Using constraint programming to model complex plans in an integrated approach for planning and scheduling

Antonio Garrido, Eva Onaindía, Marlene Arangu
Dpto. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia (Spain)
{agarridot,onaindia,marangu}@dsic.upv.es

Abstract

The objective of this paper is to present an integrated architecture for planning and scheduling (P&S) and, more particularly, the role of a constraint programming module (scheduler) within this framework. The underlying idea is to have two separate modules, a planner and a scheduler, solving jointly a P&S problem. In this collaborative framework, we have firstly focused our research on the modelling of a plan with complex P&S constraints as a CSP. Then a CSP solver takes as input the plan along with all its constraints and validates such a plan. The model presented in this paper can also be easily extended to solve P&S problems rather than just validating plans.

Introduction and motivation

The resolution of P&S problems has followed two different perspectives. In one of the research directions, the temporal planning approach, the objective is to extend planning to cope with scheduling capabilities, that is augmenting the planning reasoning capabilities in order to handle time and resources (Chen, Hsu, & Wah 2005; Gerevini *et al.* 2004; Ghallab, Nau, & Traverso 2004). In the other direction, planning embedded into scheduling, the solution consists of including planning capabilities into a scheduler (Smith & Zimmerman 2004). In this latter approach, the starting point is usually a pre-planned set of ordered activities and planning is called each time it is necessary to release, set up or make any problem component available. However, it is possible to come up with a more general and flexible model where both P&S have an important role in the problem solving, which is a hot topic of research. We use such a model to combine a planning solver and a CSP solver (acting as a scheduling module) as presented in Figure 1, which depicts the structure of our integrated architecture:

- The input data. The input is the problem model (domain+problem definition in any specification language, e.g. PDDL3 (Gerevini & Long 2006)). Additionally, we allow an extension of the model by incorporating further constraints like quantitative temporal restrictions or more sophisticated local conditions for actions. The overall problem definition is divided into two parts: the **propositional part** that includes those aspects related to the causal structure of the problem and the **additional constraints** which comprise the numeric constraints, the preferences and hard constraints and the extra constraints of the extended model. By separating the problem modelling into these two parts we can use a classical planner, as simple (in terms of expressivity and calculus) and efficient as possible, to solve the propositional component of the problem. Moreover, we might even use an input plan generated by the user as a sequence of activities. It is important to note that this plan may abstract out the scheduling (time+resources) requirements, i.e. the plan does not need to be executable because the objective of the integrated module is precisely to repair a given plan and make it fully executable *w.r.t.* all the scheduling requirements. To sum up, we can use a pure STRIPS planner or a PDDL planner or a hand-tailored plan. The output will be, in any case, a skeleton plan or causal structure for the problem at hand. Obviously, the more advanced method we use for generating the plan, the better plan quality.

- The modelling. The second step in our framework is the problem formulation. Instead of encoding planning structures (a planning graph, for instance) (Kambhampati 2000), we encode the plan causal structure and the additional constraints as a CSP. Thus, this paper shows, in detail, the formulation of plans with complex constraints based on the works of (Refanidis 2005; Vidal & Geffner 2006). We can undertake the modelling of a complete plan as a

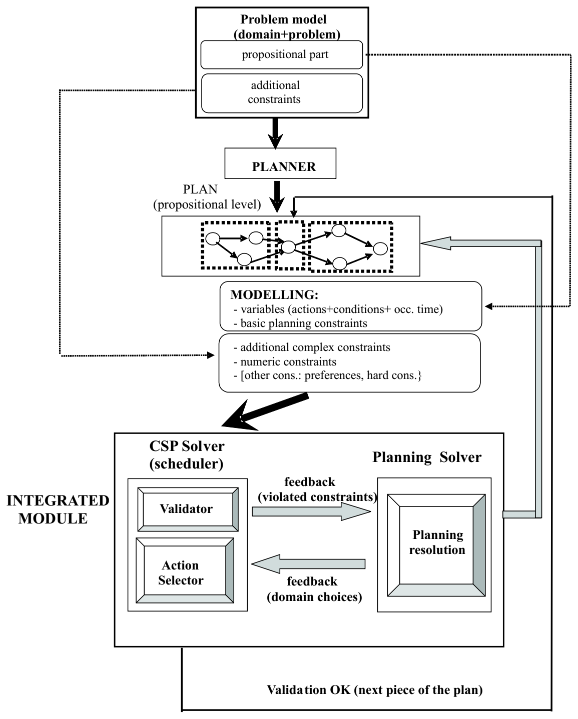

Figure 1: Structure of the integrated architecture.

whole or either a piece of it. The underlying idea is to check the plan piece by piece (where a piece can be a single action or a subplan) so as to perform an incremental validation of the given plan.

