INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

# Planning for Spatial Missions Using Answer Set Programming

**Rodrigo Joel Lucas Santos**

Dissertation submitted to obtain the Master Degree in

## Information Systems and Computer Engineering

**Jury**

President: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof.ª Maria Inês Camarate de Campos Lynce de Faria
Members: Prof.ª Maria Luísa Torres Ribeiro Marques da Silva Coheur

**November 2012**

# Abstract

Exploring space has always been a priority to mankind. One example of this is the International Space Station, where scientist crews develop the space programs of countries all over the world. Finding an optimal work schedule for the scientist crew is a hard task, and automated planners have tried to solve that problem. Each crew member's schedule is constrained by the tasks he is forced to perform, as well as the specific constraints on each available task. The goal is to find a plan that assigns all tasks to a crew member, and this plan must have the shortest possible length.

In this work, we use the declarative language of Answer Set Programming to assign a work schedule to each crew member of the spatial mission. We introduce the ASP language, the programming methodology and the tools we use. We present four dedicated programs to solve the described problem, elaborating on ASP program design and optimization of the grounding and solving processes. The encodings follow two separate models which are based on different interpretations of the problem's constraints. Finally, we compare our work to planners submitted to the International Planning Competition.

## Keywords

Answer Set Programming
Artificial Intelligence Planning
Crew Planning
Scheduling

# Resumo

Explorar o espaço é há muito tempo uma prioridade para a espécie humana. Um exemplo disto é a Estação Espacial Internacional, onde tripulações de cientistas desenvolvem os programas espaciais de países de todo o mundo. Encontrar um horário de trabalho óptimo para a tripulação de cientistas é uma tarefa difícil, e planeadores automáticos tentam resolver esse problema. O horário de cada membro da tripulação é restringido pelas tarefas que este é obrigado a fazer, tal como pelas restrições específicas de cada tarefa disponível. O objectivo é encontrar um plano que atribui todas as tarefas a um membro da tripulação, e este plano deve ter a menor duração possível.

Neste trabalho, utilizamos a linguagem declarativa Answer Set Programming para atribuir um horário de trabalho a cada membro da tripulação da missão espacial. Introduzimos a linguagem ASP , o método de programação e as ferramentas utilizadas. Apresentamos quatro programas dedicados que resolvem o problema descrito, elaborando sobre o desenho de programas em ASP e na optimização dos processos de grounding e solving. As codificações seguem dois modelos distintos que estão baseados em interpretações diferentes das restrições do problema. Finalmente, comparamos o nosso trabalho a planeadores submetidos na Competição Internacional de Planeamento.

## Palavras-Chave

Answer Set Programming
Planeamento com Inteligência Artificial
Planeamento da Tripulação
Escalonamento

## Acknowledgements

First of all, I would like to thank my advisor, Prof.ª Inês Lynce. Her wisdom and direction guided this boat to a safe port, was always supportive and encouraging, both in the good and the bad moments, and was always available to help in any way I needed. I would also like to thank the FCT, for supporting and financing this work under the project ASPEN (PTDC/EIA-CCO/110921/2009), and INESC-ID, for providing material and resources during this work.

Then, I would like to thank my family. They are my rock and I would have never achieved this if I didn't have their support: my mother and father, who have done everything in their power so that I could fulfill my potential; my brother and sister, who are amazing, each in their own way, and are the best brothers one could hope for; and my grandparents, who have always dreamed of a brighter future for me.

Finally, I would like to thank all of my friends, for making my life meaningful and for all the fun moments. A special thanks to Filipe Gouveia and Tiago Almeida, who worked together with me, and their collaboration and friendship made this work a lot easier to handle.

*Rodrigo Santos*

*Escravilheira, 15 de Outubro de 2012*

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Mankind has always been fascinated by space. Ever since the time of early civilizations humans venerated astral objects as gods – there is a Sun god in most ancient civilizations. The science of astronomy accompanied human evolution, going from the geocentric model of Plato and Aristotle to the heliocentric model of Copernicus and Galileo. With the technology of the 20th century, we started to venture into space, in a journey that culminated with the Apollo 11 mission and the first human landing on the Moon.

The International Space Station (ISS) is a permanently inhabited station, that orbits around the Earth. It is a joint effort of five international partners: USA, Russia, Japan, Canada and Europe. These partners work together in order to develop their space programs, while studying the conditions of life in outer space. The low-gravity environment of the ISS allows the conduction of scientific experiments that would otherwise be impossible.

The planning of all ISS activities is greatly detailed. Each year of the mission goes through several planning stages: planning starts with a document describing the ground rules and constraints of the mission, a first plan is created that summarizes the priorities for the year, which gets more detailed in several planning iterations. The final product is a collection of short-term plans that are executed in the ISS. In this work we will address a small part of this planning structure, which concerns the short-term plans of the activities of the crew members.

Answer Set Programming (ASP) is an up-and-coming declarative language that has earned a place in Knowledge Representation and Artificial Intelligence conferences, despite its youth. Its appeal comes from being an expressive, efficient and easy to use language. Because of this, we adopted ASP as the technology used in this work. Our goal is as much to solve the planning problem of the ISS, as it is to explore the potential of ASP for the resolution of planning problems.

In chapter 2 – Background we introduce the background concepts required to understand this work. In section 2.1 we will present the crew planning problem, and explain the instances of this problem in some detail. In section 2.2 we will give some background about Planning, starting with some basic concepts, then presenting two possible representations for planning problems and introducing scheduling problems as a class of planning problems. In section 2.3 we introduce the declarative language of answer set programming, exemplify its programming methodology, and describe some of the tools that may be used with it.

In chapter 3 – Solution Development we present the developed work and explain the development process. In section 3.1 we present additional constraints that were considered when tackling the problem. In section 3.2 we give a list of the ASP predicates used to represent the problem. In section 3.3 we

exemplify the process of building and understanding a problem instance. In sections 3.4 to 3.7 we present several solution proposals for the crew planning problem. Finally, in section 3.8 we discuss alternative resolution approaches that were not successful.

In chapter 4 – Experimental Evaluation we evaluate the developed programs, by comparing them to each other and to other planners. In section 4.1 we provide insight about the instances used in the evaluation process. In section 4.2 we present and analyse the results of each evaluation.

Finally, in chapter 5 – Conclusion we present the conclusions we derived from this work. Furthermore, in section 5.1 we add possible directions to follow up on this work.

This work contributes with four dedicated programs designed to plan the schedule of ISS crew members. Furthermore, we present extensive background and useful methodology for anyone willing to work with ASP, particularly about ASP program design and optimizations methods.

# 2 Background

## 2.1 Crew Planning

The complexity of the ISS makes the task of planning for all its operations a hard one. The approach taken by NASA is to plan separately for each module of the ISS[1], and then coordinate the different modules in order to have a plan that can be executed as a whole. We will focus on describing the **Operations (Ops) Planner**, which controls the daily activities of the habitants of the ISS. From here on, the work of the Ops Planner will be referred to as **crew planning**.

The crew planning problem consists in designing a schedule for the activities of the crew members of the ISS, over a number of days[3][2]. The scheduled plan must be feasible – each crew member can perform at most one activity at any instant. If an activity is part of the plan, then all its constraints must be respected. There are also medical restrictions that cannot be violated; in order to ensure the safety of the crew members, and as such some activities are enforced in the plan. As long as the schedule respects the boundaries presented above, it can then be optimized in order to obtain the best possible plan. These optimizations involve giving the crew members a stable schedule over several days, respecting their personal preferences, and fulfilling science goals. Science goals are the ultimate motivation for the existence of the ISS; as such, it is expected that the crew performs as much science experiments as possible.

The basic plan for one day is composed of the following activities:

- At the start of the day, the crew has a fixed **post-sleep period** of 90 minutes. In this period, the crew members have breakfast and prepare themselves for the day. Time remaining in this period can also be used as personal time. It is up to each crew member to organize the activities in this period.
- Immediately after the post-sleep period, there is a **daily planning conference** (DPC) that takes 15 minutes. The DPC is essentially a briefing used to communicate to the crew their planned activities for the day.
- At the end of the day, the crew has a fixed **pre-sleep period** of 120 minutes. This period includes a debriefing of the activities performed during the day, dinner and personal time.

[1] The different modules of the ISS are briefly described in http://www.shuttlepresskit.com/ISS_OVR/flt_control.htm

[2] A description of the domain is given at http://babelfish.arc.nasa.gov/trac/europa/browser/PLASMA/trunk/src/PLASMA/ANML/test/parser/CrewPlanning-model.anml


Figure 2.1: Incomplete schedule of one day

- The pre-sleep period is followed by a **sleep period** of 8,5 hours (510 minutes). The sleep period ends a day cycle of 24 hours.
- In the middle of the day each crew member takes 1 hour for **lunch**. It is recommended, but not mandatory, that all crew members have lunch together, at the same time. Since this is the only time the crew members take a break to eat during the day, it is required that there are at least 4 hours between meals, in order to ensure their eating schedule is balanced. This means there are at least 4 hours between the end of the post-sleep period and the beginning of lunch time, and likewise, there are at least 4 hours between the end of lunch time and the beginning of the pre-sleep period.
- Each crew member is required to **exercise** for 2,5 hours each day, in order to counter the adverse effects of being in a low-gravity environment. The exercise time may be completed at once, or be divided into two periods: one period of 90 minutes and one period of 60 minutes, not necessarily in this order. A period of exercise is associated with a machine; each machine can be used by only one person at a time. The crew members can state their preferences with respect to the number of exercise periods, the machines they use and the time of the day they exercise. However, the crew members cannot exercise within one hour after they finish a meal.

Figure 2.1 shows how does a plan with only the essential tasks look like. Each block corresponds to 1 hour, and is divided into four periods of 15 minutes. Note that the lunch break and the exercise periods could be scheduled in another time of the day; this is only an example.

The remaining activities depend on the problem instance. Some of these activities are required when we are planning for several days. For example, each crew member must attend a **medical conference**

every two days, the duration of a medical conference being 60 minutes. If the duration of the plan is one day, then this activity is not mandatory unless it is specifically required in the problem instance.

Another example is that every two days, the **air filters** of the ISS must be changed. Like the medical conferences, this activity is not mandatory in 1-day plans unless it is required in the problem instance. Only one crew member needs to perform this task. The duration of a filter change task is 60 minutes.

The **science experiments** to be scheduled must always be present in the problem instance. Each science experiment is treated like a single task. The duration of the task depends on the experiment to be performed. A science experiment can be performed by one or several crew members. Experiments may be physically restraining, in which case there is a limit on the numbers of such experiments that one crew member can perform in a day. Experiments can also use a communications channel, which is a limited resource. Communication channels are treated in the same way exercise machines are – each channel can only be associated with one task at a time. All the information about the experiment itself, and other variables (like the capacity of each crew member to perform physical experiments, and the number of communication channels), is detailed in the problem instance.

Problem instances may also include maintenance or repair operations that must be performed during the schedule time. One example of such an operation is the replacement of the **Remote Power Controller Module** (RPCM). This is a complex task that consists in several smaller tasks, which are executed in the following order:

1. - Reconfigure Thermal Loops
   - Remove Sleep Station
2. Remove Power Source
3. Replace RPCM
4. Place Power Source
5. - Reconfigure Thermal Loops
   - Assemble Sleep Station

Tasks which are marked in a bullet list do not have a specific order of execution. They can be performed in any order, or even simultaneously. The duration of each task is 60 minutes. Different tasks can be executed by different crew members, as long as the temporal order is respected.

Similarly, other operations which depend on executing several tasks can be performed, as long as the order dependencies between tasks and the duration of each task are specified in the instance.

## 2.2 AI Planning

Planning is the area of Artificial Intelligence (AI) dedicated to the study of goal-oriented reasoning, of how can an AI entity form a strategy of action in order to achieve a goal. Not only is it important to

understand intelligence and reasoning in the theoretical plane, but it has also a strong practical side. There are many applications of AI planning, from evacuation planning[36] to the game of bridge[32].

In this section we define the basic concepts of planning, present some of the most used representations for planning problems, and show how planning is applied to problems similar to the crew planning problem. The section is strongly based on the work of Russell and Norvig[30] and Ghallab, Nau and Traverso[19] in the area.

### 2.2.1 Basic concepts

In order to represent a planning problem, it is essential to understand the notions of state and action.

A **state** is a static representation of the environment. It should contain all the relevant information about the environment – every characteristic related to the problem in hand and that may change under specific conditions. There is a special state, the **goal state**, which fulfills the conditions of the problem. One common representation for a state is a set of boolean variables; the state is then defined by the combination of truth values of its variables, which is unique to each state.

**Actions** are the catalysts for transitions between states. In most cases, an action changes the environment in some way, bringing it to a different state. Each action is defined by its **preconditions** and its **effects**. An action is **applicable** to a state if the state meets its preconditions; the effects describe the modifications applied to the current state in order to generate the next state.

