (ii) the latest open condition first; and (iii) the first encountered mutex in the plan. As indicated in Vidal and Geffner (2006), these heuristics have a significant influence on performance.

On the whole, we can say that CPT implements a branching scheme based on POCL-planning heuristics for flaw selection by means of posting constraints and closing the result under the CP propagation rules.

4.2 Branching situations in our model

In our CP formulation, branching is implicitly embedded in the model as other constraints (see Table 4). Although the number of constraints to represent branching situations may be very high, they are only managed when the actions involved in the constraints belong to the plan, i.e. $\forall a : InPlan(a) = 1$.

When using a standard CSP engine to solve the CP formulation, search performance entirely relies on the performance and efficiency of the engine's techniques. In other words, search is performed in a *blind-planning* way, which makes the incorporation of planning strategies to intelligently guide search very difficult. The point is that all variables and constraints encoded in the formulation lose their *planning meaning* when the model is solved by a CSP engine. This issue marks a clear difference between a planner, such as CPT, and our model. Particularly, CPT implements a POCL search to guide the solving process, whilst our resolution entirely depends on the CSP search engine itself. Therefore, there exist two alternatives to improve search performance: (i) implement our own POCL-search engine, but then we would be subject to work under such a solver; or (ii) encode standard branching heuristics in the formulation, so it can work under any general CSP solver. As one of our main goals was to design a CSP-solver-independent CP formulation, we opted for the second alternative.

Branching heuristics

Typically, a CSP is solved by iteratively applying the following steps: select a variable, assign it a value from its domain, propagate the new value, and repeat until a solution or an inconsistency is found. When an inconsistency appears, a chronological backtracking stage occurs and the last selected variable is assigned a new value, repeating the process once again. In a CP formulation with hundreds, or even thousands of variables, the criterion to select variables becomes very important; a wrong, blind selection of the variables may lead to unfruitful thrashing.

With the idea of keeping our formulation valid for any type of CSP solver, but aiming at improving search performance, we have developed two simple heuristics to be applied at the branching point that appears when supporting a causal link $Sup(p, a)$ or $Sup(\phi, a)$. These heuristics are equivalent to the usual *variable* and *value selection* heuristics in CSPs.

Variable selection Instead of *blind-planning* selections like choosing the variable with the minimum domain or the variable with the maximum number of constraints, we choose first the *Sup*-variables of the actions that are in the current plan. Intuitively, this selection guides the search to first support all the causal links of the actions with $InPlan = 1$. In order to do this, we incrementally build a list with all the *Sup*-variables (causal links) that need to be solved to find a plan. This list is initialised before starting the solving process with all the *Sup*-variables for the action *End*, i.e. the top-level of the problem. When the solver assigns a value v_i to one of these *Sup*-variables, it is actually asserting that action v_i will be the supporter of the causal link. At that moment, the list is updated with all the *Sup*-variables that action v_i needs. If the CSP solver backtracks and replaces the value v_i by v_j, the list is updated accordingly and all the *Sup*-variables inserted for v_i are replaced by those for v_j, and the process resumes again. Loosely speaking, this heuristic is thought to provide a *kind of planning behaviour* at a very *low cost* by solving first the variables which represent the causal structure of the plan (planning component). Once the *Sup*-variables are solved, the remaining variables are those related to the allocation of the actions in time (scheduling component), and they can be selected according to any classical heuristic, such as the variable with minimum domain, the variable which involves more constraints, etc.

In our particular implementation we use the list as a stack and select first all the propositional $Sup(p, a)$-variables and then all the numeric $Sup(\phi, a)$-variables. Note that assigning a value to these variables, e.g. $Sup(p, a) = b_i$ or $Sup(\phi, a) = b_i$, implies: (i) binding values to the *InPlan*, *Time*, *Persist* and V_{actual} variables (see Table 2); and (ii) performing a propagation that subsequently can lead to other bindings. When the list becomes empty the remaining uninstantiated *InPlan*-variables of the problem are set to 0 and the solver continues with the resolution of the scheduling variables according to their minimum domain. If no plan is found with the current *Sup*-assignments, the solver backjumps (avoiding all the *InPlan*-variables set to 0) to the last *Sup*-variable and resumes a classical chronological backtracking.

Value selection This branching heuristic establishes an ordering criterion according to the variables' values. The domain of the *Sup*-variables is the set of actions $\{b_i\}$ which support the causal link and our heuristic selects first the value (action b_i) which is already in the plan, i.e. $InPlan(b_i) = 1$; otherwise, the action with the earliest start time is selected first. Intuitively, the heuristic gives more priority to the existing actions in the plan to support new conditions. As a result the number of open conditions to be satisfied is reduced, and this benefits the search performance. The cost of applying this heuristic is polynomial *w.r.t.* the domain size of each *Sup*-variable.

To implement this heuristic, we create a set which contains the actions that are currently in the plan. When a *Sup*-variable is selected, a loop iterates all over the values of its domain and selects the first value that belongs to this set. Note that after a variable is instantiated to a value, the constraint propagation mechanism can detect and rule out other variables (actions) which do not satisfy the problem constraints.

It is clear that the two branching heuristics presented in this section are not very powerful in comparison with the heuristics used in modern planners but, at least, no specific implementation of a search engine is required. For instance, CPT applies more powerful heuristics to select the next flaw according to its type (support threat, open condition, or mutex threat) but they need to be implemented in a particular search engine and, therefore, cannot be directly moved to a standard CSP solver.

4.3 Pruning mechanisms

Generally, the success of POCL planning relies on the heuristics used to reduce the search space. A CP solving process can adopt a similar solution: use heuristic estimates, automatically *extracted* from the problem definition and directly *included* in the constraint formulation itself, as lower and upper bounds to reduce the variables' domain. The underlying idea is to use estimates similar to the ones used by heuristic planners to approximate the cost, usually the makespan, of achieving a set of propositions/fluents or actions from an initial state. The heuristic estimates can be calculated in different ways: (i) using a relaxed planning graph (Bonet and Geffner 1999, 2001; Do and Kambhampati 2001a); (ii) using a relaxed plan (Do and Kambhampati

2001a; Hoffmann and Nebel 2001); or (iii) using approximated equations (Bonet and Geffner 2001; Haslum 2006; Haslum and Geffner 2001). In particular, a simple heuristic calculated through a relaxed planning graph can estimate the earliest time when an action can be executed as an optimistic measure of reachability, that is, the distance between the dummy action *Start* and each action a, namely $\delta(Start, a)$. This way, the binding constraint $E(Start) \leq S(a)$ of Table 2 now becomes $E(Start) + \delta(Start, a) \leq S(a)$. Similarly, this estimate can be generalised to calculate the distance between each pair of actions a_i, a_j ($\delta(a_i, a_j)$), as in CPT (Vidal and Geffner 2006). Hence, any constraint in the form $E(a_i) \leq S(a_j)$ becomes $E(a_i) + \delta(a_i, a_j) \leq S(a_j)$, where lower bound $\delta(a_i, a_j)$ helps reduce the search space. Clearly, this approximates significantly better the start time of action a_j, which turns out to be very useful for threat resolution as well as for estimating the distance for the threatening action in the promotion/demotion operations. Further, like in any other planner a wide range of estimates can be applied and included in the CP formulation, taking into account that the properties of each heuristic estimate will affect the optimality of the solution. For instance, in a sound and complete CP approach it is easy to compute optimal makespan plans: given an initial bound for variable $S(End)$, the bound is increased until a plan is found, or it is decreased until no plan is found, as done in Vidal and Geffner (2006).

Problem formulation heuristics. Reachability Based on CPT's ideas, in our CP formulation we have included a distance-based reachability heuristic that estimates the earliest start time of each action. This estimate is calculated *once* per every problem, as a preprocessing step, by simply calculating a relaxed planning graph on the planning problem and *encoded* directly in the problem formulation. This planning graph is calculated in polynomial time.

More specifically, we build a two-spike structure, similar to the one presented in Garrido and Long (2004), which simulates a relaxed planning graph that includes information about numeric variables. This planning graph contains information about the earliest time of each action, starting from action *Start*, and considering both its propositional and numeric conditions, thus extending the propositional approach of CPT. Hence, the value approximated for each action a is $\delta(Start, a)$, which shrinks the domain of $S(a)$ and, consequently, of $E(a)$. For instance, if action a requires a condition supported in the end of actions b and c, with durations 10 and 15, respectively, which both start at time 0, the estimate included in the model responds to the constraint $S(a) \geq \delta(Start, a) = \max(10, 15) = 15$.

Although this reachability heuristic is primarily aimed at initialising the earliest start time of each action a, it is also possible to exploit this calculus to prune the actions which do not appear in the relaxed planning graph. In such a case, the value $\delta(Start, a) = \infty$ conflicts with $E(Start) + \delta(Start, a) \leq S(a)$ and $E(a) \leq S(End)$ ($S(End) < \infty$ always hold). Obviously, eliminating the unnecessary action reduces the formulation complexity.