- The CSP solver. Once all plan constraints are encoded, a CSP solver implemented under Choco[1] is called. The main job of the CSP solver is to act as a validator by checking the plan constraints and allocating the occurrence time of propositions and actions. If the plan is valid, the CSP solver is invoked again with the next piece of the plan and the same procedure is applied again. At this point, it is important to remark that the CSP solver cannot only act as a validator but also as an action selector. If the encoding of the causal structure considers different choices of supporting the actions conditions, then the CSP solving process will be contributing to *plan* or take decisions about which actions must be used as supporters. This issue is also commented in more detail in our formulation section. All in all, our idea in mind is not to specifically use a CSP solver as a planning solver but to help the planning solver take some decisions during the process.
- The Planning solver. If the validation fails, the CSP solver calls the planning process in order to jointly solve the flaw. The CSP solver will provide the planning solver information about the violated constraints found during the resolution. This feedback will help the planning solver determine the source of the problem and take the most appropriate action to overcome the flaw:
  1. The plan may need to be repaired or even some replanning might also be necessary; in such a case, the planning solver will update the input plan by adding/deleting actions.
  2. Another possibility is to use the CSP solver as an action selector to discover the best supporters for actions. Through the use of heuristic estimations the planning solver can determine the set of the most appropriate candidates for supporting actions (domain choices) and let the CSP solver take the final decision. That is, the planning solver performs an intelligent filtering and pass the resulting information to the CSP solver.

As it can be observed, the three crucial points in this integrated framework are: i) the feedback provided by the CSP solver to the planning solver, ii) the data about the domain choices that the planning solver provides the CSP solver, and iii) the planning solver decision of either letting the CSP solver undertake the flaw resolution or opting for modifying the input plan. These three relevant processes are marked with grey arrows in Figure 1.

More specifically, the rest of this paper focuses on the role of the CSP solver as a validator and as an action selector. Our main contribution is the proposal of a plan formulation that includes very expressive complex (additional) constraints and shows how easily this formulation can be extended to find a plan (action selection).

Extra constraints in the extended model

Our combined model for P&S includes new features and complex constraints between actions (persistences, precedences, temporal windows, etc.) along with numeric capabilities to manage continuous resources. More particularly, the extra features that can be modelled by using our constraint programming formulation are:

- A more elaborate model of actions. First of all, the duration of the actions does not need to be a fixed value, but it can change within a known interval. Moreover, the possibilities when requiring conditions and generating effects go beyond

[1]Choco is a java library for constraint satisfaction problems that can be downloaded from `http://choco.sourceforge.net`

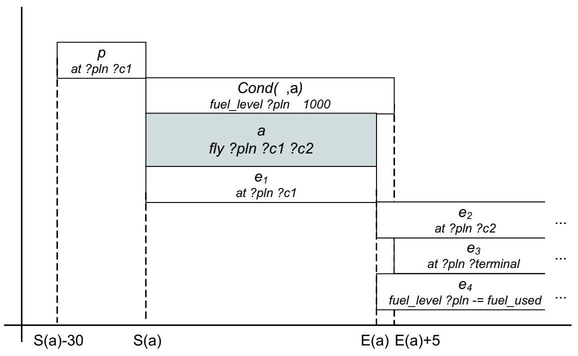

Figure 2: Example of a simple fly action with a more elaborate model of conditions and effects.