Ultimately, the planning problem consists in finding a sequence of actions that, given a starting state, will lead to the goal state. Often, this sequence represents a timeline, that may or may not allow several actions to be executed in parallel. Plans that allow parallel actions are called **partial-order plans**, while plans that define a strict order relationship between actions are **total-order plans**.

### 2.2.2 STRIPS

STRIPS (short for Stanford Research Institute Problem Solver) is a representation created by Richard Fikes and Nils Nilsson[8], used to describe planning problems over restricted state-transition systems. This kind of problems has the following characteristics:

- They are **deterministic** – every action has a predictable outcome.
- They are **static** – the only way of changing the current state is by applying an action.
- They are **discrete** – states, actions and transitions have finite domains, instead of being continuous.
- They are **fully observable** – we always know everything about the current state; there is no hidden information.

**Definition 1** *A restricted state-transition system can be described as a tuple $\Sigma = (S, A, \gamma)$, where $S$ is a set of states $S = \{s_1, s_2, ..., s_n\}$, $A$ is a set of actions $A = \{a_1, a_2, ..., a_n\}$ and $\gamma$ is a transition function,*

*in which for every pair $(s, a)$, $s \in S$, $a \in A$ and $a$ is applicable to $s$, $\gamma(s, a)$ is the state generated by applying $a$ to $s$.*

STRIPS uses first-order logic to describe states and actions. A state is a conjunction of positive literals. STRIPS employs the **closed-world assumption**, so any literals not present in the state are assumed to be negative. The possible literals for states are either grounded first-order predicates or constants; there are no function symbols in STRIPS. This way we are guaranteed to have a finite set of terms, and consequently a finite number of states. The start and goal states of a problem are also described by a conjunction of positive literals. While the start state is treated in the same way as any other state, the goal is considered to be a partial specification of a state, which forces the goal state to contain all the literals specified, but allows other literals to be present. To check if a state fulfills the goal, we introduce the notion of **satisfiability**: a state $s$ satisfies a goal $g$ ($s \models g$) if every literal of $g$ is in $s$.

Actions in STRIPS are expressed through the use of **operators**. An operator is composed of a name, its preconditions and its effects. The name of the operator is an expression of the form $n(x_1, ..., x_k)$, $n$ being a name which is unique for each operator and $x_1, ..., x_k$ are the variables used in the preconditions and effects. The preconditions of the operator are a set of positive literals, which must all be present in a state in order to apply an action. The effects are a set of literals which describe the modifications to the state to which the action is applied: Positive literals in this set are added to the new set, while negative literals are removed from it. Because of this, the set of effects can also be divided in two separate sets: *add* and *delete*. An action is then a grounded instance of an operator: all variables present in first-order predicates are replaced by constants, in order to determine exactly the preconditions and effects of an action (we can say that an operator is a generalization of an action).

The following is an example of a STRIPS problem and the generated plan:

```
Initial state: At(Bedroom)


Goal State: Tidy(Kitchen)


Operators:
      Move(X,Y)
      Preconditions: At(X)
      Effects: not At(X), At(Y)


      Clean(X)
      Preconditions: At(X)
      Effects: Tidy(X)
```

$s_0 = At(Bedroom)$

$a_1 = Move(Bedroom, Kitchen)$

$s_1 = \gamma(s_0, a_1) = At(Kitchen)$

$a_2 = Clean(Kitchen)$

$s_2 = \gamma(s_1, a_2) = At(Kitchen), Tidy(Kitchen)$

Notice that in state $s_1$, the literal *not* $At(Bedroom)$ is not represented, because as stated before, only the positive literals are present – everything else is assumed to be false.

While STRIPS is useful to represent basic planning problems, it is not powerful enough to represent the crew planning problem. The major flaw here is that STRIPS only allows preconditions to have positive literals; but in this problem, we will often need to use negative literals and boolean expressions as preconditions to some actions. Thus, a more complex representation is required.

### 2.2.3 ADL

In order to improve on the limitations of STRIPS, AI researchers worked on extensions that provide added functionality (and with it, complexity). ADL, created by Pednault[28], is one of such extensions.

The basic assumptions for STRIPS about the environment still hold for ADL; we are still dealing with a restricted state-transition system. However, instead of assuming a closed-world, ADL establishes that the world is **open** – literals that do not appear in a state are unknown. Here lies the major difference between STRIPS and ADL, because now we need strong negation to represent negative literals, and as such, positive and negative literals are dealt with in the same way. We are no longer required to describe states and preconditions only with positive literals, giving us more freedom to precisely represent the world; on the other hand, effects have now added complexity. Taking an action from our STRIPS example (here translated to ADL):

```
Action(
    Move(X: Room, Y: Room),
    Precondition: At(X) ∧ X != Y,
    Effect: ¬At(X) ∧ At(Y)
)
```

In STRIPS, the effect ¬At(X) ∧ At(Y) meant that we would add At(Y) to the generated state and delete At(X) from the generated state. In ADL, since negative literals are represented explicitly in states, the effect means we add ¬At(X) and At(Y) to the new state, and delete At(X) and ¬At(Y) (the negation of the literals in the effect) from it.

We can see some of the additional functionality of ADL in this example. When declaring the action, we can give information about the type of each variable – here, X and Y are both objects of the type Room. ADL supports boolean expressions, which in this case are used to ensure that X and Y are different rooms (and stop the planner from creating actions which move to the same room).

Other characteristics of ADL relate to the description of goals and effects. Goals in ADL do not need to be simply a conjunction of literals, but can also contain disjuction and quantified variables. Still following our example, imagine that our goal is now to clean the bedroom or the kitchen; this can be represented in ADL as:

```
Goal State: Tidy(Kitchen) ∨ Tidy(Bedroom)
```

Going further, if we wish to have cleaned at least one room by the end of the plan, our goal may be represented as:

```
Goal State: ∃X Tidy(X)
```

Actions in ADL may have conditional effects – this is used when there are literals that, while not being required to complete an action, their presence brings some additional effect. The syntax for conditional effects is:

*when Condition : Effect.*

Taking the Move operator we used before, suppose that we now consider the person that is changing rooms, and that if this person is dirty, then he or she will mess the room. We represent it as:

```
Action(
    Move(P: Person, X: Room, Y: Room),
    Precondition: At(P,X) ∧ X != Y,
    Effect: ¬At(P,X) ∧ At(P,Y) ∧ when  ¬Tidy(P) :  ¬Tidy(Y)
)
```

Notice we want to express that when a person is dirty (not tidy) then the room also becomes dirty: that is why we use negative literals in the condition.

ADL is a theoretical notation; in order to express planning problems in the ADL notation, another language must be used. One of the options is PDDL[18] – a language used to standardize different frameworks for planning into the same syntax, providing combined functionality and an universal approach. PDDL supports ADL, STRIPS and HTN, an alternative planning method in which the goal is to execute a set of tasks.

### 2.2.4 Scheduling

The planning methods we have seen until now dealt only with the sequencing of actions in a plan. For most real world problems, that is not enough – plans may need to take into account **time** (when is an action executed, and how long does it take) and **resources** (each action is allocated to a resource). Such problems where time and resources are involved are called **scheduling problems**.

Essentially, when representing time, one defines the start and end instants of a task. This allows us to perform **interval algebra** on tasks. Interval algebra uses seven different basic operators, which will be described here using the tasks $i$ and $j$:

- *$i$ before $j$*: $i$ ends before $j$ starts.
- *$i$ meet $j$*: $i$ ends in the same instant $j$ starts.
- *$i$ overlap $j$*: $i$ starts before $j$, and ends while $j$ is being performed.
- *$i$ start $j$*: $i$ and $j$ start at the same time, and $i$ ends before $j$ ends.
- *$i$ during $j$*: $i$ starts after $j$ starts, and ends before $j$ ends.
- *$i$ finish $j$*: $i$ starts after $j$ starts, and $i$ and $j$ end at the same time.
- *$i$ equal $j$*: $i$ and $j$ start and end in the same instant.

Each of the operators (save for *equal*) have a reverse operator; for example, *$i$ after $j$* is the same as *$j$ before $i$*.

With these operators, we can define many complex order relationships between tasks that would be otherwise impossible, giving us control over when an action is executed. As for task duration, since we now can determine the length of a task from its start and finish time, we can use basic mathematical operators to control the duration of a set of tasks.

One class of scheduling problems which is particularly relevant to this work is **machine scheduling**[20]. In machine scheduling, we use the concepts of **task**, **job** and **machine**. A task is the equivalent to an action for scheduling problems. A job is a set of related tasks, which may have a defined order of execution. A machine is a resource, which is used to perform a task. Generically, a machine scheduling problem consists in, given a number of jobs and machines, schedule the execution of the jobs, defining for each task which machine is going to perform the task and when, under these conditions:

- A machine cannot perform two tasks at the same time.
- Tasks of the same job cannot be performed at the same time.

Note that different machines and different jobs are always independent; the constraints influence only a single job/machine.

Variants of machine scheduling specify whether the tasks in each job have to be executed in a strict order, or exclusively in a single machine. In **job-shop problems**, both restrictions apply; each job must be performed in a specific machine, and in a specific order. **Flow-shop problems** go further and, in addition to job-shop restrictions, they constrain the number of tasks in a job to be equal to the number of machines; also, for every job, the first task is performed on the first machine, the second task on the second machine, and so on. **Open-shop problems** work in the same way, with the difference that the order restriction is relaxed. In **parallel machines problems**, each job has only one task, which can be performed in any machine.

We will now give an example of a machine scheduling problem. In this example, there are two jobs (job 1 and job 2) and two machines (machine A and machine B). Each job is composed of a sequence of

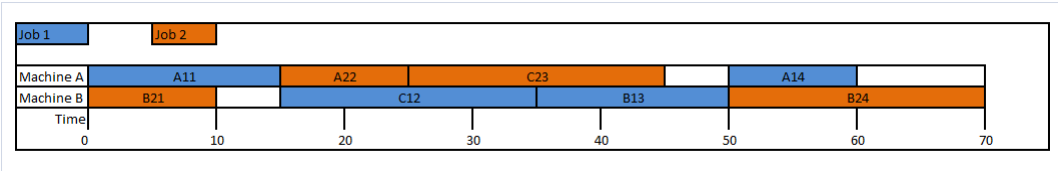


Figure 2.2: Solution of machine scheduling example

tasks. There are tasks of type A and type B, which can only be scheduled to the corresponding machine, and tasks of type C, which can be scheduled to any machine.

The following is the list of tasks of each job. The duration of each task is referred between parenthesis.

```
Job 1: A11 (15), C12 (25), B13 (15), A14 (10)
Job 2: B21 (10), A22 (10), C23 (20), B24 (20)
```

Figure 2.2 shows an optimal solution for the example problem. Notice that, although both jobs have the total duration of 60 minutes, the optimal solution has the duration of 70 minutes, due to the additional constraints over the tasks and machines.

## 2.3 Answer Set Programming

Answer Set Programming (ASP) is a logic programming paradigm, which consists in computing sets of literals (**answer sets**) that solve a given logic program. ASP has close ties with non-monotonic reasoning and default logic. Programs in ASP are written in a truly declarative way, with a syntax similar to that of Prolog.

The origin of ASP semantics goes back to the stable model semantics, defined by Gelfond and Lifschitz[17]. Years of work in the area led to Marek and Truszczynski[25] and Niemelä[26] establishing ASP as new paradigm of Logic Programming. Their work was consolidated by Baral[2] in a book which is a major reference in the area of ASP.

### 2.3.1 Answer Set Semantics

An ASP program is a set of rules. We start by presenting the basic elements that allow to write programs.

A **term** is a constant, a variable or a function where all its arguments are also terms. Variables start with an uppercase character, functions and predicates start with a lowercase character, constants start either with a lowercase character or with a numeric character.

**Definition 1** *An atom is an expression of the form*

$$p(t_1, ..., t_n)$$

*Where $p$ is a predicate symbol and $t_1, ..., t_n$ are terms.*

A **literal** is an atom or its negation.

**Definition 2** *A rule is an expression of the form*

$$l_0 \leftarrow l_1, ..., l_m, not\ l_{m+1}, ..., not\ l_n$$

*Where each $l_k$ is a literal.*

This is a simplified definition of a rule, similar to the definition of **clause** used in logic – rules and clauses are basically the same thing, but used in different situations (the term clause is used in logic, and the term rule is used in programs) – and does not cover cardinality constraints and weight constraints (which will be explained later). Rules are implications – in the above rule, if each literal $l_1, ..., l_m$ is true and each literal $l_{m+1}, ..., l_n$ can be assumed to be false, then the literal $l_0$ is true.