It is important to highlight that the application of heuristic pruning mechanisms improves the overall solving process thanks to the use of bounds that reduce the variables' domain and help discard some partial plans. Moreover, this heuristic computation, which is encoded in the problem formulation, does not introduce an overhead in the CSP solving process (search) as it is computed in a preprocessing stage *before* invoking the CSP solver, i.e. the heuristic computation is independent of the CSP solving process itself. This kind of heuristics is very valuable for problems with many variables and constraints. For instance, CPT makes an extensive use of this type of heuristics for estimating distances between actions and propositions. As a result, CPT turns out to be faster than other optimal temporal planners and competitive *w.r.t.* other parallel planners when dealing with a simple model of time.

4.4 Properties

This CP formulation for planning offers very nice properties:

– The formulation shares the advantages of other CSP-like approaches, including the expressiveness of the modelling language, which shows very appealing for specifying complex planning and scheduling problems. Particularly, the underlying model allows the user to define a very elaborate type of actions and constraints, from basic causal links used in classical planning to very complex constraints, such as variable persistences, synchronisations, deadlines, and customised derived constraints among variables. Additionally, it is straightforward to include information on heuristic estimates in the formulation. This shows that the model can represent the physics of the planning and scheduling domain as well as valuable heuristic information calculated in a preprocessing stage to guide the search process.
– The formulation is a purely declarative representation of the planning domain + problem and is independent of any CSP solver. Consequently, any systematic or local search-based solver, which uses its own heuristics, can interpret and handle this constraint model.
• The whole formulation is automatically derived from the planning domain + problem definition, without the necessity of specific hand-coded domain knowledge. Other approaches, like CPlan (Van Beek and Chen 1999), require an expert user to develop a CSP model for each new problem, and this modelling stage may entail much intellectual effort. On the contrary, our formulation provides a CSP

model that is automatically and directly generated from the problem definition.[8] Obviously, this does not prevent the user from including any type of advice on how to better solve the problem and find a better plan, since the user can insert additional constraints to *recommend* and guide the generation of the plan (e.g. inserting a constraint that denotes an action must be always executed prior to another one).

– The formulation can be used for different purposes, such as just scheduling, pseudo-planning or planning + scheduling. Intuitively, the variables $\mathit{InPlan}(a)$, $\mathit{Sup}(p,a)$, and $\mathit{Sup}(\phi,a)$ contain the action choices to find a solution. If the domain of these variables is initially open, the CSP solver will decide which supporting actions are to be included in the plan according to the model and considering all complex constraints; otherwise, the only task of the CSP will be to allocate actions in time, that is scheduling a plan.

Regarding the formal properties of the formulation, it is important to note that it inherits the theoretical properties of a POCL framework, such as soundness, completeness, and optimality. Basically, soundness and completeness are guaranteed by two factors: (i) the definition of the model itself (all the branching alternatives to support causal links and to solve both the mutexes and threats are considered); and (ii) the completeness of the CSP solver that performs a complete exploration of the domain of each variable. Optimality is also guaranteed because of the completeness that explores all possible solutions and can be easily achieved by using a lower (upper) bound, which can be iteratively incremented (decremented) until a solution is found (no solution is found). However, this way of proceeding slightly depends on the precision supported by the CSP solver. Let us consider the makespan as the metric to be optimised and a sequential plan with two actions a and b of fixed duration 1, such that $\mathit{Req}_{end}(p,a) = E(a) = 1$ and $\mathcal{T}(b,\neg p) = S(b)$, i.e. p is required by a until its ending, and b deletes p when it starts. If the promotion rule to solve the mutex imposes the constraint $\mathit{Req}_{end}(p,a) < \mathcal{T}(b,\neg p)$, i.e. that action a is planned before b, $S(a) = 0$, $E(a) = 1$, $S(b) = 1.1$ is part of an (optimal) plan. However, $S(b) = 1.01$ also gives rise to a better but *still the same* plan, and the same happens if $S(b) = 1.001$. Consequently, the constraint $\mathit{Req}_{end}(p,a) < \mathcal{T}(b,\neg p)$ actually becomes $\mathit{Req}_{end}(p,a) + \epsilon \le \mathcal{T}(b,\neg p)$, where ϵ represents the minimal granularity managed by the solver.[9] It is important to note that this minimal granularity needs to be small enough not to alter the quality of the plan. This means that all the '$<$' or '$>$' constraints are internally managed by the solver like '$\le$' or '$\ge$', respectively, plus the corresponding ϵ. Therefore, optimality is possible but taking into consideration the solver's precision.

4.5 Implementation

Given a grounded version of a PDDL2.1 planning domain + problem,[10] we have implemented an automated translator in Java that generates the CP model with all the elements presented in the paper. The extra constraints not supported by PDDL2.1, such as synchronisation and/or quantitative precedence among actions, complex conditions/effects, intermediate deadlines, persistences, etc., are posteriorly encoded in the model. The CSP model is generated following the particular syntax of the CSP solver. First, we used CON'*FLEX*[11] as our solver but then changed to Choco[12] because it provides a more complete and extensible API based on Java. Actually, one of the advantages of Choco is that it allows users to define their own classes for variable and value selection and *plug* them directly into the solver. The current implementation consists of a class Problem—the central element of a Choco program—which is a factory with an API to create variables and post n-ary constraints in the constraint network. Choco has an event mechanism that performs a chronological backtracking search. It selects a value for a variable of the model and propagates that value throughout the constraint network, updating the lower and upper bounds of other variables. Each updating operation triggers new events that, in turn, propagate the values and perform the corresponding updatings until a feasible solution is found.

In our implementation of the CP model, we have focused on checking the validity of the model rather than on investigating and applying different heuristics to improve performance. So far, we have implemented on top of the Choco engine the two branching heuristics presented in Sect. 4.2, and we have encoded the reachability heuristic discussed in Sect. 4.3 in the CP formulation.

As we will show in the next section, the application of these two basic heuristics improve solver performance significantly.

[8]Particularly, our CSP model is automatically generated from the durative domain and problem files in PDDL2.1 format (Fox and Long 2003; Gerevini and Long 2006).

[9]The use of an $\epsilon > 0$ also appears in PDDL2.1 plan validation (Long and Fox 2001) in order to specify the minimal precision with which evaluate the quality of the plans. Although changes in this value can slightly change the makespan of the *same* optimal plan, this seems to be more a *philosophical* than a meaningful question in the constraint formulation.

[10]We use MIPS-XXL (Edelkamp et al. 2006) to perform the grounding of the actions. We thank Stefan Edelkamp and Shahid Jabbar for their assistance on using MIPS-XXL.

[11]CON'*FLEX* is a tool for solving CSPs that provides a command-based language to model CSP problems. It is available at http://www.inra.fr/internet/Departements/MIA/T/conflex.

[12]Choco is an object-oriented Java library for constraint satisfaction problems that can be downloaded from http://choco.sourceforge.net.

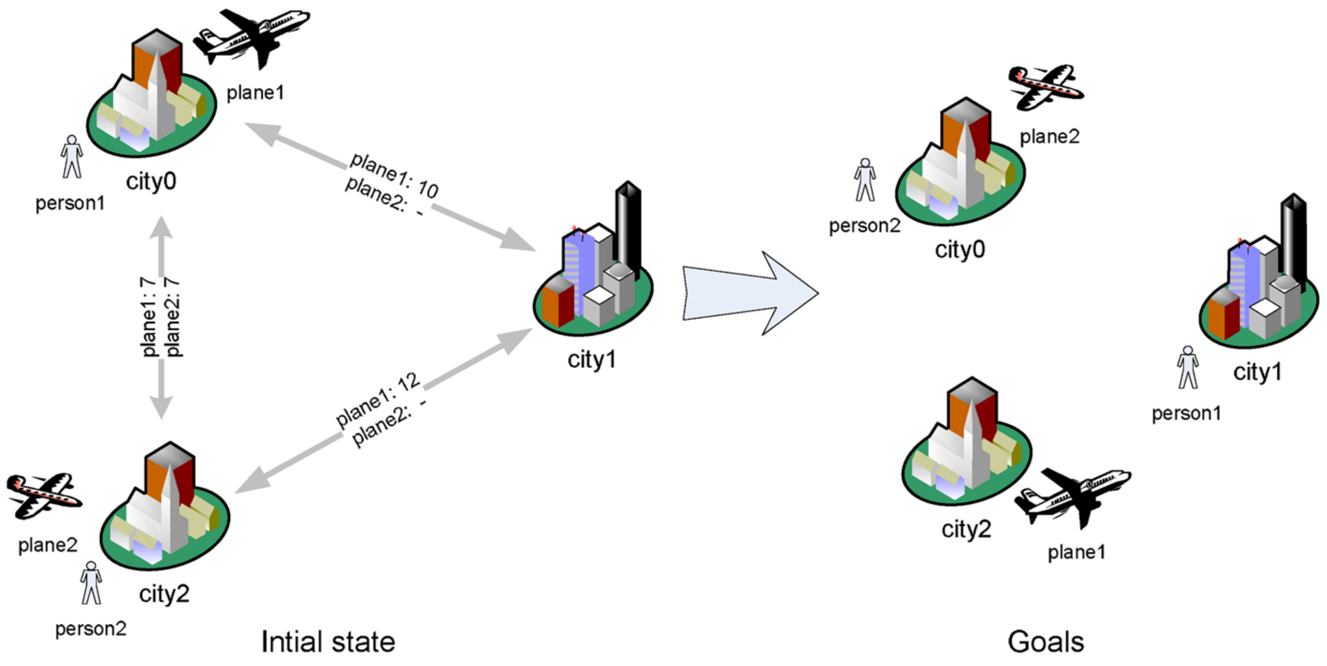


Fig. 2 Domain for the applications example with the initial state, the goals, and the duration of actions *fly* for each plane

5 An example of application

In this section we present a complete example of application to show the scope of our CP formulation. We will start with a simple problem and gradually include extra features, until generating a complex P&S problem. Our main interest with this example is to show the expressiveness of the model and check the validity of the obtained plans rather than solving large-size problems that involve many variables and constraints. Actually, we do not put special emphasis on performance as this issue is more concerned with the performance of the CSP solver, which in turn highly depends on the implemented heuristics. We first present the basic problem along with its formulation in Sect. 5.1 and then, in Sect. 5.2, the incremental extensions that represent more complex constraints.