the conservative model of actions used in (Vidal & Geffner 2006), and the non-conservative one used in PDDL2.1 (Fox & Long 2003) and its successors (Gerevini & Long 2006). On the one hand, our model allows conditions to be required in **any** interval of time that can be placed totally, partially or even out the execution of the action (see Figure 2). This is interesting since it allows to model conditions that do not fall within the execution of the actions, i.e. real **pre**conditions to be satisfied some time before the actions start. For instance, in Figure 2 the propositional condition $at\ ?pln\ ?c1$ is required since 30 time units before the action fly starts, and the numeric condition $fuel\_level\ ?pln \geq 1000$ is required beyond the end of the action, i.e. **after** the execution of the action. On the other hand, effects can also be generated at any time (within or outside the execution of the action) and persist throughout some time that can be easily specified. In Figure 2 there exist three effects: e_1 is generated when fly starts and persists until it ends, whereas e_2 and e_3 have an infinite persistence. Additionally, e_3 is generated 5 time units after the action finishes because it is necessary the plane traverses the airport segments to arrive at the terminal.

- Precedence constraints between actions. In addition to the implicit qualitative constraints of causal links, it is now possible to model **explicit** ordering constraints between actions, thus including any combination between their start and end times, such as start-start, start-end and so on.
- Temporal constraints between propositions, actions and propositions-actions. Similarly to precedence constraints, our model considers **quantitative** temporal constraints between the generation times of propositions, the start/end times of actions and any combination between them.
- Deadlines in actions and goals. Actions and goals can now be required before a particular deadline, which may constrain the structure of the plan. This allows to model top-level goals to be satisfied not only in the end of the plan, but also at any time during the execution of the plan. This is particularly interesting when it is necessary to accomplish some **intermediate goals** but they do not have to persist until the end of the plan.
- Temporal windows (like timed initial literals of PDDL2.2 (Edelkamp & Hoffmann 2004)) either for propositions or actions. External constraints can include limitations on the availability of some propositions (e.g. sunlight availability is limited), or time intervals when actions must be executed (e.g. actions must only occur during opening times, or fitted within a particular schedule).
- Capabilities for numeric expressions to manage continuous resources, such as energy, profit, etc. Dealing with numeric features helps encode complex constraints (inequalities) on the expressions (e.g. $energy \geq 150$) and numeric assignations (e.g. $profit+ = 50$). Additionally, this also allows to define **multiobjective optimisation** that combines logical (propositional) and numeric expressions.

Formulating plans and complex constraints via constraint programming

Once the structure of causal links of the input plan is obtained, the plan and all the complex constraints that need to be satisfied are formulated via constraint programming (see Figure 1). This formulation builds on the basis of the POCL formulation presented in (Vidal & Geffner 2006), and is extended with the features presented in the previous section. We introduce the constraint programming formulation in two parts. First, we focus on the propositional part, which consists of the definition of variables, basic planning constraints, and additional complex constraints. Second, we present the extension for including numeric capabilities, which defines the variables and planning constraints.

Formulating the propositional part of the plan

Variables and domains The variables are basically used to define actions in the plan and conditions[2] re-

[2]Note that we use the term *condition* instead of *precondition* since conditions can now be required in any position

quired by actions, along with the actions that support the conditions and the time when these conditions occur (time is modelled in $\mathbb{R}$). We also include two dummy actions $Start$ and End; $Start$ supports the propositions of the initial state, whereas End requires the problem goals. These variables, the domains and their description are shown in Table 1.

The meaning for most variables in Table 1 and $Req_{start}(p,a)$, $Req_{end}(p,a)$ require a deeper explanation. The variable $Persist(p,a)$ allows to model persistences (to simulate different states in propositions) in a very flexible way, based on the action b_i that supports p, and the action a for which p is supported. For instance, if $Sup(p,a) = b_i$ the value for $Persist(p,a)$ 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 for $Persist(p,a')$ can be different to $Persist(p,a)$. Therefore, the persistences may depend on two factors: i) the action b_i that generates p, and ii) the action a that requires p. Note that a value $Persist(p,a) = \infty$ represents the infinite persistence used in traditional planning (once an effect is generated it only disappears when it is explicitly deleted by an action). On the other hand, the variables $Req_{start}(p,a)$, $Req_{end}(p,a)$ (obviously, $Req_{start}(p,a) \leq Req_{end}(p,a)$) provide a high expressivity for dealing with conditions, allowing to represent both punctual conditions (when $Req_{start}(p,a) = Req_{end}(p,a)$) and conditions that are required throughout a longer interval.