Note that there are two different types of negation. **Strong negation** represents that there is factual knowledge that an atom is false, and is expressed as $-a$ for an atom $a$. **Weak negation** represents that it was not proven that a literal is true, so it is assumed to be false. Weak negation is expressed as *notl* for a literal $l$ – $l$ itself may be a positive or negative literal, that is, $a$ or $-a$.

Some concepts related to rules:

- The left side of the rule is called the **head**, and the right side of the rule is called the **body**. The head of a program is the union of the heads of all its rules.
- **Facts** are rules in which the body is empty, and represent literals that are always true.
- **Constraints** are rules in which the head is empty, and represent a conjunction of literals that cannot be true.

**Cardinality constraints** (also called *choice rules*) and **weight constraints** give ASP an additional layer of expressiveness. Cardinality constraints are expressions of the form

$$m\ \{l_1, ..., l_k\}\ n$$

The expression means that, from the set of literals inside the clause, at least $m$ and at most $n$ are true. When $m$ or $n$ are not mentioned, it means there is no lower or higher bound, respectively. Cardinality constraints can be used in the head and in the body. The following are some examples of the use of cardinality constraints:

```
1 { hasDog(X), hasCat(X), hasBird(X) } 1 :- person(X).
```

This rule means that one person must have exactly one dog, one cat or one bird.

```
goodPicture(X) :- picture(X), 2 { yellow(X), red(X), blue(X), green(X), orange(X) }
```

This rule means that a picture must have at least two colours from the set represented above in order to be a good picture.

Cardinality constraints are often used together with conditional expressions:

```
1 { owner(X,Y) : animal(Y) } 1 :- person(X).
```

This rule means that one person must be owner of exactly one animal.

The set of literals inside the cardinality constraint is obtained by creating one atom $owner(X, Y)$ for every $Y$ which is an animal. We can say that the condition represents a domain for a variable which is not used in the body of the rule. As such, in the rule above, $animal(Y)$ is a local variable, processed in the grounding stage, that does not appear in the ground program. Assuming that the domain of the *animal* predicate is the following:

```
animal(dog). animal(cat). animal(bird).
```

In that case, the conditional expression above is an abbreviation for:

```
1 { owner(X,dog), owner(X,cat), owner(X,bird) } 1 :- person(X).
```

This is a conditional expression because it imposes a constraint on the $Y$ variable: in this example, $Y$ must be an animal. In general, the right-hand side of the conditional expression can only contain domain predicates or built-in expressions (for example, comparison predicates).

Weight constraints are expressions of the form

$$m\ [l_1 = w_1, ..., l_k = w_k]\ n$$

The values $w_i$ are **weights** for each literal. The weight constraint means that the sum of the weights for the set of true literals in the expression must be at least $m$ and at most $n$. This is essentially a generalized version of cardinality constraints. If $m$ or $n$ are not defined, then there is no lower or higher bound. If some value $w_i$ is not defined, then it is assumed that the weight for the corresponding literal is 1. For example:

```
goodPicture(X) :- picture(X), 3 [ yellow(X), red(X), blue(X)=2, green(X)=3, orange(X)=2 ]
```

A green picture has a weight of 3, so is considered a good picture; a blue and orange picture has a total weight of 4, so it is also a good picture, but a yellow and red picture is not a good picture because the sum of the weights is 2.

ASP supports arithmetic functions and comparison predicates in its rules. Values modified by arithmetic functions can be used as long as the original value is present in a domain predicate (for example, $p(X) :- q(X+1)$ is not supported, but $p(X-1) :- q(X)$ is, because in the latter X is defined in the body).

**Intervals** allow to write several predicates in a clean, easy way. An interval between $x$ and $y$ is written as $x..y$. For example,

```
number(1..5).
```

is short for

```
number(1). number(2). number(3). number(4). number(5).
```

### 2.3.2 Grounding and solving

The process of obtaining the solutions from a program is divided in two stages: **grounding** and **solving**.

A few concepts needed to grasp the grounding process:

- A term is **ground** if there are no variables in it. Furthermore, atoms and literals are ground if the terms they are composed of contain no variables.
- The **Herbrand Universe** (HU) of a program is the set of ground terms that can be constructed from the rules in the program (i.e., the constants and the functions).
- The **Herbrand Base** (HB) of a program is the set of all ground atoms that can be constructed using predicates from the program and terms from the HU.

In what follows, all the code and syntax rules follow the syntax of **gringo**[10], an ASP grounding tool, unless otherwise stated.

For example, given the following program:

```
p(1).
p(2).
p(f(X)) :- p(X).
```

The HU of the program is the set:

$$\{1, 2, f(1), f(2), f(f(1)), f(f(2)), ...\}$$

And the HB of the program is the set:

$$\{p(1), p(2), p(f(1)), p(f(2)), p(f(f(1))), p(f(f(2))), ...\}$$

In the grounding stage, a **ground program** is produced. A ground program is a program with all its rules grounded. If the rule is already ground, that is, if all the literals in the rule are ground, then there is no need to change it; otherwise, we compute the ground rule as a set of rules with all possible substitutions of variables in the rule by elements of the HU[2].

For the previous example, the ground program will be composed of the rules:

```
p(1).
p(2).
p(f(1)) :- p(1).
p(f(2)) :- p(2).
p(f(f(1))) :- p(f(1)).
p(f(f(2))) :- p(f(2)).
p(f(f(f(1)))) :- p(f(f(1))).
p(f(f(f(2)))) :- p(f(f(2))).
...
```

Notice that this ground program has an infinite number of rules, so the grounding process would never finish.

In the solving stage, the ground program is used to produce the **answer sets**. An answer set is a set of elements of the HB that satisfies all the rules in the ground program, and that can be derived from those rules – every element in the answer set must be in the head of a rule of the ground program. If a program is composed only by rules of the form $l_0 \leftarrow l_1, ..., l_m$, then it has one answer set if it is satisfiable, and zero answer sets if it is not satisfiable. Otherwise, the program may have multiple answer sets.

The Gelfond-Lifschitz transformation[17] is a method used to compute answer sets of a program. This method is described below in the form of an algorithm.

**Definition 3** *(Gelfond-Lifschitz transformation)*

*Given a ground program P:*

- *Step 1: Create a* ***candidate set*** *S, which is an arbitrary subset of the HB of P.*
- *Step 2: Delete every rule in P that has an expression of the type not L in its body, where L is a literal of an atom in S.*
- *Step 3: Delete every expression of the type not L of the remaining rules in P.*

*If the resulting program has an answer set and this answer set coincides with S, then S is an answer set of P.*

By repeating this process for all subsets of the HB, we can find all the answer sets of a program. Modern ASP solvers, though, use more advanced techniques in order to optimize solving performance.

### 2.3.3 Programming methodology

There is not a universal, mandatory methodology to solve problems using ASP, as the language allows different approaches to be taken. However, there are some techniques used by experts that can be taken as guidelines to help the programmer. One method in particular, **generate-and-test**[2, 4, 34, 12], is commonly used as a standard. We will illustrate generate-and-test with the development of a program that solves Langford's problem, and later we will present some other techniques.

Langford's problem[9, 24] consists in finding a sequence of numbers of length $2n$, in which the numbers $1, 2, ..., n$ appear twice, and for each pair of a number $k$, there are exactly $k$ numbers between them in the sequence (the two 1's must be one position apart, the two 2's must be two positions apart, and so on). For example, for $n = 4$, a solution for this problem is:

4 1 3 1 2 4 3 2

We will start by defining the domain of the problem. This is the data that defines an instance of the problem. The encoding of an ASP program should be uniform (independent of the instance), and as such, it is a good practice to write the facts and the rules in two separate files. This way, we can test different instances by compiling the rules file together with one instance file.

To represent the domain, we need to describe the numbers of the sequence, and the positions they can be at. Let $value(N)$ be the encoding for a number and $position(I)$ the encoding for a position. The instance for $N = 4$ will be:

$$value(1..4). \tag{2.1}$$

$$position(1..8). \tag{2.2}$$

The next step is to generate the solution space in which the ASP solver is going to search. We define the predicate $valueAtPosition(N, I)$, which means the number $N$ is at position $i$. Our solution will be a set of these predicates, which together describe the full sequence. The following is a first approach to a generator rule:

$$\{valueAtPosition(N, I)\} :- value(N),\ position(I). \tag{2.3}$$

This rule will produce answer sets with an arbitrary amount of predicates $valueAtPosition(N, I)$. However, we know every solution will have the same size – there will be exactly one value at each position, that is, from each predicate $position(I)$ we can derive a predicate $valueAtPosition(N, I)$, in which $N$ comes from a predicate $value(N)$. So, we will discard 2.3 and write the following rule:

$$1\{valueAtPosition(N, I) : value(N)\}1 :- position(I). \tag{2.4}$$

This rule will generate solutions as sequences in which there is exactly one number at each position. Now we need to specify to the solver which of these solutions are correct – this is the *tester* part of the program. But before going further, notice that we can use the same mechanism as in (2.4) to constrain the search space to generate only solutions in which each number appears exactly twice. This is described by the rule:

$$2\{valueAtPosition(N, I) : position(I)\}2 :- value(N). \tag{2.5}$$

With these rules, the program produces answer sets corresponding to sequences with appropriate length and correct numbers; yet, the order of the sequence was not specified. We want to make sure that pairs of equal numbers are in the correct position relative to each other. The rules of the problem state that a pair of numbers $k$ must have $k$ positions between them – that is, if one element is at position $I$, then the other must be at position $I + k + 1$ or at position $I - k - 1$. We will express this rule in ASP as follows:

$$:- valueAtPosition(N, I),\ valueAtPosition(N, J),\ I > J,\ I\ != J + N + 1. \tag{2.6}$$

Given that two similar values are at positions $I$ and $J$, and assuming $I > J$, $I$ must be equal to $J + N + 1$. In 2.6 we state that there is a contradiction if $I$ and $J + N + 1$ are not equal. We do not need to write an additional rule for the opposite case (in which $I < J$), because that corresponds to exchanging $I$ and $J$, which is already covered in this rule.

Once a basic and working encoding is established, one can delve further into optimizations:

- Mancini et al.[24] suggest the use of symmetry-breaking rules as an attempt to optimize an ASP encoding. There are several ways to break symmetries in ASP programs. Note that it depends on the problem in hand whether a method is applicable or not, and also whether it helps to improve performance: in some cases the additional constraints actually harm performance. Amongst the symmetry-breaking techniques are the assignment of some values beforehand and fixing the relative order of some elements. If successful, these techniques will cut down large parts of the search tree, greatly improving solving time.

- Gebser et al.[12] use the database-inspired technique of projection, which consists in transforming predicates with a large number of variables into smaller predicates, using the abbreviated versions in rules that do not need the extra variables, thus easing the process of grounding those rules. It will be benefitial if the time gained computing the rules overcomes the time lost grounding the new predicates.

### 2.3.4 ASP Tools

Through its short history there were plenty of tools developed for ASP already. Grounders and solvers have evolved in time, with increasing functionality and performance.

**Gringo**[16] and **clasp**[14] are the state-of-the-art tools for grounding and solving in ASP, respectively. These tools were developed in 2007, at the University of Potsdam. We will describe gringo and clasp in more detail later. Gringo and clasp, along with other ASP tools based on them, are available for download[3].

**Smodels**[31] is the oldest ASP tool. Its development started in 1995, at the Helsinki University of Technology. The smodels tool is based on the stable model semantics and includes a solver, *smodels*, and a grounder, *lparse*, which are publicly available[4].

**DLV**[21] is a tool developed to solve disjunctive logic programs for databases, but its semantics also allows the computation of ASP programs. It was developed in 1997, at the Vienna University of Technology, and is publicly available[5]. DLV includes its own grounder and solver tools, as well as a dedicated tool for planning problems, named *DLVk*.

Similarities between ASP and SAT led to the development of the SAT-based solvers **ASSAT**[23][6] and **cmodels**[22][7]. These solvers use lparse to ground the ASP programs, convert the grounded program to SAT and then delegate the solving task to a SAT solver.

#### Gringo

Gringo[16] is the most recent grounder for ASP programs. Gringo combines the features of other two ASP grounders, *lparse* and *dlv*, improving on both of them. Gringo accepts $\lambda$-restricted programs, as opposed to lparse's $\omega$-restricted programs[33]. This allows for the inclusion of cyclic definitions of predicates as long as there is one rule in the cycle that ties the variables used to some domain predicates. In lparse, in order to compute cycles, it is mandatory to include domain predicates for every variable, in every rule. Gringo also evolved from DLV's grounding algorithm, by separating relevant and irrelevant variables in a rule, thus preventing the creation of the same grounded rule several times.

Part of gringo's input language was already discussed in section 2.3.1 – Answer Set Semantics. Here we describe some additional commands that extend the ASP language[10].