5.1 Domain and problem. The basics

For the application example we have chosen a logistics domain, traditionally used in planning, in which the goal is to transport people among cities with the minimum makespan using different planes. Figure 2 depicts the problem domain, the initial state, and the problem goals. The scenario is composed of three cities ($city0$, $city1$, and $city2$), two planes ($plane1$ and $plane2$) and two people ($person1$ and $person2$). We assume that $plane1$ can fly between any pair of cities, whereas $plane2$ can only fly between $city0$ and $city2$, and back. Both $person1$ and $plane1$ are initially located at $city0$, and $person2$ and $plane2$ are in $city2$. In the goal situation, there are four goals: $person2$ and $plane2$ must be in $city0$, $person1$ must be in $city1$, and $plane1$ must reach $city2$. In order to keep the example simple enough, we do not allow people to transfer between planes, i.e. only one *board*/*debark* is allowed by a person, which implies that the whole route must be done using the same plane. The actions in the domain follow the template (operator schema in planning terminology) shown in Table 5. For each operator we include its conditions/effects and the time points when they are required/generated. Assuming that the duration[13] for all actions *board* is 3, for *debark* is 2, and for *fly* is 7, 10, or 12, depending on the cities and the plane as shown in Fig. 2, the plan with the optimal makespan, i.e. that with the minimum duration, is shown in Fig. 3. Basically, the optimal plan of duration 29 consists of two parallel branches as independent sequences of actions, one shorter for transporting $person2$ with $plane2$ and the other longer for transporting $person1$ with $plane1$ and flying $plane1$ to $city2$.

CP formulation of the problem

According to the representation described in Sect. 3, the formulation of the model must include the corresponding variables and constraints for all actions in the problem. For instance, the formulation of the variables and constraints for the action $board(person1, plane1, city0)$ are given in Tables 6 and 7, respectively. The variables

[13] We have used integer domains because real variables are still under development in Choco.

Table 5 Operator schema of the domain actions. $S()$ and $E()$ stand for the start and end point, respectively, of the operator

Name	Conditions	Effects
fly$(?plane, ?origin, ?destination)$	$at(?plane, ?origin) : S()..S()$	$\neg at(?plane, ?origin) : S()$ $at(?plane, ?destination) : E()$
board$(?person, ?plane, ?city)$	$at(?plane, ?city) : S()..E()$ $at(?person, ?city) : S()..S()$	$\neg at(?person, ?city) : S()$ $in(?person, ?plane) : E()$
debark$(?person, ?plane, ?city)$	$at(?plane, ?city) : S()..E()$ $in(?person, ?plane) : S()..S()$	$\neg in(?person, ?plane) : S()$ $at(?person, ?city) : E()$

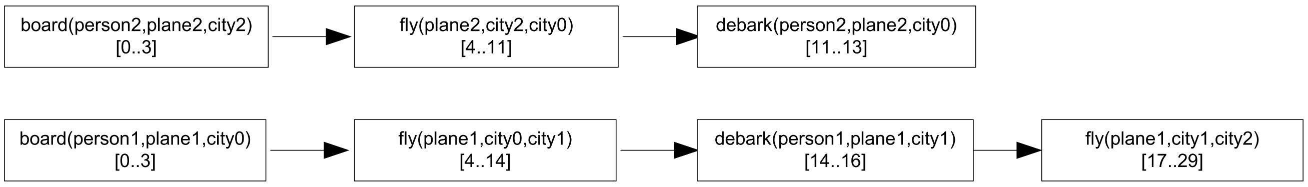

board(person2,plane2,city2) [0..3]
fly(plane2,city2,city0) [4..11]
debark(person2,plane2,city0) [11..13]
board(person1,plane1,city0) [0..3]
fly(plane1,city0,city1) [4..14]
debark(person1,plane1,city1) [14..16]
fly(plane1,city1,city2) [17..29]

Fig. 3 Optimal makespan plan for the application example

Table 6 Variables generated for the action *board*$(person1, plane1, city0)$. Note that the domain of variables $S()$, $E()$ can be reduced by applying a reachability-based heuristic estimate that calculates with more precision the earliest start time of any action. For instance, the earliest time for action *fly*$(plane1, city1, city2)$ is 10 as $plane1$ first needs to reach $city1$, which requires at least 10 time units

Variables
$S(board(person1, plane1, city0)), E(board(person1, plane1, city0)) \in [0, \infty[$
$dur(board(person1, plane1, city0)) \in [3, 3]$
$InPlan(board(person1, plane1, city0)) \in [0, 1]$
$Sup(at(plane1, city0), board(person1, plane1, city0)) \in \{Start, fly(plane1, city1, city0), fly(plane1, city2, city0\}$
$Sup(at(person1, city0), board(person1, plane1, city0)) \in \{Start, debark(person1, plane1, city0)\}$
$Time(at(plane1, city0), board(person1, plane1, city0)) \in [0, \infty[$
$Time(at(person1, city0), board(person1, plane1, city0)) \in [0, \infty[$
$Req_{start}(at(plane1, city0), board(person1, plane1, city0)) = S(board(person1, plane1, city0))$
$Req_{end}(at(plane1, city0), board(person1, plane1, city0)) = E(board(person1, plane1, city0))$
$Req_{start}(at(person1, city0), board(person1, plane1, city0)) = S(board(person1, plane1, city0))$
$Req_{end}(at(person1, city0), board(person1, plane1, city0)) = S(board(person1, plane1, city0))$

and constraints for actions $board(?person_i, ?plane_j, ?city_k)$, $debark(?person_i, ?plane_j, ?city_k)$, and $fly(?plane_i, ?origin_j, ?destination_k)$ are generated similarly.

5.2 Extending the application example with complex constraints

In this section we take the previous problem as a basis and incrementally extend it with more complex constraints, thus creating new scenarios of application. Particularly, we present five tests that correspond with five slightly different scenarios.

Test 1 corresponds to the basic problem, and each new test gradually introduces new constraints over the previous one. We solve each test under two perspectives: (i) computing a plan from scratch, i.e. when the solver acts as a planner + scheduler; and (ii) scheduling a given plan, i.e. acting only as a scheduler. In the former, the domain of all variables are open and the solver needs to find a feasible solution to the variables. In the latter, the values for the *InPlan* variables are set either to 0 or 1 according to the plan found in the P&S execution, the values for the *Sup*-variables are left unbound and the solver needs to find an appropriate time and resource allocation for the actions. In both cases, the Choco solver was run to *optimise* the plan's makespan (with no initial upper bound), instead of just finding one solution. This

Table 7 Constraints generated for the action *board*(*person*1, *plane*1, *city*0)

Constraints
$S(board(person1, plane1, city0)) + dur(board(person1, plane1, city0)) = E(board(person1, plane1, city0))$
$E(Start) \leq S(board(person1, plane1, city0))$
$E(board(person1, plane1, city0)) \leq S(End)$
$Time(at(plane1, city0), board(person1, plane1, city0)) \leq Req_{start}(at(plane1, city0), board(person1, plane1, city0))$
$Time(at(person1, city0), board(person1, plane1, city0)) \leq Req_{start}(at(person1, city0), board(person1, plane1, city0))$
if $Sup(at(plane1, city0), board(person1, plane1, city0)) = Start$ then
$InPlan(Start) = 1$
$Time(at(plane1, city0), board(person1, plane1, city0)) = E(Start) = 0$
if $Sup(at(plane1, city0), board(person1, plane1, city0)) = fly(plane1, city1, city0)$ then
$InPlan(fly(plane1, city1, city0)) = 1$
$Time(at(plane1, city0), board(person1, plane1, city0)) = E(fly(plane1, city1, city0))$
if $Sup(at(plane1, city0), board(person1, plane1, city0)) = fly(plane1, city2, city0)$ then
$InPlan(fly(plane1, city2, city0)) = 1$
$Time(at(plane1, city0), board(person1, plane1, city0)) = E(fly(plane1, city2, city0))$
if $Sup(at(person1, city0), board(person1, plane1, city0)) = Start$ then
$InPlan(Start) = 1$
$Time(at(person1, city0), board(person1, plane1, city0)) = E(Start) = 0$
if $Sup(at(person1, city0), board(person1, plane1, city0)) = debark(person1, plane1, city0)$
$InPlan(debark(person1, plane1, city0)) = 1$
$Time(at(person1, city0), board(person1, plane1, city0)) = E(debark(person1, plane1, city0)$

means that Choco needs to perform an exhaustive search of solutions until no feasible solution can improve the quality of the best solution found until that time. We ran all the tests on a Pentium IV 2.60 GHz. with 1280 Mb of RAM and limited the search to 300 seconds, though in none of the tests this limit was finally reached.