Basic planning constraints The basic planning constraints consist in creating the relations between the variables, assigning its initial values and imposing disjunctions to solve the threats and the mutex relationships. The constraints are defined for each variable that involve action a or condition p for a. These constraints and their description are shown in Table 2.

The two first constraints in Table 2 are simple. Init-$Req_{start}(p,a)$, $Req_{end}(p,a)$ initialises the time interval when each condition p is required in a, as provided in the action description. For instance according to Figure 2, $Req_{start}(p,a) = S(a) - 30$ and $Req_{end}(p,a) = S(a)$. As can be seen, the three types of conditions introduced in PDDL2.1 can be simply managed as: i) *at start*: $Req_{start}(p,a) = Req_{end}(p,a) = S(a)$, ii) *over all*: $Req_{start}(p,a) = S(a) \wedge Req_{end}(p,a) = E(a)$, and iii) *at end*: $Req_{start}(p,a) = Req_{end}(p,a) = E(a)$. Other types of condition requirements can be defined similarly. Init-$Time(p,a)$, $Persist(p,a)$ is a conditional constraint that initialises the two variables depending on the action that supports p for a. According to Figure 2, if a is the action that supports e_3 for a new action b, the initialisation is: $Time(e_3,b) = E(a) + 5$ and $Persist(e_3,b) = \infty$. The two last constraints are included to solve threats and mutexes. In the first case, a disjunction to promote or demote the action that provokes the threat is posted. In the second case, a distinct constraint is posted to avoid the simultaneous modification (effects interference) of the same proposition p. In Figure 2, assuming that a new action b requires proposition e_1, the two resulting constraints are: $(S(a) < Time(e_1,b)) \vee (Req_{end}(e_1,b) < S(a))$ and $(S(a) \neq Time(e_1,b))$, respectively. Note that the mutex constraint does not prevent two actions that are mutex from overlapping in any way (as it happens in a conservative model of actions). On the contrary, in our model the actions cannot change the same proposition simultaneously, but they can still partially overlap.

(before, immediately before, some time during, immediately after, etc.) *w.r.t.* the execution of the action.

Additional complex constraints Using constraint programming makes easier the formulation of complex constraints (precedences, persistences, etc.) that are not usually included in traditional planning. Following this line, the constraints that can be modelled, together with their description are shown in Table 3.

Formulating the numeric part of the plan

The previous purely propositional representation is not adequate to model numeric variables that express the use of continuous resources in actions. Fortunately, we can extend the constraint programming model to formulate the numeric part of the plan, thus including the numeric variables used in the conditions and effects of the actions. We denote these numeric variables as fluents to avoid confusion with the variables of the constraint programming model. Like in the formulation of the propositional part of the plan, we first introduce the variables of the model and later the basic constraints that involve such constraints.

Variables and domains The meaning of the variables are very similar to those defined in Table 1, but now related to fluents instead of propositions (see Table 4). We split the variables into three groups. First, the variables that are necessary for supporting each fluent ($Sup(\phi,a)$ and $Time(\phi,a)$, for each action a that has ϕ as a condition or effect). Second, the variables that are necessary if action a requires ϕ as a numeric condition ($Req_{start}(\phi,a)$ and $Req_{end}(\phi,a)$). Finally, the variables that are necessary to propagate the value of ϕ throughout the variables of the model ($V_{actual}(\phi,a)$ and $V_{updated}(\phi,a)$). All these variables and their description are shown in Table 4.