One useful to abbreviate predicates is the **;** operator, which separates several values for the same argument in a predicate. For example,

```
father(John,Lewis;Mary;Anthony).
```

expands into

[3] http://potassco.sourceforge.net/
[4] http://www.tcs.hut.fi/Software/smodels/
[5] http://www.dlvsystem.com/
[6] http://assat.cs.ust.hk/
[7] http://www.cs.utexas.edu/users/tag/cmodels/

```
father(John,Lewis). father(John,Mary). father(John,Anthony).
```

Gringo inherits support for **optimization** statements from lparse. The existent optimization statements are *#maximize* and *#minimize* and allow to choose, between several answer sets of a program, which one is optimal. For example,

```
#minimize[trash(X) = X]
```

minimizes the sum of the values $X$ of predicates $trash(X)$ in the answer set.

In order to ease the task of reading the output of an answer set program, we can use the statements *#hide* and *#show* in order to determine which literals are hidden and shown in the output. These statements are processed by their order in the file. For example,

```
#hide.
#show valueAtPosition/2.
```

will first hide every predicate from the output, and then show the predicate valueAtPosition with 2 arguments.

Lastly, some useful input options for gringo are listed below[10]:

- **-t**: Prints the output ground program in readable text format.
- **−verbose=3**: Prints the representation of rules during grounding (useful for debugging).
- **-c x=t**: Attributes to the constant $x$ the value $t$.

### Clasp

Clasp[14] is an ASP solver whose main strength is that it was designed to work with conflict-driven clause learning. This is a technique used in SAT (satisfiability checking) solvers, which consists in learning a clause whenever the solver finds an inconsistency after assigning a set of values. This clause prevents the solver from assigning such combination of values again. Usually, the learnt clause consists in a constraint which is the conjunction of all the assigned values – this leads to the creation of large clauses, in which many of the literals are irrelevant. Instead, clasp further optimizes the process of learning a clause by adopting the concept of *nogood*, used in the CSP(constraint solving programs) area. A nogood is a minimal set of attributions which cannot be present in any solution, so that if one variable is removed from the nogood, the restriction no longer holds. By using nogoods, the clauses learnt by clasp are much more compact, as they contain only the relevant variables that led to the inconsistency in the solving process. Clasp's efficiency allowed it to win several ASP and SAT competitions[13].

Clasp is equipped with plenty of options[10] that allow the user to optimize solver performance. It is often useful to try different values for those options, to find out which setup is better to solve a specific problem.

- **–heuristic=<Value>**: Uses the heuristic determined in $Value$. The values allowed are *Berkmin, Vmtf, Vsids, Unit* and *None*.
- **–trans-ext=<Value>**: Determines whether cardinality and weight constraints are converted into boolean rules. The values allowed for $Value$ are *choice* (converts only choice rules), *weight* (converts only weight rules), *all* (converts all rules), *dynamic* (follows an heuristic to determine if a rule should be converted) and *no* (does not convert any rules).
- **–sat-prepro**: Enables SAT-preprocessing. Particularly useful when using the option *–trans-ext*, as SAT-preprocessing will clean eventual unnecessary boolean rules generated by that process.

Other options of clasp that are useful overall:

- **–quiet**: Does not print the output answer sets (recommended for benchmarking).
- **-n <n>**: Outputs the first $n$ answer sets. Default value for this option is 1. $n = 0$ prints all answer sets.
- **–brave / –cautious**: Computes the brave (union of all answer sets) and cautious (intersection of all answer sets) answer set, respectively.
- **–time-limit=<t>**: Program stops after $t$ seconds.

**iClingo**

Usually, when working on planning problems, one does not know the size of the solution beforehand. In order to get the best plan possible, which is, for example, the plan that requires the shortest amount of time to solve the problem, we need to progressively increase the time variable, until we find a solution. When using ASP, that means we have to call the grounder and solver once for each value of the variable. However, doing so is a waste of resources, because many ground rules are repeated between iterations. This motivated the creation of a tool that could optimize the problem of solving iterative programs in ASP.

iClingo [11] is a tool designed to solve iterative problems. It was created in 2008, based on gringo and clasp. The key element of iClingo is the use of incremental logic applied to ASP.

Incremental logic programs are divided into three parts:

- **Basic** rules ($B$) are rules which do not depend on the parameter over which the problem iterates.
- **Cumulative** rules ($P[k]$) are rules generated at the iteration $k$ and that maintain their validity in future iterations.
- **Volatile** rules ($Q[k]$) are rules which are valid only for iteration $k$, being meaningless in other iterations.

At iteration $i$, the incremental logic program contains the rules in $B$, all rules $P[k]$ for $1 \leq k \leq i$, and rules $Q[i]$.

In order to distinguish the different kind of rules, iClingo includes the commands *#base*, *#cumulative <variable>* and *#volatile <variable>*, in which *variable* is the name of the variable over which the program iterates. An example of an incremental ASP program (credited to Gebser et. al.[13]):

```
#base.

holds(P,0) :- init(P).

#cumulative t.

1  occ(A,t) : action(A)  1.
:- occ(A,t), pre(A,F), not holds(F,t-1).

holds(F,t) :- holds(F,t-1), not ocdel(F,t).
holds(F,t) :- occ(A,t), add(A,F).
ocdel(F,t) :- occ(A,t), del(A,F).

#volatile t.

:- query(F), not holds(F,t).
```

Below are some input options for iClingo[10] which allow the user to adapt its behaviour:

- **–ilearnt**: Chooses whether to *keep* or *forget* learnt constraints between iterations.
- **–iheuristic**: Chooses whether to *keep* or *forget* heuristic information between iterations.
- **–imin=n/–imax=n**: Imposes a minimum or maximum limit of iterations to execute.
- **–iquery=n**: Starts solving only for the parameter value $n$ (note that cumulative rules for earlier iterations are still computed).

iClingo inspired the creation of a system for Finite Model Computation, **fmc2iasp**[15].

### ASPviz

ASPviz [5] is a tool that tackles a problem inherent to the current ASP solvers: the lack of a graphical representation. ASP solvers only give a textual output, showing solutions as long strings of literals – for large problems this makes it hard to verify and debug programs. A graphical representation, on the other hand, can be much more intuitive when done correctly. ASPviz is available for download[8].

[8] http://www.cs.bath.ac.uk/~occ/aspviz/

ASPviz takes as input the output given by an ASP solver and an additional ASP program that describes which graphical commands will be used, as well as under which conditions. ASPviz's graphical primitives are ASP atoms themselves, so they can be generated the same way one computes answers for a regular ASP program.

The basic commands needed to write an ASPviz program are:

- **brush(BrushName)**: defines a brush named $BrushName$. The brush is used to draw lines and figures. A brush can be modified using the commands **brush_width(Brush,Width)** and **brush_color(Brush,Color)**.
- **font(FontName)**: defines a font named $FontName$. The font is used to draw text. A font can be modified using the commands **font_style(Font,Style)** and **font_color(Font,Color)**. $Style$ takes the values *bold* and *italic*.
- **color(ColorName,rgb(R,G,B))**: defines a color named $ColorName$ using its RGB values. Colors may be used to personalize brushes and fonts. Some standard colors (such as black, white and blue) are already defined in ASPviz library.
- **p(X,Y)**: defines a point in the coordinates (X,Y). Points are used to determine the position of other drawing commands.

Usually the commands specified above will be facts of the ASPviz program (except for $p(X, Y)$).

With the above commands, we can define the actual drawing primitives:

- **draw_line(B,P1,P2)**: draws a line from point $P1$ to point $P2$ using the brush $B$.
- **draw_text(F,c,c,P,Text)**: draws the string $Text$ centered on point $P$ using the font $F$. Replacing $c$ with anything else aligns the string to the left starting on point $P$ instead.
- **draw_rect(B,P,W,H)**: draws a rectangle starting on point $P$, with width $W$ and height $H$, using the brush $B$. A similar command **fill_rect(B,C,P,W,H)** fills the inside of the rectangle with the color $C$.
- **draw_ellipse(B,P,W,H)**: draws an ellipse centered on point $P$, with width $W$ and height $H$, using the brush $B$. A similar command **fill_ellipse(B,C,P,W,H)** fills the inside of the ellipse with the color $C$.

These commands are used in rules of the ASPviz program, where the head is the drawing predicate and the body is a literal or sequence of literals from the output of the program.

We will illustrate ASPviz with a program used to draw solutions for the Kakuro problem.

**Kakuro** is a puzzle game played on a grid. The grid has black cells, white cells and *clue* cells. The goal of the game is to write one number from 1 to 9 in each white cell, in such a way that, for each line

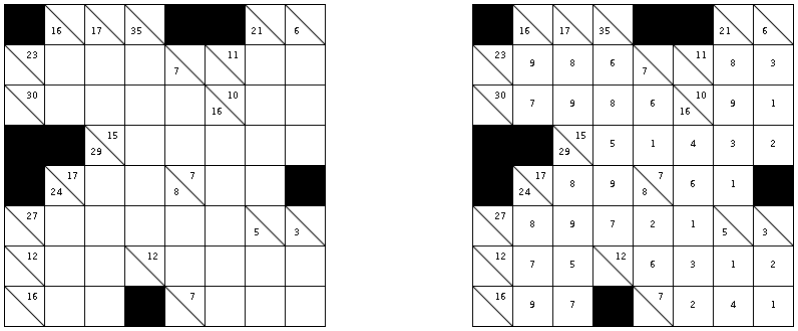


Figure 2.3: Kakuro puzzle and its solution

of continued white cells, the sum of the values of its cells corresponds to the clue given in the clue cell, and every cell in the line has a different number. The clue cell for each line is to its left, if the line is in a row, or up, if the line is in a column.

The following is an ASPviz program which draws the image seen in Figure 2.3 for an instance of the Kakuro problem.

```
brush(standard).
brush_color(standard,black).
brush_width(standard,1).

font(myFont).
font_size(myFont,20).

draw_rect(standard,p((X-1)*40,(Y-1)*40),40,40) :- pos(X,Y).
fill_rect(standard,black,p((X-1)*40,(Y-1)*40),40,40) :- black(X,Y).
fill_rect(standard,white,p((X-1)*40,(Y-1)*40),40,40) :- pos(X,Y), not black(X,Y).

draw_text(myFont,c,c,p((X-1)*40 + 20,(Y-1)*40 + 20),N) :- num(N,X,Y).

draw_line(standard,p((X-1)*40,(Y-1)*40),p(X*40,Y*40)) :- diagS(S,X,Y).
draw_line(standard,p((X-1)*40,(Y-1)*40),p(X*40,Y*40)) :- diagI(S,X,Y).

draw_text(myFont,c,c,p((X-1)*40 + 12,(Y-1)*40 + 28),N) :- diagS(N,X,Y).
draw_text(myFont,c,c,p((X-1)*40 + 28,(Y-1)*40 + 12),N) :- diagI(N,X,Y).
```

Notice that, although we cannot draw the Kakuro puzzle perfectly (clue cells with one clue should have a black triangle filling the half of the cell which is unused), this is a better way of representing a solution than simple textual output.

# 3 Solution Development

In this chapter we present several possible solutions to the crew planning problem. While researching the subject, we noticed there was not a single clear definition for the problem, so we worked under two different formulations: the "standard model" and the "competition model". They represent two different ways of approaching the crew planning problem. The standard model is a formulation that we believe represents the problem in a logical way and in a way that is adequate for the use of ASP. On the other hand, the competition model follows closely the domain model used in the IPC (International Planning Competition)[1].

Although we separate our work in such a way, there is still a strong connection between the two parts. Many names are common to both versions, some files are shared, some programs are similar and may contain identical rules. The distinction lies in details that may not look substantial, but that have deep consequences in the way ASP works. Note that ASP is a declarative language and, because of this, more emphasis is given to details.

## 3.1 Plan constraints

In section 2.1 – Crew Planning we described the tasks that may appear in a crew schedule. Some of these tasks have been changed, or are not present at all in this formulation. Therefore, we will list again the tasks we use in the formulation of a solution, as well as the constraints regarding each task. The list of tasks is common to both the standard model and the competition model.

The basic plan, with activities that are mandatory for every crew member to perform every day, is now composed of:

- A **post-sleep period** of 195 minutes (3 hours and 15 minutes), at the beginning of the day.
- A **sleep period** of 600 minutes (10 hours), at the end of the day.
- An interval for **lunch** of 60 minutes (1 hour). The lunch period can be scheduled anywhere between the post-sleep period and the sleep period.
- An **exercise** period of 60 minutes. A period of exercise is associated with a machine, and each machine can be used by only one person at a time.

[1] The domain file can be downloaded in http://ipc.informatik.uni-freiburg.de/Domains

The tasks in the basic plan are the only mandatory tasks in a schedule. Every other task must be specified in the problem instance in order to be present in the schedule.