Each test was solved with two different strategies. **DEF** is the DEFault strategy used by Choco, which selects the variable with the minimum domain and the values of the domain are selected in ascending order. **HEUR** applies the two branching heuristics presented in Sect. 4.2 and the reachability heuristic introduced in Sect. 4.3. Our aim is to check the impact of the application of simple and general heuristics that can be used in any standard CSP solver. The results of the five tests for each of these strategies are shown in Tables 8 and 9. Table 8 shows the results in terms of execution time, number of nodes explored, and the makespan of the plan when generating a plan from scratch, whereas Table 9 shows the same results when scheduling a plan. The input plans that used to fix the values of the *InPlan*-variables of the tests in Table 9 are the plans obtained from the tests in Table 8. In both tables, for each test and strategy, we show different solutions in the same order in which they were found. For instance, in Test 1 with the DEF strategy the first solution, with makespan 39, was found in 31.8 seconds and after exploring 1611 nodes. The second solution returned a plan of makespan 29 and needed 32 seconds; the third row in this test shows the solver needed 32.3 seconds to explore 3148 nodes until exhausting the search space, thus guaranteeing that the optimal solution is the one of duration 29.

The resolution of Test 1 in Table 8 shows that the HEUR strategy behaves significantly better than DEF, both in execution time and number of explored nodes. As can be seen, the HEUR strategy finds the first solution very fast, though its quality is not as good as the first solution found by DEF. However, the optimal solution is later found by HEUR in much less time than DEF (see Table 8). When the solver is scheduling a plan, the results are very similar with the two strategies (see Table 9).

Test 2 (Conditions and effects out of the action interval) In this test we include the following constraint: condi-

Table 8 Results of the five tests when generating a plan from scratch in format 'execution time in seconds/nodes explored (makespan)'. The symbol '–' means that no better solution was found, whereas '∄' means that no feasible solution was found

Test	**DEF**	**HEUR**
Test 1	31.8/1611 (39) 32.0/2103 (29) 32.3/3148 (–)	0.1/31 (45) 0.2/62 (29)
Test 2	23.4/1531 (42) 23.6/2016 (32) 23.9/3153 (–)	0.1/29 (48) 0.2/60 (32)
Test 3a	11.5/18347 (∄)	16.5/21274 (∄)
Test 3b	94.2/82839 (55) 94.3/82866 (51) 98.3/89717 (–)	119/85235 (55) 135.5/104170 (51) 139.4/109221 (–)
Test 4a	50.1/71149 (42) 50.4/72279 (–)	0.1/28 (57) 58.2/25533 (42) 58.6/26574 (–)
Test 4b	0.5/708 (46) 0.7/1454 (–)	0.1/40 (46)
Test 5	48.2/75043 (37) 120.8/238120 (–)	9.2/10125 (37) 269.9/422882 (–)

Table 9 Results of the five tests when scheduling a given plan in format 'execution time in seconds/nodes explored (makespan)'. The symbol '–' means that no better solution was found, whereas '∄' means that no feasible solution was found

Test	**DEF**	**HEUR**
Test 1	0.1/12 (29) 0.1/12 (–)	0.1/20 (29)
Test 2	0.1/12 (32) 0.1/12 (–)	0.1/20 (32)
Test 3a	0.25/92 (∄)	0.3/135 (∄)
Test 3b	0.2/43 (55) 0.3/70 (51) 0.4/118 (–)	0.2/29 (55) 0.3/49 (51)
Test 4a	0.1/20 (42) 0.1/20 (–)	0.1/20 (42) 0.1/20 (–)
Test 4b	0.1/19 (46) 0.1/19 (–)	0.1/19 (46)
Test 5	0.1/20 (37) 0.1/22 (–)	0.1/20 (37) 0.1/20 (–)

tion $at(?person, ?city)$ for boarding must happen before the action *board* starts. Additionally, we make the effects $at(?person, ?city)$ and $at(?plane, ?destination)$ of actions *debark* and *fly*, respectively, to be generated beyond the end time of the actions. These situations are simple but very common. For instance, a person needs to be at the airport some time before/after boarding/debarking to pass security controls, or the plane needs some time to traverse the airport. In a classical planning representation, this would imply creating new dummy actions to model the delay, but in our model this can be done by simply modelling the three constraints shown in Table 10, where the values for $time_{before}$ and $time_{after}$ represent the corresponding delays. In this test we have set:

$$time_{before} = time_{after} = 1,$$

which changes the makespan of the optimal plan to 32 as depicted in Fig. 4.

In this test, the use of the HEUR strategy proves again to be more efficient than DEF. Particularly, the execution time and the number of nodes explored to find an optimal solution

Table 10 Additional constraints for Test 2. $time_{before}$ and $time_{after}$ represent the corresponding delays

Constraints
$Req_{start}(at(person_i, city_j), board(person_i, plane_k, city_j)) = S(board(person_i, plane_k, city_j)) - time_{before}$
if $Sup(at(person_i, city_j), board(person_i, plane_k, city_j)) = debark(person_i, plane_l, city_j)$ then $InPlan(debark(person_i, plane_l, city_j)) = 1$ $Time(at(person_i, city_j), board(person_i, plane_k, city_j)) = E(debark(person_i, plane_l, city_j)) + time_{after}$
if $Sup(at(plane_i, city_j), a_k) = fly(plane_i, city_l, city_j)$ then $InPlan(fly(plane_i, city_l, city_j)) = 1$ $Time(at(plane_i, city_j), a_k) = E(fly(plane_i, city_l, city_j)) + time_{after}$

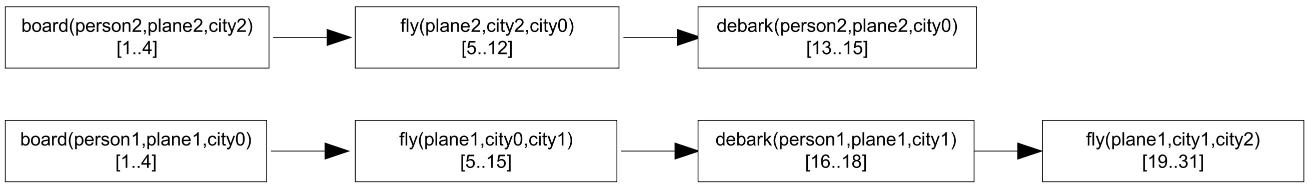

board(person2,plane2,city2) [1..4]
fly(plane2,city2,city0) [5..12]
debark(person2,plane2,city0) [13..15]
board(person1,plane1,city0) [1..4]
fly(plane1,city0,city1) [5..15]
debark(person1,plane1,city1) [16..18]
fly(plane1,city1,city2) [19..31]

Fig. 4 Optimal makespan plan for the example of Test 2

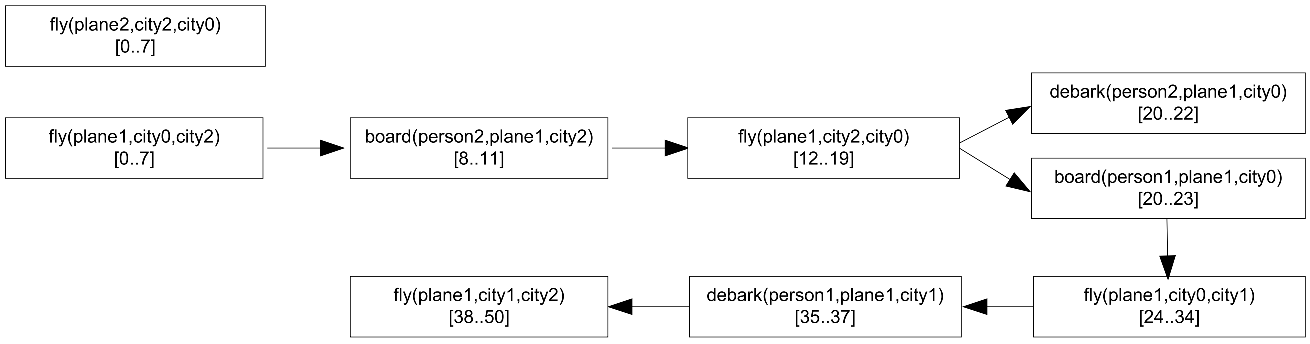

fly(plane2,city2,city0) [0..7]
fly(plane1,city0,city2) [0..7]
board(person2,plane1,city2) [8..11]
fly(plane1,city2,city0) [12..19]
debark(person2,plane1,city0) [20..22]
board(person1,plane1,city0) [20..23]
fly(plane1,city0,city1) [24..34]
debark(person1,plane1,city1) [35..37]
fly(plane1,city1,city2) [38..50]

Fig. 5 Optimal makespan plan for the example of Test 3

using HEUR are better than the values required to find the first solution when using DEF (see Table 8). On the other hand, when the task is plan-scheduling the results are very similar (see Table 9).