Variable	**Domain**	**Description**
$S(a), E(a)$	$[0, \infty]$	Start and end time, respectively, of action a. Clearly, $S(Start) = E(Start) = 0$ and $S(End) = E(End)$
$dur(a)$	$[dur_{\min}(a), dur_{\max}(a)]$	Duration of the action within two positive bounds. Clearly, $dur(Start) = dur(End) = 0$
$Sup(p, a)$	$\{b_i\} \mid b_i$ adds p for a	Symbolic variable with the supporter b_i that represents the causal link $b_i \xrightarrow{p} a$. Although given a plan only one action supports p for a, we define the domain of $Sup(p, a)$ as as set to keep it more general for additional extensions (see section *Using the CSP solver as an action selector*)
$Time(p, a)$	$[0, \infty]$	Time when the causal link $Sup(p, a)$ happens, i.e. the time in which the action b_i selected as a value for variable $Sup(p, a)$ generates p
$Persist(p, a)$	$[0, \infty]$	Persistence of condition p for a
$Req_{start}(p, a)$, $Req_{end}(p, a)$	$[0, \infty]$	Interval $[Req_{start}(p, a), Req_{end}(p, a)]$ in which action a requires p

Table 1: Definition of propositional variables and domains.

Constraint	**Description**
$S(a) + dur(a) = E(a)$	$Start\text{-}End$: *tie* the start and end of action a
$E(a) \leq S(End)$	$E\text{-}End$: places action End as the last action in the plan
$Req_{start}(p, a) = S(a)/E(a) + x, x \in \mathbb{R}$ $Req_{end}(p, a) = S(a)/E(a) + x, x \in \mathbb{R}$	Init-$Req_{start}(p, a)$, $Req_{end}(p, a)$: assigns the initial value according to the definition of action a and its condition requirements
if $Sup(p, a) = b_i$ then $Time(p, a) =$ time when b_i adds p $Persist(p, a) =$ persistence given by b_i	Init-$Time(p, a)$, $Persist(p, a)$: conditional constraint that assigns the initial value according to the definition of a
$Time(p, a) \leq Req_{start}(p, a)$	Causal link: forces to support p for action a before a requires it
$\forall b_i \in Sup(\neg p)$ $time(b_i, \neg p) \leftarrow$ time when b_i deletes p $(time(b_i, \neg p) < Time(p, a)) \vee$ $(Req_{end}(p, a) < time(b_i, \neg p))$	Threat resolution: assuming $Sup(\neg p)$ contains the actions that delete p, it solves the threat that such actions provoke to the causal link $Sup(p, a)$ by using promotion or demotion
$\forall b_i \in Sup(\neg p)$ $time(b_i, \neg p) \leftarrow$ time when b_i deletes p $time(b_i, \neg p) \neq Time(p, a)$	Mutex resolution: assuming $Sup(\neg p)$ contains the actions that delete p, it solves the mutex situation with action that supports p of a

Table 2: Definition of propositional basic planning constraints.

Constraint	**Description**
Var_1 `comp-op` $Var_2 + x, x \in \mathbb{R}$	Precedence and quantitative temporal constraints, which can involve propositions, actions and propositions-actions. This constraint is very flexible and can represent any combination of $Var_1, Var_2 \in \{Time(p, a), Req_{start}(p, a), Req_{end}(p, a), S(a), E(a)\}$ and `comp-op` $\in \{<, \leq, =, \geq, >, \neq\}$, such as $S(Start) + 100 < E(End)$ that restricts the plan makespan
$Req_{end}(p, a) \leq Persist(p, a)$	Persistence constraints: states that the upper bound of the interval of a condition requirement never exceeds the value for the persistence of such a condition
$Var_1 \leq x, x \in \mathbb{R}$	Deadlines, which can involve the start/end times of actions or propositions, where $Var_1 \in \{S(a), E(a), Time(p, a)\}$
$\min(tw(p)) \leq Req_{start}(p, a)$ $\leq Req_{end}(p, a) \leq \max(tw(p))$ $\min(tw(a)) \leq S(a)$ $\leq E(a) \leq \max(tw(a))$	Temporal windows, which encode external constraints that propositions and actions must hold, where $tw(p)$ and $tw(a)$ are the temporal windows for p and a, respectively
$Expression(Var_1, Var_2 \ldots Var_n)$	Other customised derived constraints, which encode more complex constraints among several variables of the problem

Table 3: Definition of propositional additional complex constraints.