A **payload** task (formerly referred to as a **science experiment**) is a task with the duration of 60 minutes, and a deadline which limits the latest day at which the task can be performed. Payload tasks can be performed by any crew member, on any day up to the deadline.

A **medical conference** has the duration of 60 minutes. This task is performed by a specific crew member in a specific day. This information is stated in the problem instance.

A **filter change** task has the duration of 60 minutes. The task is scheduled for a specific day, but can be performed by any crew member.

The complex task regarding the replacement of the **Remote Power Controller Module** (RPCM) is present, with three of the smaller tasks consolidated into one; "Remove Power Source", "Replace RPCM" and "Place Power Source" become a three-hour task "Replace RPCM", with the sequence of tasks becoming now:

**RPCM sequence**

1. - Reconfigure Thermal Loops
   - Remove Sleep Station
2. Replace RPCM
3. - Reconfigure Thermal Loops
   - Assemble Sleep Station

All tasks have the duration of 60 minutes, with the exception of *Replace RPCM* which has a duration of 180 minutes. Each task in the sequence must be finished before the next task in the sequence starts. Each task can be performed by any crew member independently. The whole sequence of tasks must be completed until an established deadline day.

Figure 3.1 shows an example plan of one day for one crew member, which contains the mandatory activities from the basic plan, as well as one sequence of RPCM tasks and one payload task. The blank space in the figure represents time that is not allocated to any task.

### 3.1.1 Differences specific to the competition model

The competition model was written in the PDDL language, using only STRIPS-like constraints. Because of this, it has some differences with respect to the standard model that come from limitations in the expressibility of STRIPS. Days do not have a fixed length. Instead, there is a minimum amount of hours required to have passed since the beginning of the schedule in order to advance to the next day, corresponding to 24 hours for each previous day (Day 2 cannot start after at least 24 hours, Day 3 after 48 hours, and so on). To further explain this rule, we will present some possible scenarios (note that these scenarios ignore the existing task-related constraints, and are used only to explain this rule):

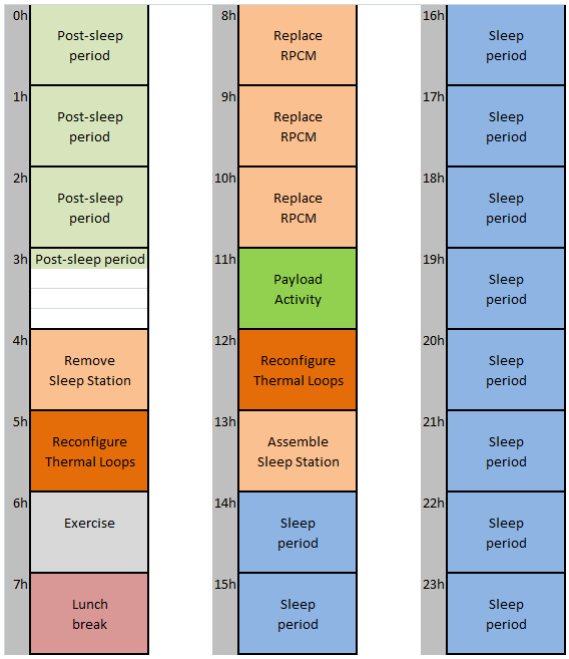


Figure 3.1: Schedule of one day

- A 2-day schedule where both days last 36 hours is valid (Day 2 starts after 36 hours).
- Any schedules in which all days have a length of at least 24 hours are valid.
- A 1-day schedule with the length of 16 hours is valid (days can last less than 24 hours).
- However, a 2-day schedule in which Day 1 lasts 16 hours and Day 2 lasts 30 hours is not valid, because Day 2 starts before 24 hours have passed.
- A 3-day schedule in which Day 1 lasts 36 hours, Day 2 lasts 12 hours and Day 3 lasts 24 hours is valid. Independently of how short Day 2 is, Day 3 starts after 48 hours have passed and thus this schedule follows the rule.

Not only days have variable lengths among themselves, but also crew members can work in different shifts. So the same day can have different lengths for different people. There are no constraints related to this, i.e. it is irrelevant to each crew member where the rest of the crew stands in their schedules.

## 3.2 List of Predicates

In what follows is given a comprehensive list of all the predicates used in the proposed encodings. Note that most encodings do not use all of the predicates in this list.

General facts:

- *crewMember(C)*: $C$ is a crew member.
- *day(D)*: $D$ is a day (numeric domain).

- *equipment(E)*: $E$ is an exercise equipment.
- *period(P)*: $P$ is a period. A period is the base unit of time in the context of the program, and represents a time interval of 15 minutes.
- *duration(Type,Time)*: Task of type *Type* has duration *Time*. The duration of each type of task is measured in periods of time. For example, a task which takes 1 hour will have a duration of 4 periods.
- *maxperiod(P)*: $P$ is the last period allowed in a plan.
- *dayone(TD1)*: The time spent in tasks scheduled to day 1 is at least *TD1* periods.
- *daytwo(TD2)*: The time spent in tasks scheduled to day 2 is at least *TD2* periods.
- *daythree(TD3)*: The time spent in tasks scheduled to day 3 is at least *TD3* periods.

Task-related facts:

- *task(Tid)*: *Tid* is the ID of a given a task (numeric domain).
- *taskName(Tid,Name)*: The task *Tid* has name *Name*. This predicate has no influence in the generation of the plan, being used only to guarantee that a task can be identified more easily.
- *taskType(Tid,Type)*: The task *Tid* is a task of the type *Type*.
- *taskDeadline(Tid,Day)*: The deadline of the task *Tid* is in day *Day*.
- *knownDay(Tid,Day)*: The task *Tid* will be executed on day *Day*.

RPCM facts:

- *isRPCM(Tid,R)*: The task *Tid* is part of the RPCM sequence $R$.
- *stageRPCM(Tid,S)*: The task *Tid* occurs in the stage $S$ of a RPCM sequence.

Answer predicates:

- *execute(C,Tid)*: The crew member $C$ will perform task *Tid*.
- *taskTime(Tid,P1,P2)*: The task *Tid* begins at period *P1* and ends at period *P2*.
- *taskDay(Tid,Day)*: The task *Tid* will be executed on day *Day*.
- *busy(E,Tid)*: The equipment $E$ is busy while task *Tid* is underway.
- *taskPerform(Tid,C,P1,P2,D)*: The task *Tid* will be performed by crew member $C$, beginning at period *P1* and ending at period *P2* of day $D$.

## 3.3 Instance Encoding Example

We will now show how to encode an instance of the planning crew problem with the predicates in section 3.2 – List of Predicates, using the same example as before: a plan of one day for one crew member, that includes a RPCM procedure and a payload activity. This instance is valid for the encoding described in section 3.4 – Standard model: Basic encoding. For other encodings, changes to this model instance are described in an appropriate subsection.

$$day(1). \tag{3.1}$$

$$crewMember(c1). \tag{3.2}$$

$$equipment(e1). \tag{3.3}$$

These predicates determine how many days long the plan is and how many crew members are involved. The exercise equipments available to use in the plan are defined next.

$$task(1). \tag{3.4}$$

$$taskName(1, \text{"}\textit{post-sleep-c1-1}\text{"}). \tag{3.5}$$

$$taskType(1, \text{"}\textit{post-sleep}\text{"}). \tag{3.6}$$

$$taskDay(1, 1). \tag{3.7}$$

$$execute(c1, 1). \tag{3.8}$$

$$taskTime(1, 1, 13). \tag{3.9}$$

The next step is to introduce information about the tasks that form the basic plan. Each task is referred to by its ID; in line (3.4) is introduced 1 as a valid ID for this task. The task is given the name *post-sleep-c1-1* in line (3.5). The name is used only to make the output easier to read, so this line can be omitted. In line (3.6) is determined the type of the task. This is used to differentiate the constraints applied to each task.

Given that this task is part of the basic plan of crew member c1, we add the lines (3.7) and (3.8). These lines state that task 1 is scheduled for day 1, and that crew member c1 executes that task. Depending on the type of task, this information may be determined by the ASP solver, instead of being given in the instance. Also, because this is a post-sleep task (which is performed at the start of the day), we introduce line (3.9) in the instance. It means that task 1 starts at period 1 and ends at period 13. This is the only possible time interval for this task, so it can be given as input.

We also introduce the sleep, exercise and meal tasks for crew member c1 on day 1, in a similar way to lines (3.4)-(3.8). For the sleep task a line similar to (3.9) is added, as it always starts at period 57 and

ends at period 96.

After writing down all the tasks from the basic plan, one can then add any other tasks to be included in the plan. In this case the tasks that make up a RPCM sequence will be added.

$$task(5). \tag{3.10}$$

$$taskName(5, "reconfigurate\text{-}thermal\text{-}loops\text{-}1\text{-}rpcm1"). \tag{3.11}$$

$$taskType(5, "reconfigurate\text{-}thermal\text{-}loops"). \tag{3.12}$$

$$isRPCM(5, rpcm1). \tag{3.13}$$

$$stageRPCM(5, 1). \tag{3.14}$$

$$taskDeadline(5, 1). \tag{3.15}$$

Two special facts to represent RPCM tasks are needed. With line (3.13), this task is binded to a particular RPCM sequence (which, in this case, has the code *rpcm1*), and with line (3.14) is determined that the task belongs to stage 1 of the sequence. In other words, among all tasks in the sequence, this task (and other tasks in stage 1) have to be executed first. In addition, is set a deadline for the task in line (3.15) that determines what is the last day in which the task can be executed. In this case there is no difference between this rule and a *taskDay* rule; however, if the deadline is, for example, 2, then the task can be executed in either day 1 or day 2.

Along with this task are introduced the other tasks of the *rpcm1* sequence, in a similar way to lines (3.10)-(3.15).

$$task(10). \tag{3.16}$$

$$taskName(10, "payload\text{-}pa1\_1"). \tag{3.17}$$

$$taskType(10, "payload"). \tag{3.18}$$

$$taskDeadline(10, 1). \tag{3.19}$$

The example instance also has a payload task. Other than its name and type, the only other piece of information about this task is its deadline.

The duration of each task is not present in the instance file. Instead, because the duration is fixed independently of the instance, we provide a different file with all the duration facts, which are of the form:

$$duration("payload", 4). \tag{3.20}$$

Line (3.20) shows that the duration of each payload task is 4 periods. By matching line (3.20) with line (3.18) the program infers that task 10 has the duration of 4 periods, and can attribute a time for the task accordingly.

## 3.4 Standard model: Basic encoding

In this section is presented the basic encoding of the crew planning problem as well as the reasoning behind it. We have previously listed the predicates used in the encoding and given an example instance based on this encoding. Now, the rules of the program will be presented, followed by the presentation of the resulting plan for the example instance. Finally, we show how the encoding evolved with some optimizations.

The encoding is focused on organizing the information about each task. Because there is a significant amount of information to handle and the amount of information varies depending on the type of task, each bit of information about a task is encoded using a different predicate. There is a code to refer to each task, that is present in all task-related predicates, so that one can easily access all the information about a task. The exception to this rule is the information concerning the duration of a specific task, which is grouped according to the type of task (for example, all exercise tasks have the same duration). Other than the information about tasks, one has to represent the days and crew members involved in the plan, the existing exercise equipments and the timespan of the plan. The resulting plan is obtained from a series of answer predicates (predicates that were generated by the solved with new information) that relate each task to each of the other existing domains. Due to the nature of some types of tasks, parts of the resulting plan may already be present as facts (for example, for a task that is part of a basic plan, we already know the day it is on and the crew member that will perform it).

### 3.4.1 Rules

We followed the generate-and-test [4] approach when writing the rules for our program. This approach consists in first generating the search space, to then apply the necessary constraints to filter out answer sets that are not solutions.

$$period(1..96). \tag{3.21}$$

The range of the domain predicate *period* is defined in (3.21). A value of 15 minutes is chosen to define the length of a period because it allows to represent accurately the duration of every type of task, and it is the biggest possible value that does so. With a smaller period (for example, 1 minute), one would have a bigger range of possible values and a larger grounded file, harming the performance of the program. In (3.9), the values 1 and 13 are used because the task starts at the first value of the domain (which is 1) and has the duration of 13 periods, ending at period 13. Note that 13 periods corresponds to 3 hours and 15 minutes.

$$1\{execute(C, Tid) : crewMember(C)\}1 \text{ :- } task(Tid). \tag{3.22}$$

Rule (3.22) states that each task is performed by a single crew member. We will use this rule to further explain how a "choice rule" works. Let us assume an instance with three tasks and two crew members:

```
task(1).
task(2).
task(3).
crewMember(Alice).
crewMember(Bob).
```

The grounder program starts by creating rules based on rule (3.22) for every possible task:

```
1 { execute(C,1) : crewMember(C) } 1 :- task(1).
1 { execute(C,2) : crewMember(C) } 1 :- task(2).
1 { execute(C,3) : crewMember(C) } 1 :- task(3).
```

Then, in each rule, the program fills the *execute* predicate with facts from the domain predicate *crewMember*:

```
1 { execute(Alice,1), execute(Bob,1) } 1 :- task(1).
1 { execute(Alice,2), execute(Bob,2) } 1 :- task(2).
1 { execute(Alice,3), execute(Bob,3) } 1 :- task(3).
```

In the end, the ground program has three rules, one for each task, that state that tasks 1, 2 and 3 can be performed by either Alice or Bob. The solver will then need to choose which one is true; hence the name *choice rule*.

$$\begin{aligned} 1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1)\}1 \\ \text{:- } task(Tid), taskType(Tid, T), duration(T, K). \end{aligned} \tag{3.23}$$

Rule (3.23) generates the possible *tasktimes*. A single *tasktime* is assigned to each task (that is, each task appears in the plan only once). The task must start and end at acceptable periods. Hence, the solver will need to assign to each task a period $P$ so that the period at which the task ends must be a valid period as well (*P+K-1*: the starting period $P$, plus the duration of the task $K$ associated to the type of task, minus 1).