Test 3 (Deadlines) Test 3a extends Test 2 to manage several goal deadlines that force a change in the plan. We have included the deadline constraint $Time(at(plane2, city0), End) \leq 10$. This obliges *plane*2 to directly fly from *city*2 to *city*0 with no time for boarding *person*2. Therefore, *person*2 must use *plane*1 to fly, thereby producing a longer plan of 51 as the optimal makespan (see Fig. 5). In addition to this, we have included the deadline on the plan makespan $End < 50$. Due to this strict constraint the problem has no solution. As shown in Table 8, the CSP finds no solution after a few seconds, and the same happens in Table 9, where the input plan is the optimal one without the $End < 50$ deadline. In this test, the DEF heuristic is slightly better because the deadline makes the domain of the variables smaller and the minimum domain selection criterion is more efficient, though not in a very significant way. Instead of using this deadline, if we post a feasible deadline, such as $End < 60$ (Test 3b), the results are similar. When imposing a tight deadline, the DEF strategy proves to be a bit more efficient than HEUR, particularly in the number of nodes explored in the plan generation experiments.

Test 4 (Temporal constraints between actions) Test 4a removes the constraint $Time(at(plane2, city0), End) \leq 10$ of Test 3 to work with the original plan of Test 2, but it keeps the $End < 60$ deadline and include new constraints between actions. First, we state that *plane*2 must arrive at *city*0 some

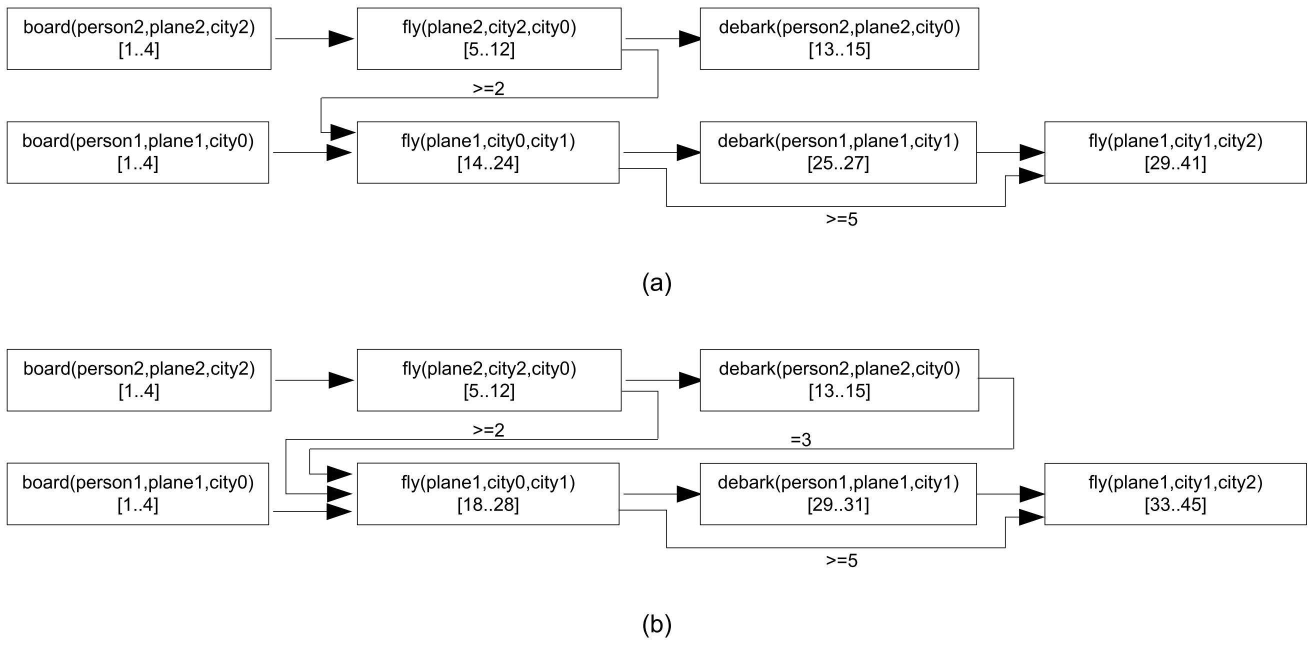

board(person2,plane2,city2) [1..4]
fly(plane2,city2,city0) [5..12]
debark(person2,plane2,city0) [13..15]
>=2
board(person1,plane1,city0) [1..4]
fly(plane1,city0,city1) [14..24]
debark(person1,plane1,city1) [25..27]
fly(plane1,city1,city2) [29..41]
>=5
(a)
board(person2,plane2,city2) [1..4]
fly(plane2,city2,city0) [5..12]
debark(person2,plane2,city0) [13..15]
>=2
=3
board(person1,plane1,city0) [1..4]
fly(plane1,city0,city1) [18..28]
debark(person1,plane1,city1) [29..31]
fly(plane1,city1,city2) [33..45]
>=5
(b)

Fig. 6 Optimal makespan plan for the two examples of Test 4: Test 4a is the original test, and Test 4b includes a synchronisation constraint

time before *plane*1 leaves (quantitative temporal constraint). This constraint represents a particular ordering constraint when the cabin crew are the same for the two planes. Second, we state that it is necessary for some time to elapse between two consecutive *fly* actions of the same plane in order to perform maintenance tasks or to give a rest to the crew. The new two constraints included in the model are:

$$E\big(fly(plane2, city2, city0)\big) + time_{before} \le S\big(fly(plane1, city0, city_i)\big),$$
$$E\big(fly(plane_i, city_j, city_k)\big) + time_{consecutive} \le S\big(fly(plane_i, city_k, city_l)\big),$$

where $time_{before}$ and $time_{consecutive}$ represent the delays. In this test we have set $time_{before} = 2$ and $time_{consecutive} = 5$.

Additionally, we allow actions to have variable duration, i.e. instead of being a fixed value it can vary within an interval. Thus, the optimal makespan for Test 4a is 42, as represented in Fig. 6. HEUR finds the first sub-optimal solution very fast but it requires more time to find the optimal one. Again, the restrictive deadline is not beneficial for the HEUR strategy as it is focused on the action supports and not on the length of the variables' domains. As in other tests, there is no difference at all for the scheduling scenario.

Finally, Test 4b includes a synchronisation constraint: the start time of action $fly(plane1, city0, city1)$ must be exactly synchronised with the end of $debark(person2, plane2, city)$ plus three time units (for instance, for security reasons in the airport). Hence, we post the next constraint:

$$S\big(fly(plane1, city0, city1)\big) = 3 + E\big(debark(person2, plane2, city0)\big).$$

The new optimal plan has makespan 46, as depicted in Fig. 6. In this case, both DEF and HEUR behave similarly but the number of nodes explored by HEUR is considerably smaller in the plan-generation scenario.

Test 5 (Numeric variables) This test extends the original description of the application example given in Test 1 by including a fluent to model the fuel consumption of each plane. This way, the CSP can manage the resource constraints associated with conditions and effects of the actions *fly*. Additionally, new actions *refuel* are generated to act as fuel producers. The new variables included in the model are given in Table 11. The model also needs to include constraints to bind variables $S()$, $E()$, and $dur()$ for actions $refuel(plane_i, city_j)$, together with the constraints between $Sup()$, $Time()$, and $Req_{start}()$.

Additionally, this test also includes temporal windows constraints. Particularly, the *refuel* actions can only be executed in the disjunctive intervals [10, 15] or [20, 25]. This type of constraint turns very useful to encode timetable constraints by simply posting:

$$\text{if } (InPlan(refuel(plane_i, city_j)) = 1) \text{ then}$$
$$10 \le S(refuel(plane_i, city_j)) \le E(refuel(plane_i, city_j)) \le 15 \lor$$
$$20 \le S(refuel(plane_i, city_j)) \le E(refuel(plane_i, city_j)) \le 25.$$

The optimal makespan for this test is 37, since the plan needs the action *refuel(plane1, city1)*, as indicated in Fig. 7. This is the most complex test as it involves many variables to