Variable	Domain	Description
$Sup(\phi, a)$	$\{b_i\} \mid b_i$ supports ϕ for a	Symbolic variable with the supporter b_i of ϕ. Similarly to $Sup(p, a)$, we define the domain of $Sup(\phi, a)$ as as set to keep it more general
$Time(\phi, a)$	$[0, \infty]$	Time in which the action b_i selected as a value for variable $Sup(\phi, a)$ updated ϕ
$Req_{start}(\phi, a)$, $Req_{end}(\phi, a)$	$[0, \infty]$	Interval $[Req_{start}(\phi, a), Req_{end}(\phi, a)]$ in which action a must satisfy the numeric condition $Cond(\phi, a)$ (see description in Table 5)
$V_{actual}(\phi, a)$	$[-\infty, \infty]$	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 $S(a)$
$V_{updated}(\phi, a)$	$[-\infty, \infty]$	Value that a updates for fluent ϕ. This variable is only necessary if a modifies the value of ϕ

Table 4: Definition of numeric variables and domains.

Note that the main difficulty in this formulation is how to find out which action supports each fluent ϕ and how to propagate its value throughout the variables of the model. First of all, when dealing with fluents the term *support* is not very appropriate since there is no a particular action that produces 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 while propagating this value, the necessity of $V_{actual}(\phi, a)$ is twofold. First, if there exists a numeric condition on ϕ in a, $V_{actual}(\phi, a)$ stores the value of ϕ (that must obviously satisfy the numeric condition). Second, if there exists a numeric effect on ϕ in a (i.e. $V_{updated}(\phi, a)$ is present), $V_{actual}(\phi, a)$ stores the initial value to help calculate the updated value because the effects of a can increase or decrease such an initial value (for instance, in Figure 2 the effect e_4 is calculated as $V_{updated}(\phi, a) = V_{actual}(\phi, a) - fuel\_used$).

Basic planning constraints Basically, these constraints represent the same relations that the planning constraints of the propositional part of the plan and they are shown in Table 5.

The main differences *w.r.t.* the propositional basic constraints are twofold. First, $Cond(\phi, a)$ that represents the numeric condition that $V_{actual}(\phi, a)$ must satisfy (see Figure 2). Second, the assignment of $V_{actual}(\phi, a)$ which represents the true propagation of the value of ϕ from the value updated by b_i to the actual value of a. Moreover, note that the mutex resolution only prevents from updating the same fluent simultaneously. In some cases, a stricter constraint might be generated, preventing two actions that update the same fluent from overlapping in any way. Finally, like in the propositional part, additional complex constraints could be generated among the variables of the model.

Using the CSP solver as an action selector

The CSP solver can be used not only to validate the plan formulated via constraint programming, but also as an action selector that helps the planner discover the best supporters for actions (see Figure 1). In order to perform this selection, two extensions are needed:

- The domain of variables $Sup(p, a), Sup(\phi, a)$ will now contain all the action choices the planner provides the CSP. This way, the supporting action is not uniquely decided by the planner, but the scheduler also takes part in choosing the best action according to the model, thus considering all complex constraints. The domain of these variables will be now larger and therefore this implies a higher branching in the CSP solving process. Therefore, this makes more necessary the use of heuristic estimations to bound and prune inadequate alternatives.
- New activation variables are required to activate the variables related to the action selected by the CSP. For instance, if the planner provides $Sup(p, a) = \{b_i, b_j\}$ and the CSP selects b_i, all the variables $S(b_i)$, $E(b_i)$, $dur(b_i)$, $Sup(p_i, b_i)$, $Sup(\phi_i, b_i)$ and so on, and the constraints that involve these variables need to be activated in the model, while variables related to b_j will not be considered. This general formulation can be done following the process of selecting supporting actions presented in (Vidal & Geffner 2006).