Rules (3.22) and (3.23) do not interfere with the possibility of the predicates they generate being given as facts. In that case, the result of these rules will agree with the facts.

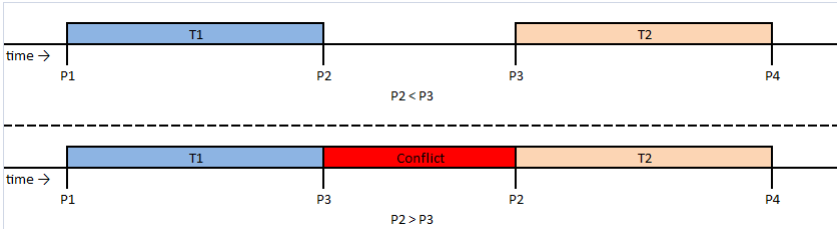


Figure 3.2: Time overlap between two tasks

$$1\{taskDay(T, Dx) : day(Dx) : Dx <= D\}1 \text{ :- } taskDeadline(T, D). \tag{3.24}$$

If a task is given a deadline, it will be performed on a day that is either the day of the deadline, or a previous day. Every task has either a predetermined *taskDay* fact or a *taskDeadline* fact, and this rule assigns *taskDay* facts to tasks with deadlines, so that we can be sure in the end that every task has an established day in which it is performed.

$$1\{busy(E, T) : equipment(E)\}1 \text{ :- } taskType(T, "exercise"). \tag{3.25}$$

Each exercise task uses an equipment, that will be busy while the task is being performed.

With the previous rules all the answer predicates are generated. The next step is to write further constraints in order to obtain a valid plan.

$$\begin{aligned} \text{:- } execute(C, T1), execute(C, T2), T1 \mathrel{!=} T2, taskDay(T1, D), taskDay(T2, D), \\ taskTime(T1, P1, P2), taskTime(T2, P3, P4), P1 <= P4, P3 <= P2. \end{aligned} \tag{3.26}$$

Rule (3.26) guarantees that there is no time overlap among tasks performed by the same crew member (each crew member performs a single task at a time). Figure 3.2 shows how rule (3.26) works: assuming a fixed point where P1 < P4, if P2 < P3 then there is no overlap between the two tasks. However, if P2 > P3, then the tasks overlap.

$$\begin{aligned} \text{:- } isRPCM(T1, R), isRPCM(T2, R), stageRPCM(T1, S1), stageRPCM(T2, S2), S2 > S1, \\ taskDay(T1, D), taskDay(T2, D), taskTime(T1, P1, P2), taskTime(T2, P3, P4), P2 >= P3. \end{aligned} \tag{3.27}$$

$$\begin{aligned} \text{:- } isRPCM(T1, R), isRPCM(T2, R), stageRPCM(T1, S1), stageRPCM(T2, S2), \\ S2 > S1, taskDay(T1, D1), taskDay(T2, D2), D1 > D2. \end{aligned} \tag{3.28}$$

The rules above establish the order relation among tasks of the same RPCM sequence. Rule (3.27) takes two tasks *T1* and *T2* of a RPCM sequence *R* that are scheduled for the same day *D*. *T2* belongs

Table 3.1: ASP result of example instance (Standard Model: Basic Encoding)

| Name | Time | Day | Crew Member |
|---|---|---|---|
| taskName(1,"post-sleep") | taskTime(1,1,13) | taskDay(1,1) | execute(c1,1) |
| taskName(4,"meal") | taskTime(4,14,17) | taskDay(4,1) | execute(c1,4) |
| taskName(3,"exercise") | taskTime(3,20,23) | taskDay(3,1) | execute(c1,3) |
| taskName(10,"payload-pa1_1") | taskTime(10,24,27) | taskDay(5,1) | execute(c1,5) |
| taskName(5,"reconfigurate-thermal-loops-1-rpcm1") | taskTime(5,28,31) | taskDay(6,1) | execute(c1,6) |
| taskName(6,"remove-sleep-station-rpcm1") | taskTime(6,32,35) | taskDay(7,1) | execute(c1,7) |
| taskName(7,"replace-rpcm-rpcm1") | taskTime(7,36,47) | taskDay(8,1) | execute(c1,8) |
| taskName(8,"reconfigurate-thermal-loops-2-rpcm1") | taskTime(8,48,51) | taskDay(9,1) | execute(c1,9) |
| taskName(9,"assemble-sleep-station-rpcm1") | taskTime(9,52,55) | taskDay(10,1) | execute(c1,10) |
| taskName(2,"sleep") | taskTime(2,57,96) | taskDay(2,1) | execute(c1,2) |

to a later stage in the sequence than *T1* and therefore *T2* cannot start before *T1* ends. Rule (3.28) is a similar rule for two tasks of the sequence being performed in different days. The task in a later stage cannot be performed before the task in an earlier stage.

$$\begin{aligned} :\text{-}\ busy(E,T1), busy(E,T2), T1\ != T2, taskDay(T1,D), taskDay(T2,D), \\ taskTime(T1,P1,P2), taskTime(T2,P3,P4), P3 <= P2, P1 <= P4. \end{aligned} \tag{3.29}$$

This rule is similar to rule (3.26), but is used only for exercise tasks: if two exercise tasks *T1* and *T2* use the same exercise equipment $E$, then they cannot overlap in time.

**Solution Example**

By solving the instance example presented in section 3.3 – Instance Encoding Example using the rules from section 3.4.1 – Rules, we get a solution with the predicates shown in Table 3.1, represented also in graphical form in Figure 3.3.

Each line of the table represents one task of the plan. For each task is displayed information about the day and time of execution, and which crew member performs that task. The tasks are ordered by their time of execution. It is noteworthy that the predicates *taskDay* and *execute* shown in the table are all similar to each other, with the only change being the task code. This happens because the plan has the length of one day and only one crew member available. Hence, every task is performed on day 1 by crew member *c1*.

The post-sleep and sleep tasks are performed at the beginning and at the end of the day, respectively, respecting the plan constraints. The tasks 5 to 9, which form a RPCM sequence, are executed in the correct order. In the *Time* column, the periods in the *taskTime* predicates do not intersect each other. This means the tasks never overlap, respecting the restriction of each crew member performing only one task at a time.

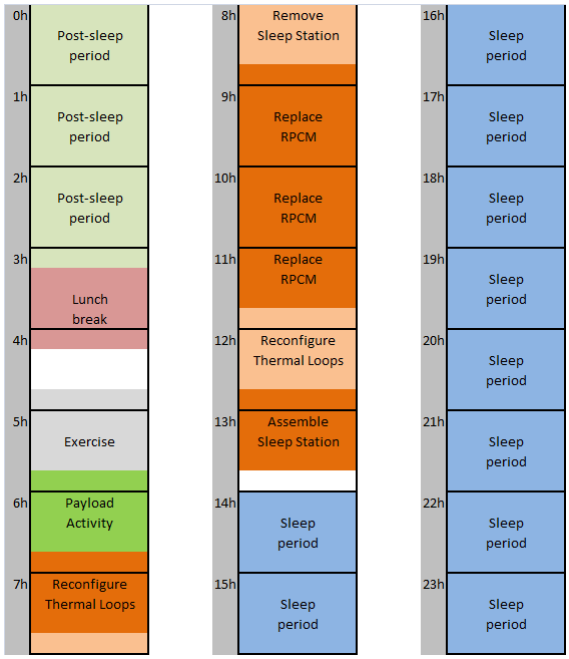


Figure 3.3: Example of a solved plan

### 3.4.2 Optimizations

With the basic encoding, all the problem's constraints are successfully represented. However, its performance drops for big instances. In many cases, the solver is unable to find a plan before running out of memory. We realized that the size of the grounding files was too big, to the point that the solver would run out of memory just by reading the ground file.

It is possible to generate a readable ground file and use it to analyse the ground rules with gringo options *-t* and *--verbose*. In this analysis it became clear that rule (3.26) was generating too many ground rules.

The following is an excerpt from the ground program for the instance presented in section 3.3 – Instance Encoding Example, with the result of applying rule (3.26) to tasks 3 and 4, starting in period 1 and ending in period 4.

```
:-taskTime(4,1,4),taskTime(3,1,4).
:-taskTime(4,1,4),taskTime(3,2,5).
:-taskTime(4,1,4),taskTime(3,3,6).
:-taskTime(4,1,4),taskTime(3,4,7).

:-taskTime(3,1,4),taskTime(4,1,4).
:-taskTime(3,1,4),taskTime(4,2,5).
:-taskTime(3,1,4),taskTime(4,3,6).
:-taskTime(3,1,4),taskTime(4,4,7).
```

The first and fifth line of this excerpt represent exactly the same constraint. In fact, every one of this

constraints is repeated somewhere in the ground file in its symmetrical form. This problem is addressed by Gebser et. al. in [12] and can be fixed by establishing an order among the tasks. The easiest way to do this is to use the numerical code for each task - fixing that T1 < T2 stops the grounder from creating rules to compare both pairs (T1,T2) and (T2,T1), thus breaking the symmetry.

$$\begin{aligned} :\text{-}\ execute(C,T1), execute(C,T2), T1 < T2, taskDay(T1,D), taskDay(T2,D), \\ taskTime(T1,P1,P2), taskTime(T2,P3,P4), P1 <= P4, P3 <= P2. \end{aligned} \tag{3.30}$$

The new overlap rule reduced the size of ground files in about 50%. We also applied the same technique to rule (3.29). Afterall, we improved performance in small and medium instances (some of which we could not solve before). However, it still was not enough to solve big instances.

There is another problem with the ground excerpt - all the rules present in there are unnecessary. As it was shown in rule (3.9), there is already a task scheduled for periods 1 to 13, and this fact holds no matter how many crew members are available to perform tasks, because the scheduled task is the post-sleep task, present in the plan of every crew member. In order to solve this problem, we limited rule (3.23) so that it cannot generate *tasktime* facts that coincide with periods of time already occupied. Also, there is no need to generate facts for tasks which are already scheduled. The new rule is as follows:

$$\begin{aligned} 1\{taskTime(Tid,P,P+K-1) : period(P) : period(P+K-1) : P > 13 : P+K-1 < 57\}1 \\ :\text{-}\ task(Tid), taskType(Tid,T), T\ != \text{"}post\text{-}sleep\text{"}, T\ != \text{"}sleep\text{"}, duration(T,K). \end{aligned} \tag{3.31}$$

The change in this rule trims the ground file to one third of its size, which combined with rule (3.30) results in a ground file six times smaller than the original file.

Lastly, a small optimization was made to the instances, consisting of replacing facts that express task deadlines for day 1 by facts that imply that the task is indeed performed at day 1. This helps the solver by skipping the generation rule for those tasks. However, the gains from this optimization are not very significant.

## 3.5 Standard model: Encoding based on choice rules

Even after applying several optimizations to the encoding presented in section 3.4 – Standard model: Basic encoding, its performance was not as good as desired. The ground files for big instances were still too big for the solver to handle, and even though we optimized the overlap rule to (3.30), it was still responsible for most of the grounding space, generating a big amount of constraints. A new approach was needed in order to solve this problem, which resulted in a new version of the program.

In this section is presented an alternative encoding of the crew planning, which relies heavily on choice

rules. We start by listing the changes in the list of predicates used in the encoding, and its implications in the example instance. Afterwards, the rules of the program are presented, followed by the presentation of the resulting plan for the example instance.

For this encoding to work, we were required to step in the opposite direction regarding information organization, and created one single predicate which contains all information about the scheduling of a task, with the day, time, and crew member that performs it. This brings redundant information to some of the constraints, but allows to rewrite others so that they work now more efficiently.