Table 11 Additional variables for Test 5

Variables
$S(refuel(plane_i, city_j)), E(refuel(plane_i, city_j)) \in [0, \infty[$
$dur(refuel(plane_i, city_j))$. In this test the variable duration for *refuel* is [4, 5] for *plane*1 and [2, 3] for *plane*2
$InPlan(refuel(plane_i, city_j)) \in [0, 1]$
$Sup(at(plane_i, city_j), refuel(plane_i, city_j)) \in \{fly(plane_i, city_k, city_j)\}$. Note that the supporter can be also *Start* if the $plane_i$ is in $city_j$ in the initial state
$Sup(fuel(plane_i), refuel(plane_i, city_j)) \in \{fly(plane_i, city_k, city_j), refuel(plane_i, city_l), Start\}$
$Sup(fuel(plane_i), fly(plane_i, city_j, city_k)) \in \{fly(plane_i, city_l, city_j), refuel(plane_i, city_j), Start\}$
$Time(at(plane_i, city_j), refuel(plane_i, city_j)) \in [0, \infty[$
$Time(fuel(plane_i), refuel(plane_i, city_j)) \in [0, \infty[$
$Time(fuel(plane_i), fly(plane_i, city_j, city_k)) \in [0, \infty[$
$Req_{start}(at(plane_i, city_j), refuel(plane_i, city_j)) = S(refuel(plane_i, city_j))$
$Req_{end}(at(plane_i, city_j), refuel(plane_i, city_j)) = E(refuel(plane_i, city_j))$
$Req_{start}(fuel(plane_i), refuel(plane_i, city_j)) = S(refuel(plane_i, city_j))$
$Req_{end}(fuel(plane_i), refuel(plane_i, city_j)) = S(refuel(plane_i, city_j))$
$Req_{start}(fuel(plane_i), fly(plane_i, city_j, city_k)) = S(fly(plane_i, city_j, city_k))$
$Req_{end}(fuel(plane_i), fly(plane_i, city_j, city_k)) = S(fly(plane_i, city_j, city_k))$

encode the metric information. The main drawback in problems that involve numeric variables is the high number of supports that appear for the fluents. Let us revisit the domain of *Sup*-variables for the fluent $fuel(plane_i)$ in Table 11. In this case, any *fly* or *refuel* action using $plane_i$ could potentially support $fuel(plane_i)$ as they modify the fuel value. This is particularly problematic during the search because it entails a huge branching factor. Our branching heuristics for variable and value selection help reduce this problem by achieving a more intelligent branching. On the one hand, the variable selection gives priority to the propositional *Sup*-variables *vs.* the numeric *Sup*-variables. On the other hand, the value selection tries first to reuse the actions in the plan rather than including new ones. Despite these heuristics, guaranteeing an optimal plan from scratch in this type of test is a hard task and it requires a high number of nodes to be explored (see Table 8). An interesting result here is that computing the first solution, though it may be sub-optimal, is much faster with the HEUR strategy than with DEF. As usual, in the scheduling scenario there are no differences between DEF and HEUR, as seen in Table 9.

Final remarks

These tests show that our model can be used to encode a P&S problem, from purely propositional problems to more elaborate problems where the representation mixes propositional and numeric information together with complex constraints, such as qualitative and quantitative orderings, deadlines, temporal windows, etc.

The fact of using a standard, general-purpose CSP engine rather than a specific engine for planning search makes the resolution of the CP formulation a hard task, which becomes even harder in large-size problems. Despite this, the use of general branching and pruning heuristics in HEUR provides important benefits. The HEUR strategy is notably better, both *w.r.t.* execution time and search cost, in most of the tests of the plan generation scenario. In some cases, the first solution is not as good as the first solution in DEF, but in the end HEUR reaches the same quality in less time than DEF. However, HEUR is not so beneficial when it is necessary to exhaust the whole search space or when the problem contains strict deadlines which shrink the variables' domain. In this case, the DEF strategy proves to be very helpful. More specifically, the worst results appear when trying to solve an unfeasible problem, such as Test 3a, where the CSP solver needs to exhaust the whole search space before guaranteeing that no possible solution exits. But even in that type of situation, the CP formulation and the constraint propagation can guarantee that the problem is unsolvable. Obviously, in much more complex problems, with several thousands of variables and constraints, the use of HEUR can significantly improve the resolution process.

On the whole, we can conclude that the HEUR strategy provides a simple but good guidance for building up the logical structure of the plan. On the other hand, in the scheduling scenario both strategies are equally efficient. The experiments evidence that the hardest task in a P&S problem is to find the logical structure of the plan: once the set of actions

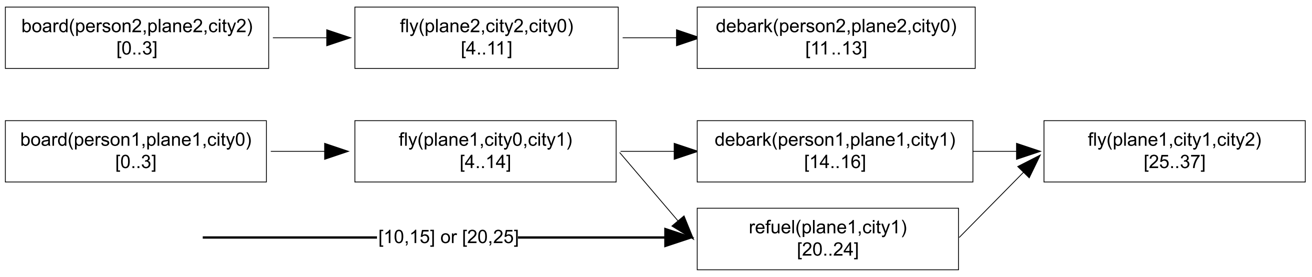


Fig. 7 Optimal makespan plan for the example of Test 5

of the plan is known, the CSP only has to set the appropriate supports to find a feasible time allocation for actions.

After all, the really difficult task, which is to find out the actions that form the plan, is given as an input, and the CSP solver only needs to find the adequate supports that lead to a feasible allocation time for the actions in the plan.

6 Experimental results

In this section we present some experimental results to analyse the scalability of our CP formulation in comparison with some state-of-the-art planners. In order to do this comparison we have selected some planning domains of the last International Planning Competition (IPC–5). But before doing this, it is important to highlight the following basic principles:

– As commented in the previous section, the fact that we are not working with a customised search engine turns out to be inconvenient for achieving improvements in the overall system performance. Our resolution process is subject to Choco's engine, so we cannot incorporate POCL-planning heuristics for guiding flaw selection as in CPT (Vidal and Geffner 2006), planning decomposition and subgoal partitioning techniques as in SGPlan (Chen et al. 2006), or best-first heuristic search as in MIPS-XXL (Edelkamp et al. 2006)—these three are the planners chosen for our comparison.[14] We cannot extend the comparison with the optimal planners that participated in IPC–5 as they do not accept temporal domains.
- The planning domains used in the last planning competitions are remarkably complex, even for small instances, and solving them with a standard, general-purpose CSP engine as the one implemented in Choco is a great challenge. Choco's solver makes no use of planning heuristics techniques and the only intelligent behaviour implemented in Choco is due to the branching and pruning heuristics presented in Sects. 4.2 and 4.3, respectively. Therefore, all our experiments were run with the HEUR strategy.
- Most problems' solutions are plans which contain several occurrences of the same action. Since our CP formulation does not handle the canonicity restriction we have limited the experiments to a few problems of the domains `rovers`, `pipesworld`, `storage`, and `openstacks`.

We have structured this section in two blocks. The first block deals with the problems of the `rovers`, `pipesworld`, and `storage` domains. The second block presents some experiments with problems from the `openstacks` domain, which has some particular properties that make it appropriate for a CP formulation. In this block we present an additional experiment where we include some precedence and deadline constraints in the problems. All tests were run on a Pentium IV 2.60 GHz with 1280 Mb of RAM and censored after 600 seconds.

6.1 Results for generic planning domains

In this section we focus on a few problems of the `rovers`, `pipesworld`, and `storage` domains. For the reasons mentioned above, we have limited our experiments to the following 11 problems from the *Time* or *MetricTime* track: 1–4 for `rovers`, 3–4 for `pipesworld`, and 1–3, 5–6 for `storage`. We have used SGPlan, MIPS-XXL, and CPT for comparison. CPT was not able to solve the problems of the `rovers` domain because it does not support metric features.

Figure 8 shows the results when solving the original 11 problems, i.e. in a plan generation scenario. Since they are small problem instances, all the planners computed the same plan. First remark on the results is that our running times are much higher than in other planners, specially in comparison to MIPS-XXL, which shows the best performance. This is an expected result when comparing a nearly blind search process with outstanding heuristic planners. Additionally, Choco cannot solve the two problems of the `pipesworld` domain because the initial CSP stage of constraint posting

[14]CPT: http://www.cril.univ-artois.fr/~vidal/#cpt. SGPlan v5.2.2: http://manip.crhc.uiuc.edu/programs/SGPlan. MIPS-XXL: http://ls5-web.cs.uni-dortmund.de/~edelkamp/mips/mips-xxl.html.

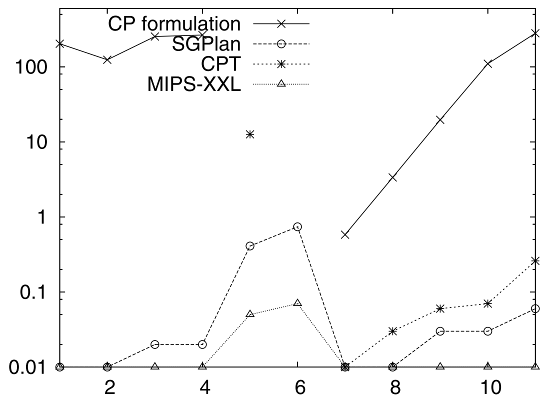


Fig. 8 Plan generation for the 11 problems. Execution time is shown in seconds (log scale in the y-axis)

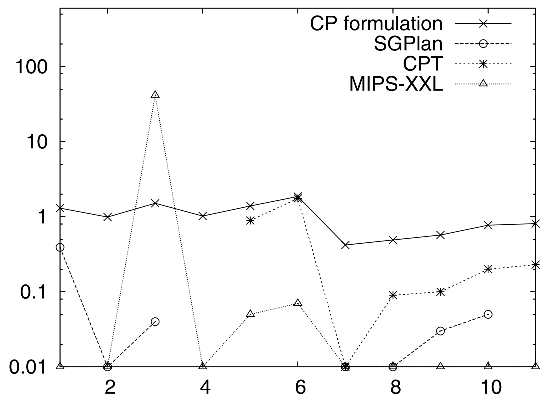


Fig. 9 Plan scheduling for the 11 problems. Execution time is shown in seconds (log scale in the y-axis)

exceeds the time given. CPT has a similar problem with the second problem of `pipesworld` and cannot find a solution in 600 seconds.