A simple example

Let us consider a simple problem of the traditional `Zenotravel` domain of IPC (Fox & Long 2003; Edelkamp & Hoffmann 2004) to board (bd), transport (fly) and debark (db) a person $pers$ in a plane pln from city $c1$ to $c2$. We assume that bd requires the plane in $c1$ until 10 units after bd ends ($E(bd) + 10$), fly requires more than 1000 l. of fuel and consumes 800 l., and generates the effect of being in the terminal of $c2$ (ready for $pers$ to debark) 5 units after end-

Constraint	Description
$Cond(\phi, a)$=condition	Numeric condition: assignment of the condition constraint that ϕ must accomplish for a in $[Req_{start}(\phi, a), Req_{end}(\phi, a)]$. It simply consists in initialising the constraint as $Cond(\phi, a) = V_{actual}(\phi, a)$ `comp-op` $x, x \in \mathbb{R}$, where `comp-op` $\in \{<, \leq, =, \geq, >, \neq\}$ according to the action definition
$Req_{start}(\phi, a) = S(a)/E(a) + x, x \in \mathbb{R}$ $Req_{end}(\phi, a) = S(a)/E(a) + x, x \in \mathbb{R}$	Init-$Req_{start}(\phi, a), Req_{end}(\phi, a)$: assigns the initial value according to the definition of action a and its numeric condition requirements
$V_{updated}(\phi, a)$=modification/assignment	Init-$V_{updated}(\phi, a)$: consists in creating the initialisation, based on a modification of $V_{actual}(\phi, a)$ or on an assignment of an absolute value
if $Sup(\phi, a) = b_i$ then $Time(\phi, a) =$ time when b_i updates ϕ $V_{actual}(\phi, a) = V_{updated}(\phi, b_i)$	Init-$Time(\phi, a), V_{actual}(\phi, a)$: conditional constraint that assigns the initial value according to the definition of a
$Time(\phi, a) \leq Req_{start}(\phi, a)$	Causal link
$\forall b_i$ that updates $\phi \mid b_i \notin Sup(\phi, a)$ $time(b_i, \phi) \leftarrow$ time when b_i updates ϕ $(time(b_i, \phi) < Time(\phi, a)) \vee$ $(Req_{end}(\phi, a) < time(b_i, \phi))$	*Threat* resolution: represents the disjunction to avoid the update of ϕ between $Time(\phi, a)$ and $Req_{end}(\phi, a)$
$\forall b_i$ that updates $\phi \mid b_i \notin Sup(\phi, a)$ $time(b_i, \phi) \leftarrow$ time when b_i updates ϕ $time(b_i, \phi) \neq Time(\phi, a)$	Mutex resolution: prevents two actions from updating the same fluent ϕ simultaneously

Table 5: Definition of numeric basic planning constraints.

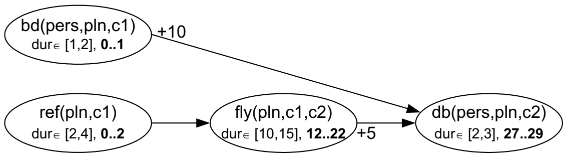

Figure 3: Plan for the application example. The possible durations are represented between brackets. The execution time validated and allocated by the CSP is represented in bold font.

ing ($E(fly) + 5$). The input plan, with the durations of each action and the basic causal links, is shown in Figure 3. This plan is then formulated via constraint programming as shown in Table 6. For lack of space, only the variables and constraints where action fly is involved are represented in that table, but the others are defined analogously.

The CSP solver validates the model of the plan by satisfying all the constraints. If the plan is valid the CSP allocates the execution times, as shown in Figure 3. Otherwise, the CSP informs about the constraint that could not be satisfied. For instance, if the problem posts a deadline on the plan makespan, such as $E(End) \leq 25$, the CSP informs that the plan is invalid because that constraint generates a contradiction and the plan needs to be repaired (see Figure 1).

Conclusions

In this paper we have presented a formulation to encode a plan with complex constraints as a CSP. The formulation presented here is highly influenced by the work in (Vidal & Geffner 2006) though there exist several differences: i) our model contemplates the formulation under a very expressive, non-conservative temporal model, ii) the modelling of additional constraints, and iii) the formulation of numeric fluents. Unlike (Vidal & Geffner 2006), our main purpose was the formulation of a plan rather than a problem in order to incorporate such a modelling as a part of an an integrated P&S module. In this sense, our interest is to develop a planning solver that cooperates with the CSP solver in the process of finding a plan. It is important to highlight that the overall approach could be also used to solve a problem starting with an empty initial plan because the formulation can be extended to model a problem (like in (Vidal & Geffner 2006)) instead of simply a plan; in this case, the CSP solver would always work as an action selector which we actually intend to use to ease the planning solver task. Our current work focuses on the feedback that the two modules provide each other and on the repair/replan activities of the planning solver.