### 3.5.1 Changes to Predicates and Instances

In this encoding, instead of a *taskTime(Tid,P1,P2)* predicate, we use the *taskPerform(Tid,C,P1,P2,D)* predicate, which consolidates most information about the execution of a task: designated crew member, starting period, ending period and day of execution. Predicates *execute(C,Tid)* and *taskDay(Tid,Day)* are no longer used as answer predicates. However, they may appear in the instance giving information about tasks which have a fixed crew member assigned to them or a fixed day in which they must be performed.

The instances for this encoding are similar to the ones described in section 3.3 – Instance Encoding Example. The only difference is that, when all the information about a task is available, it is now represented with a *taskPerform* predicate. For example, for task 1 of the example instance, we had:

$$taskDay(1, 1). \tag{3.32}$$

$$execute(c1, 1). \tag{3.33}$$

$$taskTime(1, 1, 13). \tag{3.34}$$

We now represent this information as:

$$taskPerform(1, c1, 1, 13, 1). \tag{3.35}$$

### 3.5.2 Rules

Similarly to section 3.4.1 – Rules, this encoding follows the generate-and-test approach, but it also takes into account the optimizations proposed at section 3.4.2 – Optimizations and further knowledge about the performance of the basic encoding.

$$period(1..96). \tag{3.36}$$

Like in section 3.4 – Standard model: Basic encoding, a day is composed of 96 periods.

$$
\begin{aligned}
&1\{taskPerform(Tid, C, P, P+K-1, D) : crewMember(C) : period(P) \\
&: period(P+K-1) : P > 13 : P+K-1 < 57\}1 \text{ :- } task(Tid), taskType(Tid, T), \\
&duration(T, K), taskDay(Tid, D), 0\{execute(Ck, Tid) : crewMember(Ck)\}0.
\end{aligned} \tag{3.37}
$$

$$
\begin{aligned}
&1\{taskPerform(Tid, C, P, P+K-1, D) : period(P) : period(P+K-1) \\
&: P > 13 : P+K-1 < 57\}1 \text{ :- } task(Tid), taskType(Tid, T), \\
&duration(T, K), taskDay(Tid, D), execute(C, Tid).
\end{aligned} \tag{3.38}
$$

$$
\begin{aligned}
&1\{taskPerform(Tid, C, P, P+K-1, Dx) : crewMember(C) : period(P) \\
&: period(P+K-1) : day(Dx) : Dx <= D : P > 13 : P+K-1 < 57\}1 \\
&\text{:- } task(Tid), taskType(Tid, T), duration(T, K), taskDeadline(Tid, D).
\end{aligned} \tag{3.39}
$$

We designed three different rules to generate *taskPerform* predicates, which are used depending on the information available about each predicate. Rule (3.37) is used for tasks where we know for which day they are scheduled, but not which crew member is assigned to perform them. The rule assigns then a random crew member to the task, along with the time of execution. Rule (3.38) is used when both the day and crew member respective to a task are known, so the rule takes those informations and assigns only the time period in which the task is performed. Rule (3.39) is used for tasks which have a defined deadline, and the rule assigns a day accordingly, along with information about the time periods and crew members assigned to the task. For each task in the instance, only one of this rules can be used.

$$
1\{busy(E, T) : equipment(E)\}1 \text{ :- } taskType(T, \text{"}exercise\text{"}). \tag{3.40}
$$

The rule that generates *busy* predicates remains unchanged.

$$
\begin{aligned}
&\text{:- } crewMember(C), period(P), P > 13, P < 57, day(D), \\
&2\{taskPerform(T, C, P1, P2, D) : task(T) : P1 <= P : P2 >= P\}.
\end{aligned} \tag{3.41}
$$

Rule (3.41) is the most important rule in this encoding - in the way that it was designed in order to make writing this rule possible. The concept is that, instead of comparing each task with one another at every possible period, like in rule (3.26), we now have a single rule that defines what happens in each moment in time, for a given crew member. We fix a crew member $C$, a period $P$ and a day $D$. We know that crew member $C$ can only perform one task during period $P$ at day $D$, so we write a constraint

stating it is impossible for him to perform two or more tasks at the same period in time.

Also because of this constraint, the *taskPerform* predicate was created. Otherwise, it would be impossible to write this constraint while expressing each fact separately. In the basic encoding, a rule similar to rule (3.41) would look like the following:

```
:- crewMember(C), period(P), P > 13, P < 57, day(D),
2 { taskTime(T,P1,P2) : task(T) : P1<=P : P2>=P : execute(C,T) : taskDay(T,D)}.
```

Such a rule is impossible to write, because *execute* and *taskDay* could not be at the right side of a conditional expression, as they were not domain predicates. Instead, they were generated by the solver in rules (3.22) and (3.24). To solve this problem, a new predicate *taskPerform* was written, so that the information of *execute* and *taskDay* is now on the left side of the conditional expression, thus making it possible to write this rule.

$$\begin{aligned} \text{:- } & isRPCM(T1, R), stageRPCM(T1, S1), taskPerform(T1, \_, \_, P2, D), \\ & 1\{taskPerform(T2, \_, P3, \_, D) : isRPCM(T2, R) : T1 \mathrel{!=} T2 \\ & : stageRPCM(T2, S2) : S2 > S1 : P2 >= P3\}. \end{aligned} \tag{3.42}$$

$$\begin{aligned} \text{:- } & isRPCM(T1, R), stageRPCM(T1, S1), taskPerform(T1, \_, \_, \_, D1), \\ & 1\{taskPerform(T2, \_, \_, \_, D2) : isRPCM(T2, R) : T1 \mathrel{!=} T2 \\ & : stageRPCM(T2, S2) : S2 > S1 : D1 > D2\}. \end{aligned} \tag{3.43}$$

Rules (3.42) and (3.43) establish the constraints about RPCM tasks. Rule (3.42) takes one RPCM task, which ends at period *P2*, and guarantees it is impossible for another task, of the same RPCM sequence but in a later stage, to start a period *P3* which does not take place later than *P2*. Rule (3.43) applies the same reasoning, but regarding the day in which each task is performed, making it impossible to schedule a task in a later stage to an earlier day. Notice that, in both rules, some variables are replaced with an underscore. This means they are not relevant to the logic of the rule, and thus can be filled with any possible values.

$$\begin{aligned} \text{:- } & busy(E, T1), busy(E, T2), T1 < T2, taskPerform(T1, C1, P1, P2, D), \\ & taskPerform(T2, C2, P3, P4, D), C1 \mathrel{!=} C2, P3 <= P2, P1 <= P4. \end{aligned} \tag{3.44}$$

This rule is essentially rule (3.29) with *taskPerform* predicates. Unlike other constraints, rule (3.44) was not rewritten into a *choice rule* format. This happens because, similarly to what was explained in rule (3.41), the predicate *busy* cannot appear at the right side of a conditional expression. We did not

Table 3.2: ASP result of example instance (Standard Model: Choice-Rules Encoding)

| Name | Performance |
|---|---|
| taskName(1,"post-sleep") | taskPerform(1,c1,1,13,1) |
| taskName(10,"payload-pa1_1") | taskPerform(10,c1,14,17,1) |
| taskName(6,"remove-sleep-station-rpcm1") | taskPerform(6,c1,21,24,1) |
| taskName(4,"meal") | taskPerform(4,c1,25,28,1) |
| taskName(5,"reconfigurate-thermal-loops-1-rpcm1") | taskPerform(5,c1,29,32,1) |
| taskName(7,"replace-rpcm-rpcm1") | taskPerform(7,c1,33,44,1) |
| taskName(9,"assemble-sleep-station-rpcm1") | taskPerform(9,c1,45,48,1) |
| taskName(8,"reconfigurate-thermal-loops-2-rpcm1") | taskPerform(8,c1,49,52,1) |
| taskName(3,"exercise") | taskPerform(3,c1,53,56,1) |
| taskName(2,"sleep") | taskPerform(2,c1,57,96,1) |

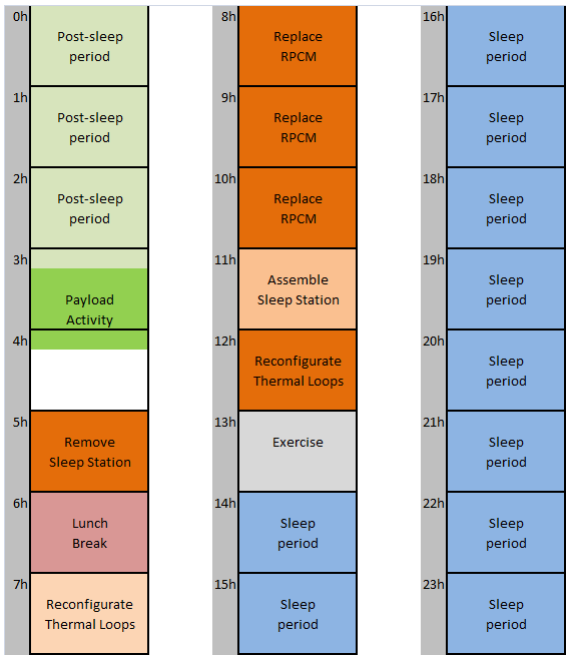


Figure 3.4: Example of a solved plan

merge the *busy* predicate with *taskPerform* because it only relates to the exercise tasks. In addition, the gain in performance from having rule (3.44) replaced by a choice-rule-based restriction is not enough to overcome the loss in performance that would derive from having a bigger predicate in every other rule. On its own, this rule is less efficient, but because there are few exercise tasks, this does not significantly hinder the program's performance and is overall a better choice.

### Solution Example

In Table 3.2 and Figure 3.4 is shown the solution given by this encoding to a similar instance to section 3.3 – Instance Encoding Example. Although using the same instance, the end result is different from the one presented in 3.4.1 – Solution Example. There are many different possible solutions to the same instance. Which one is obtained depends on the internal working of the solver.

In the *Performance* column, we can see that one *taskPerform* predicate was generated for each task. It has information about the task ID, the crew member that performs the task, the starting and ending period and the day of execution. Line (3.35), which was present in the instance, is carried on to the solution as a representation of the first task of the day. The tasks are once again ordered by their execution time, and the ordering of the RPCM tasks is correct. All the tasks in the instance are performed without overlapping in time.

This encoding solves most of the available instances and is the most successful program developed during this work. Its performance is comparable to state-of-the-art planners.

## 3.6 Competition model: Basic Encoding

In the following sections we will address the competition model and present several encodings that conform to its rules. For each encoding based on the standard model, there is a similar encoding based on the competition model that attempts to deal with the variable day length that characterizes the competition model.

### 3.6.1 Changes to Predicates and Instances

In the competition model, the range of *period* domain is not fixed. Instead, it is calculated based on the tasks present in the instance, using the following algorithm:

**Algorithm 1:** Obtaining Maximum Period

```
1  Total ← 0
2  Crew ← 0
3  foreach crew member c do
4  |  Crew ← Crew + 1 %Count the number of crew members
5  end
6  foreach task t do
7  |  Total ← Total + duration (t) %Sum the duration of all tasks
8  end
9  if Crew > 1 then
10 |  Total ← ⌈Total/Crew + 2⌉ %Divide the time of the tasks by the crew
11 end
12 return Total
```

For the example instance presented in section 3.3 – Instance Encoding Example, the result is as follows:

$$period(1..93). \tag{3.45}$$

Because of the nature of the *period* domain in this model, less information is given about the time at which some tasks are performed. In the standard model, it was always known that *post-sleep* tasks started at period 1 and ended at period 13. But in the competition model, that information is only true

for *post-sleep* tasks in the first day. In other days, the starting and ending times of *post-sleep* tasks are variable, because they depend on the length of each day, which is also variable. In the same way, the *sleep* task no longer has a fixed time.