The main conclusion that we can extract from these results is that our CP formulation cannot be *directly* used for general P&S problems unless a specific search engine is implemented. In order to be competitive with other planners a specific planner that manages the CP formulation becomes necessary, similarly to how CPT is designed.

The second experiment compares our CP formulation when dealing with a scheduling scenario. We simulate a scheduling behaviour in the three planners by creating dummy effects attached to the actions present in the plan and defining them as top-level goals. This way, the planners are forced to insert these actions without having to consider alternative supports and/or include additional actions.

The results are depicted in Fig. 9. The performance of our CP formulation is much better in this scenario and it shows scalable solutions; it solves all the problems, finding the best allocation times for actions in about one second or less. However, the three planners do not show such an scalable behaviour (see, for instance, the third problem solved by MIPS-XXL). Surprisingly, SGPlan and MIPS-XXL returned worse-quality plans for some of the problems (plans with irrelevant actions for achieving the goals),[15] and more time-consuming responses than in the plan generation scenario.

Finally, we have tested our CP formulation in a pseudo-planning scenario (see Fig. 10). For this experiment, we generate five new models which comprise the actions that are present in the plan. The different tests range from a 0%-model to a 100%-model according to the number of fixed actions in the model; 0% stands for planning generation scenario, i.e. no actions are fixed, whereas 100% stands for a scheduling scenario, i.e. all the actions of the plan are explicitly encoded in the model. Obviously, as in Fig. 8, we

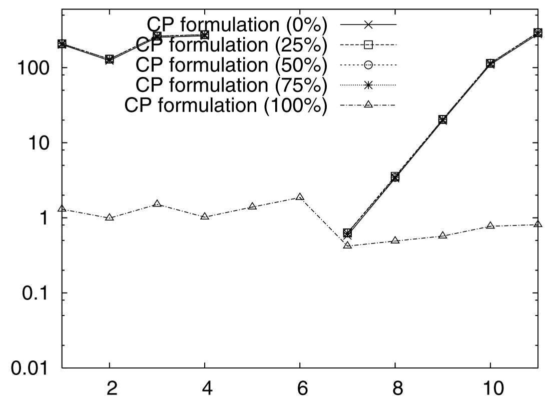
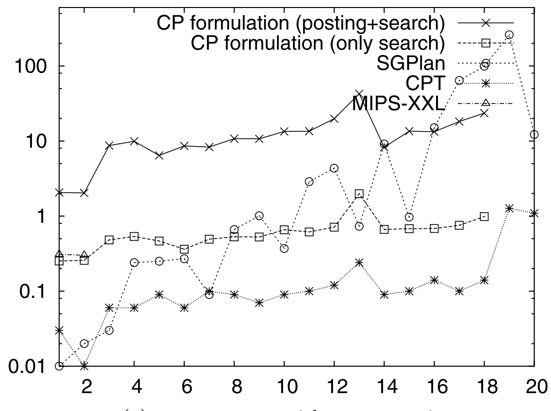


Fig. 10 Pseudo-planning for the 11 problems. Execution time is shown in seconds (log scale in the y-axis)

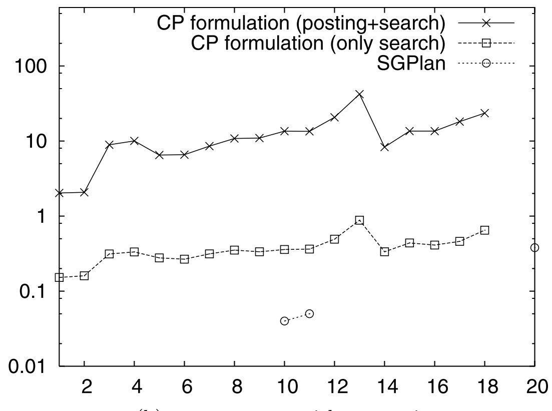
cannot solve the two problems of the `pipesworld` domain unless the solver is working in a scheduling scenario. The main conclusion here is that planning is a very costly task regardless the number of actions that are initially input. If the problem involves some planning effort, even being this effort limited, the complexity is asymptotically very high. In the figure, we can see the results for the 25%, 50%, and 75% models do not show any differences.

All in all, we can conclude that a standard CSP engine is not appropriate for solving plan generation problems and that more intelligent techniques to guide search are required. We have noticed that the complexity of the resolution process does not really come from the model expressiveness range but from the inherent complexity of a blind planning process. On the contrary, when dealing with scheduling scenarios a simple CSP solver proves to be competitive enough. If the formulation is used for solving problems in a pseudo-planning scenario then the complexity is asymptotically analogous to solving plan generation problems.

[15] SGPlan exits with a segmentation fault in 4 of the problems.



(a) - `openstacks` with no constraints



(b) - `openstacks` with constraints

Fig. 11 Execution time (in seconds) for the `openstacks` domain. Please note the log scale in the y-axis

6.2 Results in combinatorial optimisation problems. The `openstacks` domain

The `openstacks` domain represents a combinatorial optimisation problem in production scheduling which comes from the CSP benchmarks. We have chosen this domain because it does not require a high planning effort, so it is a good sample to check the behaviour of our CP formulation. The overall idea is to have a number of orders, each for a combination of different products (e.g. o1: p1, p2, and p4; o2: p1 and p3, and so on), that must be shipped. In the IPC–5 there are different versions of this domain, all including a 'stack' (a temporary storage space) to make the products, and the problem consists in minimising the number of stacks that are simultaneously open. We have slightly modified the temporal version to have only three operators, start-order, make-product and ship-order. The objective is now to ship all orders as soon as possible, thus minimising the makespan. In short, finding a plan is now a trivial operation as every ordering of the making of products is a solution.

The resolution of our CP formulation by Choco requires two stages: posting the constraints of the model and the search stage. Obviously, the two stages are indispensable in the resolution process but we include in the experiments of this section two values: the time required for posting + search, i.e. the whole process, and the time required only for search.

In our first experiment, we address the scalability of our CP formulation in the 20 problems of the `openstacks` domain with no additional constraints. The results (execution times) are shown in Fig. 11(a). All planners returned the optimal solution, which in this domain represents the optimal schedule. As can be seen, the best times are offered by CPT and SGPlan. Surprisingly, MIPS-XXL only solved the two smallest problems. The times for posting + search are above the times of the three planners, though they are slightly better than SGPlan in two of the problems. The most interesting result is that the execution time remains quite constant and scalable, particularly *w.r.t.* the search time. Actually, a remarkable result is that the time spent in doing real search is less than 1 second in all but one problem. Our model was not able to solve the problems 19 and 20 within the given time. The real bottleneck here is the time for posting the constraints, which is nearly 20 minutes, as the time for search is less than 100 seconds. Generally, time for posting represents an important overhead in the resolution process because of the high number of variables and constraints in the model. In order to have an approximate idea about the size of the generated model for this domain, Table 12 shows the number of actions, variables, and constraints for each problem.

In our second experiment (see Fig. 11(b)), we include precedence constraints between actions and deadlines on goals. In our CP formulation these constraints are easily modelled. As for the PDDL3.0 planners, we used fictitious effects and a combination of `within` and `sometime-after` constraints. Surprisingly, SGPlan could only solve three of these problems and returned no response for the others. Our impression is that SGPlan does not exhibit a good behaviour because the goal decomposition fails to find a feasible solution for problems with restrictive constraints. The available version of MIPS-XXL failed parsing the PDDL3.0 constraints tag, so we could not include it in our experiment. The behaviour of our CP formulation remains quite stable and nearly identical to the original problem with no constraints. Actually, when dealing with additional constraints the time used for search is slightly better, but due to the overhead of the posting stage the overall time is very similar.

As can be seen from this experiment, the stable behaviour of our formulation remains when the problem contains additional constraints that limit the number of valid plans. Actually, this is a favourable scenario for our CP formulation but problematic for current planners, such as SGPlan. On the other hand, it must also be said that one of the main reasons for this good behaviour relies on the low planning load of

Table 12 Number of actions for each of the 20 problems of the `openstacks` domain and number of variables and constraints of the corresponding CP formulation. Note that the plans contain all actions of the problem

Problem	**Actions**	**Variables**	**Constraints**
1	30	527	1082
2	30	527	1082
3	55	1227	2617
4	55	1307	2807
5	60	1047	2162
6	60	1047	2162
7	60	1207	2542
8	60	1367	2922
9	60	1367	2922
10	60	1527	3302
11	60	1527	3302
12	70	1871	4069
13	110	2771	6004
14	75	1227	2512
15	90	1567	3242
16	90	1567	3242
17	90	1807	3812
18	90	2047	4382
19	150	14 055	32 591
20	150	12 191	28 164

this domain. Actually, the planning activities in these problems can be solved via propagation of the *Sup*-variables, without really requiring search. This way, the search stage is only dedicated to find the right schedule that satisfies all the problem constraints. This can be also appreciated in the good performance of CPT in Fig. 11(a), even better than SGPlan. This clearly demonstrates that using a pure inference mechanism can be very valuable in some planning domains where a lot of planning can be done without really planning, but only by reasoning and inferring causal information.