Acknowledgments

This work has been partially supported by the Spanish government project MCyT TIN2005-08945-C06-06 (FEDER) and by the Valencian government project GV06/096.

Variables	**Constraints** (propositional+numeric)
$S(fly), E(fly) \in [0, \infty]$	$S(fly) + dur(fly) = E(fly)$
$dur(fly) \in [10, 15]$	$E(fly) \leq S(End)$
$Sup(pln\_c1, fly) \in \{Start\}$	$Req_{start}(pln\_c1, fly) = S(fly); Req_{end}(pln\_c1, fly) = S(fly)$
$Sup(fuel\_pln, fly) \in \{ref\}$	if $(Sup(pln\_c1, fly) = Start)$
$V_{actual}(fuel\_pln, fly) \in [-\infty, \infty]$	$Time(Sup(pln\_c1, fly) = E(Start)$
$V_{updated}(fuel\_pln, fly) \in [-\infty, \infty]$	$Persist(pln\_c1, fly) = \infty$
	$Time(pln\_c1, fly) \leq Req_{start}(pln\_c1, fly)$
	$(S(fly) < Time(pln\_c1, ref)) \vee (Req_{end}(pln\_c1, ref) < S(fly))$
	$S(fly) \neq Time(pln\_c1, bd); S(fly) \neq Time(pln\_c1, ref)$
	$Cond(fuel\_pln, fly) = (V_{actual}(fuel\_pln, fly) \geq 1000)$
	$Req_{start}(fuel\_pln, fly) = S(fly); Req_{end}(fuel\_pln, fly) = E(fly) + 5$
	$V_{updated}(fuel\_pln, fly) = V_{actual}(fuel\_pln, fly) - 800$
	if $(Sup(fuel\_pln, fly) = ref)$
	$Time(fuel\_pln, fly) = E(ref)$
	$V_{actual}(fuel\_pln, fly) = V_{updated}(ref)$
	$Time(fuel\_pln, fly) \leq Req_{start}(fuel\_pln, fly)$
	$(E(Start) < Time(fuel\_pln, fly)) \vee$
	$(Req_{end}(fuel\_pln, fly) < E(Start))$
	$E(Start) \neq Time(fuel\_pln, fly)$

Table 6: Variables and constraints for the action fly of the application example. We have used integer domains because real variables are still under development in Choco.

References

Chen, Y.; Hsu, C.; and Wah, B. 2005. Subgoal partitioning and resolution in SGPlan. In *Proc. System Demonstration Session, Int. Conference on Automated Planning and Scheduling (ICAPS-2005)*, 32–35.

Edelkamp, S., and 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*, 2–6.

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

Gerevini, A., and Long, D. 2006. Plan constraints and preferences in PDDL3. In *Proc. Int. Conference on Automated Planning and Scheduling (ICAPS-2006) – International Planning Competition*, 7–13.

Gerevini, A.; Saetti, A.; Serina, I.; and Toninelli, P. 2004. Planning in PDDL2.2 domains with LPG-TD. In *Proc. Int. Conference on Automated Planning and Scheduling (ICAPS-2004) – International Planning Competition*, 33–34.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning. Theory and Practice*. Morgan Kaufmann.

Kambhampati, S. 2000. Planning graph as (dynamic) CSP: Exploiting EBL, DDB and other CSP techniques in Graphplan. *Journal of Artificial Intelligence Research* 12:1–34.

Refanidis, I. 2005. Stratified heuristic POCL temporal planning based on planning graphs and constraint programming. In *Proc. ICAPS-2005 Workshop on Constraint Programming for Planning and Scheduling*.

Smith, S., and Zimmerman, T. 2004. Planning tactics within scheduling problems. In *Proc. ICAPS-2004 Workshop on Integrating Planning Into Scheduling*, 83–90.

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