7 Conclusions

In this paper we have presented a general formulation to encode a P&S problem as a CP formulation that is later solved by a CSP solver. Our formulation is able to model scheduling, planning, and pseudo-planning problems. This way the model is not limited to complete input plans but it can also work with semi-complete plans or empty plans. The formulation is highly influenced by the work of CPT and, consequently, it has the ability to reason about supports, mutex relations, and causal links in a very elaborate model of actions. Several extensions have been discussed in the paper: (i) a formulation under a very expressive temporal model of actions; (ii) the modelling of additional constraints; and (iii) the formulation of numeric variables for resource representation and multi-criteria optimisation. It is important to highlight that our contribution is a purely declarative formulation which is solved by a standard CSP solver but not a particular search engine to solve planning problems encoded as CP formulations. The main advantage is that our formulation can be used indistinctly by any CSP solver without extra effort. Nevertheless, as a consequence of a lack of a planning-oriented search engine, our approach is not competitive enough for solving P&S tasks in comparison to other state-of-the-art planners. In short, in this paper we have shown how far we can go when solving P&S problems with a standard CSP solver.

Our main contribution is to show the suitability of CP formulations to model more realistic P&S problems, and the model presented in this paper means a step in this direction. In general, the CSP solver performance decreases as the number of variables and constraints increase. However, performance tends to be better when the planning effort to solve the causal structure of the problem is not very high. This situation usually happens when the existence of strict constraints limit the number of valid alternatives for a solution plan and, as a consequence, the propagation mechanism helps solve the planning component of the problem. According to our experience, this kind of restrictive scenarios are not very favourable for many planners, some of which tend to systematically fail under highly-constrained problems. As an additional advantage, it is important to note that our CP formulation can be easily adapted and simplified for problems that do not require such a rich expressiveness (Garrido et al. 2008).

According to our results, we can conclude that the most complex problems are P&S tasks that involve fluents. In order to improve the CSP performance, the inclusion of powerful heuristics and the use of branching mechanisms becomes mandatory. In this paper we have presented two simple heuristics that help reduce the search space in most of the problems. As usual, the more informed the heuristic, the more successful the resolution process. For this reason, our future work is oriented towards the definition of more informed heuristics, both in the constraint formulation model and in the CSP solver. Heuristics encoded in the formulation will provide more precise distance measures to bind the variables' domain and reduce the branching factor. Integrating planning heuristics in the CSP solver is a much harder task as it implies a modification of the internal search engine.

Overall, this paper is a first step towards the modelling of more realistic problems, a hot research topic in both planning and scheduling communities. We have presented a flexible CP formulation that can be easily extended to represent any type of constraints. Our next challenge is to improve the performance of the resolution process through the integration of POCL planning heuristics into the CSP solver.

Acknowledgements This work has been partially supported by the Spanish government projects MEC TIN2005-08945-C06-06 (FEDER) and Consolider-Ingenio 2010 CSD 2007-00022. We thank the anonymous reviewers and Maria Fox for their comments to improve the quality of this paper.

References

Blum, A. L., & Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, *90*, 281–300.

Bonet, B., & Geffner, H. (1999). Planning as heuristic search: New results. In S. Biundo & M. Fox (Eds.), *Proc. European conference on planning (ECP-99)* (pp. 360–372). Springer.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*, 5–33.

Chen, Y. X., Wah, B. W., & Hsu, C. W. (2006). Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research*, *26*, 323–369.

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. *Artificial Intelligence*, *49*, 61–95.

Do, M. B., & Kambhampati, S. (2001a). Sapa: a domain-independent heuristic metric temporal planner. In A. Cesta & D. Borrajo (Eds.), *Proc. European conference on planning (ECP-2001)* (pp. 109–120).

Do, M. B., & Kambhampati, S. (2001b). Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, *132*, 151–182.

Edelkamp, S., & Hoffmann, J. (2004). PDDL2.2: the language for the classical part of IPC–4. In *Proc. int. conference on automated planning and scheduling (ICAPS-2004)—International planning competition* (pp. 2–6).

Edelkamp, S., Jabbar, S., & Nazih, M. (2006). Large-scale optimal PDDL3 planning with MIPS-XXL. In *Proc. int. conference on automated planning and scheduling (ICAPS-2006)—International planning competition* (pp. 28–30).

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*, 189–208.

Fox, M., & Long, D. (2001). *PDDL+: an extension to PDDL2.1 for modelling planning domains with continuous time-dependent effects* (Technical report). University of Durham, UK.

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

Fox, M., & Long, D. (2006). Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, *27*, 235–297.

Garrido, A., & Long, D. (2004). Planning with numeric variables in multi-objective planning. In L. Saitta (Ed.), *Proc. European conference on AI (ECAI-2004)*, Amsterdam (pp. 662–666). IOS Press.

Garrido, A., Onaindía, E., & Arangu, M. (2006). Using constraint programming to model complex plans in an integrated approach for planning and scheduling. In *Proc. 25th UK planning and scheduling SIG workshop* (pp. 137–144).

Garrido, A., Onaindia, E., & Sapena, O. (2008). Planning and scheduling in an e-learning environment. A constraint-programming-based approach. *Engineering Applications of Artificial Intelligence*, *21*(5), 733–743.

Gerevini, A., & Long, D. (2006). Plan constraints and preferences in PDDL3. In *Proc. int. conference on automated planning and scheduling (ICAPS-2006)—International planning competition* (pp. 7–13).

Ghallab, M., & Laruelle, H. (1994). Representation and control in IxTeT, a temporal planner. In *Proc. 2nd int. conference on AI planning systems* (pp. 61–67). Hammond.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated planning. Theory and practice*. San Mateo: Morgan Kaufmann.

Haslum, P. (2006). Improving heuristics through relaxed search—an analysis of TP4 and HSP*a in the 2004 planning competition. *Journal of Artificial Intelligence Research*, *25*, 233–267.

Haslum, P., & Geffner, H. (2001). Heuristic planning with time and resources. In *Proc. IJCAI-2001 workshop on planning with resources*.

Hoffmann, J., & Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

Hoffmann, J., Edelkamp, S., Englert, R., Liporace, F., Thiebaux, S., & Trug, S. (2004). Towards realistic benchmarks for planning: the domains used in the classical part of IPC–4. In *Proc. int. conference on automated planning and scheduling (ICAPS-2004)—International planning competition* (pp. 7–14).

Jónsson, A., Morris, P., Muscettola, N., Rajan, K., & Smith, B. (2000). Planning in interplanetary space: theory and practice. In *Proc. 5th int. conference on AI planning systems (AIPS-2000)* (pp. 177–186). Menlo Park: AAAI Press.

Long, D., & Fox, M. (2001). Encoding temporal planning domains and validating temporal plans. In *Proc. 20th UK planning and scheduling SIG workshop* (pp. 167–180).

Long, D., & Fox, M. (2003). Time in planning. In *Handbook of temporal reasoning in AI. Foundations of artificial intelligence* (Vol. 1, pp. 497–537). Amsterdam: Elsevier Science.

Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. In *Proc. natl. conference on artificial intelligence* (pp. 25–32).

Muscettola, N. (1994). HSTS: Integrating planning and scheduling. In M. Zweben & M. S. Fox (Eds.), *Intelligent scheduling* (pp. 169–212). San Mateo: Morgan Kaufmann.

Penberthy, J., & Weld, D. (1994). Temporal planning with continuous change. In *Proc. 12th natl. conference on AI* (pp. 1010–1015).

Penberthy, J., & Weld, D. S. (1992). UCPOP: a sound, complete, partial-order planner for ADL. In *Proc. int. conference on principles of knowledge representation and reasoning* (pp. 103–114). Los Altos: Kaufmann.

Smith, D. E., & Weld, D. S. (1999). Temporal planning with mutual exclusion reasoning. In *Proc. 16th int. joint conference on AI (IJCAI-99)*, Stockholm, Sweden (pp. 326–337).

Smith, D. E., Frank, J., & Jónsson, A. K. (2000). Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, *15*(1), 47–83.

Van Beek, P., & Chen, X. (1999). CPlan: a constraint programming approach to planning. In *Proc. natl. conf. on artificial intelligence (AAAI-99)* (pp. 585–590).

Vidal, V., & Geffner, H. (2004). Branching and pruning: an optimal temporal POCL planner based on constraint programming. In *Proc. natl. conf. on artificial intelligence (AAAI-04)* (pp. 570–577).

Vidal, V., & Geffner, H. (2006). Branching and pruning: an optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, *170*, 298–335.

Weld, D. S. (1999). Recent advances in AI planning. *AI Magazine*, *20*(2), 93–123.

Younes, H. L. S., & Simmons, R. G. (2003). VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, *20*, 405–430.