### 3.6.2 Rules

The rules in this encoding are similar to those of section 3.4 – Standard model: Basic encoding, adapted to conform to the competition model. Known optimizations are applied if possible.

$$1\{execute(C, Tid) : crewMember(C)\}1 \text{ :- } task(Tid). \tag{3.46}$$

$$\begin{aligned} 1\{taskTime(Tid, P, P + K - 1) : period(P) : period(P + K - 1)\}1 \\ \text{:- } task(Tid), taskType(Tid, T), duration(T, K). \end{aligned} \tag{3.47}$$

$$1\{taskDay(T, Dx) : day(Dx) : Dx <= D\}1 \text{ :- } taskDeadline(T, D). \tag{3.48}$$

$$1\{busy(E, T) : equipment(E)\}1 \text{ :- } taskType(T, \text{"}exercise\text{"}). \tag{3.49}$$

Rules (3.46) to (3.49) generate the answer predicates *execute*, *taskTime*, *taskDay* and *busy*, respectively. These rules are the same as rules (3.22) to (3.25) present in the basic encoding for the standard model. Remember that the range of the *period* domain depends on the instance. Because of this, it is not trivial to apply an optimization to rule (3.47) that limits the generation of *taskTime* predicates. Hence, we use a simple rule in the basic encoding.

$$\begin{aligned} \text{:- } execute(C, T1), execute(C, T2), taskTime(T1, P1, P2), \\ taskTime(T2, P3, P4), T1 < T2, P1 <= P4, P3 <= P2. \end{aligned} \tag{3.50}$$

Rule (3.50) ensures that tasks performed by the same person do not overlap. Note that the rule does not include any reference to *taskDay* facts, in opposite to rule (3.30). That happens because tasks in different days now have different *taskTime* facts.

$$\begin{aligned} \text{:- } isRPCM(T1, R), isRPCM(T2, R), stageRPCM(T1, S1), stageRPCM(T2, S2), \\ S2 > S1, taskTime(T1, P1, P2), taskTime(T2, P3, P4), P2 >= P3. \end{aligned} \tag{3.51}$$

Rule (3.51) checks the relative order of tasks in the same RPCM sequence. We can write this restriction using only one rule because each period is unique in time.

$$\text{:- } execute(C,T), taskDay(T,D), taskType(T,"post-sleep"), \\ taskTime(T,P1,P2), P1 < (D-1)*96+1. \tag{3.52}$$

Rule (3.52) represents the constraint described in section 3.1.1, that says a day cannot start until at least 24 hours for each previous day have passed. To encode this restriction, we used the post-sleep task and limited its starting time accordingly. Because it is always the first task of the day in a plan, all the other tasks in that day will follow the constraint. If we replace the day value in the equation $P1 < (D-1)*96+1$ we get:

- D=1 → P1<1 (an answer is not valid if P1 < 1)
- D=2 → P1<97
- D=3 → P1<193
- ...

So, taking for example day 2, the post-sleep task cannot start anywhere between period 1 and 96 - which make up 24 hours.

$$\text{:- } execute(C,T1), execute(C,T2), T1 \mathrel{!=} T2, taskDay(T1,D), \\ taskDay(T2,D), taskType(T1,"post\text{-}sleep"), taskTime(T1,P1,P2), \\ taskTime(T2,P3,P4), P3 < P1. \tag{3.53}$$

$$\text{:- } execute(C,T1), execute(C,T2), T1 \mathrel{!=} T2, taskDay(T1,D), \\ taskDay(T2,D), taskType(T1,"sleep"), taskTime(T1,P1,P2), \\ taskTime(T2,P3,P4), P3 > P1. \tag{3.54}$$

Rule (3.53) ensures that the first task of the day of a crew member's plan is the post-sleep task. We define *T1* as being a post-sleep task and *T2* as being any other task, as long as it is performed on the same day by the same crew member. We then fix the start time of these tasks relative to each other, so that it is impossible for task *T2* to start before task *T1*.

Likewise, rule (3.54) ensures that the sleep task is the last task of the day.

$$\text{:- } busy(E,T1), busy(E,T2), taskTime(T1,P1,P2), \\ taskTime(T2,P3,P4), T1 < T2, P3 <= P2, P1 <= P4. \tag{3.55}$$

Rule (3.55) guarantees that exercise tasks that use the same equipment do not overlap in time.

Table 3.3: ASP result of example instance (Competition Model: Basic Encoding)

| Name | Time | Day | Crew Member |
|---|---|---|---|
| taskName(1,"post-sleep") | taskTime(1,1,13) | taskDay(1,1) | execute(c1,1) |
| taskName(6,"remove-sleep-station-rpcm1") | taskTime(6,14,17) | taskDay(6,1) | execute(c1,6) |
| taskName(5,"reconfigurate-thermal-loops-1-rpcm1") | taskTime(5,18,21) | taskDay(5,1) | execute(c1,5) |
| taskName(4,"meal") | taskTime(4,22,25) | taskDay(4,1) | execute(c1,4) |
| taskName(10,"payload-pa1_1") | taskTime(10,26,29) | taskDay(10,1) | execute(c1,10) |
| taskName(3,"exercise") | taskTime(3,30,33) | taskDay(3,1) | execute(c1,3) |
| taskName(7,"replace-rpcm-rpcm1") | taskTime(7,34,45) | taskDay(7,1) | execute(c1,7) |
| taskName(9,"assemble-sleep-station-rpcm1") | taskTime(9,46,49) | taskDay(9,1) | execute(c1,9) |
| taskName(8,"reconfigurate-thermal-loops-2-rpcm1") | taskTime(8,50,53) | taskDay(8,1) | execute(c1,8) |
| taskName(2,"sleep") | taskTime(2,54,93) | taskDay(2,1) | execute(c1,2) |

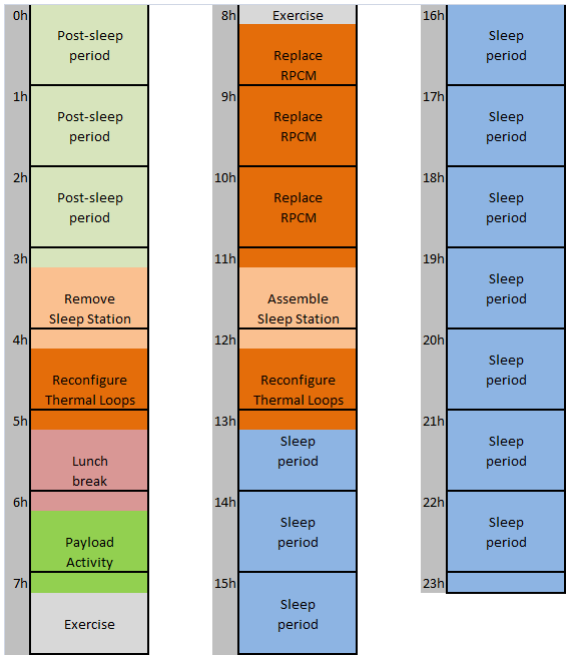


Figure 3.5: Solution of example instance

### Solution Example

Solving the example instance with the encoding just presented results in a solution shown in Table 3.3 and Figure 3.5.

A major difference is visible when looking at the figure: the length of the plan is now shorter than 24 hours, and there are no white spaces between any tasks. This is the result of the length adjustment explained in section 3.6.1 – Changes to Predicates and Instances. Consequently, the sleep task in this solution is arranged to an earlier time than in solutions produced by encodings using the standard model.

Looking at Table 3.3, it does not look much different from Table 3.1 presented in section 3.4.1. The information is presented in the same way, using the predicates *taskTime*, *execute* and *taskDay*. The difference lies in how the solver assigns times to each task. This depends on the region of the search tree

Table 3.4: Minimum starting time of tasks

| Start time | Post-sleep | Other | Sleep |
|---|---|---|---|
| Day 1 | 1 | 14 | TD1-40 |
| Day 2 | 97 | 110 | TD2+57 |
| Day 3 | 193 | 206 | TD3+153 |
| Deadline 2 | N/A | 14 | N/A |
| Deadline 3 | N/A | 14 | N/A |

TD1 - Amount of time for tasks in Day 1
TD2 - Amount of time for tasks in Day 2
TD3 - Amount of time for tasks in Day 3
N/A - Not available (there are no post-sleep and sleep tasks with a deadline)

which is explored first.

### 3.6.3 Optimizations

The basic encoding works on small instances, but has a lot of trouble solving instances for plans that last more than one day. This is a consequence of the changes in the range of the *period* predicate, combined with the problem inherent to the overlap rule, and it results in ground files extremely big. To optimize the ground file and solve this problem, we designed specific generation rules which depend on the position in time of each task of the plan.

To achieve our goal, we added some helpful information to the instances. Each task has now a *knownDay(T,D)* predicate which states whether the execution day of a task $T$ is known in the instance. $D$ takes the value of the scheduled day for the task (the same as in the *taskDay* predicate), and takes the value 0 if the day is unknown (in the case of a deadline task). We also get information about how much time is spent performing tasks associated to each day. This value is given in the predicates *dayOne(TD1)*, *dayTwo(TD2)* and *dayThree(TD3)*. The idea behind these predicates is as follows: if we sum the duration of all tasks scheduled to a specific day, and divide it equally amongst crew members, the result is the minimum amount of time each crew member spends performing tasks associated to a specific day.

We then separate the tasks in two ways:

1. According to time. The test instances provided contain plans of up to three days, so we make five categories: tasks from day 1, day 2, day 3, and tasks with the deadline in days 2 and 3.

2. According to the type of task. Post-sleep and sleep tasks have different positions in time relative to other tasks in the same day, so there are three categories: post-sleep tasks, sleep tasks and other tasks.

For each possible combination of a category from item 1 and a category from item 2, we write a specific generator rule. Each of these rules will have its own range for the *period* domain, which limits the possible values for the *taskTime* predicate of each task. The minimum and maximum values for the range of the *period* domain in each rule are specified in Table 3.4 and Table 3.5.

Table 3.5: Maximum finishing time of tasks

| Finish time | Post-sleep | Other | Sleep |
|---|---|---|---|
| Day 1 | 13 | MP-TD2-TD3-40 | MP-TD2-TD3 |
| Day 2 | MP-TD2-TD3+13 | MP-TD3-40 | MP-TD3 |
| Day 3 | MP-TD3+13 | MP-40 | MP |
| Deadline 2 | N/A | MP-TD3-40 | N/A |
| Deadline 3 | N/A | MP-40 | N/A |

TD1 - Amount of time for tasks in Day 1
TD2 - Amount of time for tasks in Day 2
TD3 - Amount of time for tasks in Day 3
MP - Maximum period
N/A - Not available (there are no post-sleep and sleep tasks with a deadline)

As an example, we will explain the reasoning behind the range for "normal" tasks in day 2. The first 96 periods of the plan are reserved to day 1: according to the rule explained in 3.1.1 – Differences specific to the competition model, tasks in day 2 cannot start until after 24 hours have passed (that is, 96 periods). Then, 13 periods are reserved to the post-sleep task of day 2, which must be performed before any other task in day 2. Thus, normal tasks cannot start until period 110, so that is the minimum value.

For the maximum value, we apply the same reasoning backwards, starting at the last possible period of the plan (the maximum period, referred to as *MP*). The last tasks in the plan will be the tasks assigned to day 3, so we take whatever time they last when distributed among every crew member (the value in the predicate *dayThree*, referred to as *TD3*). Before the day 3 tasks, the crew members perform the sleep task for day 2, which has the duration of 40 periods. And before that, they may perform tasks assigned to day 2. So the maximum period available for normal tasks in day 2 is *MP - TD3 - 40*. The same reasoning is applied to other categories of tasks, which results in Table 3.4 and Table 3.5.

The following are the resulting generator rules for the *taskTime* predicate, which replace rule (3.47):

$$\begin{aligned} &1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1) \\ &: P >= 14 : P+K-1 <= MP - TD2 - TD3 - 40\}1 \\ \text{:- } &task(Tid), taskType(Tid, T), duration(T, K), T \text{ != "}post\text{-}sleep\text{"}, T \text{ != "}sleep\text{"}, \\ &knownDay(Tid, 1), daytwo(TD2), daythree(TD3), maxperiod(MP). \end{aligned} \tag{3.56}$$

$$\begin{aligned} &1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1) \\ &: P >= 110 : P+K-1 <= MP - TD3 - 40\}1 \\ \text{:- } &task(Tid), taskType(Tid, T), duration(T, K), T \text{ != "}post\text{-}sleep\text{"}, \\ &T \text{ != "}sleep\text{"}, knownDay(Tid, 2), daythree(TD3), maxperiod(MP). \end{aligned} \tag{3.57}$$

$$
\begin{aligned}
&1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1) \\
&\qquad : P >= 206 : P+K-1 <= MP-40\}1 \\
&:\text{-}\ task(Tid), taskType(Tid, T), duration(T, K), T\ != \text{"}post\text{-}sleep\text{"}, \\
&\qquad T\ != \text{"}sleep\text{"}, knownDay(Tid, 3), maxperiod(MP).
\end{aligned} \tag{3.58}
$$

Rules (3.56) to (3.58) are generators for normal tasks scheduled to days 1, 2 and 3, respectively. For a task to use a rule, it will need to match with the *knownDay* predicate present in the rule, and respect the constraints related to the type of task (in this case, that the type $T$ is neither "post-sleep" nor "sleep").

$$
\begin{aligned}
&1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1) \\
&\qquad : P >= 97 : P+K-1 <= MP-TD2-TD3+13\}1 \\
&:\text{-}\ task(Tid), taskType(Tid, \text{"}post\text{-}sleep\text{"}), duration(\text{"}post\text{-}sleep\text{"}, K), \\
&knownDay(Tid, 2), daytwo(TD2), daythree(TD3), maxperiod(MP).
\end{aligned} \tag{3.59}
$$

$$
\begin{aligned}
&1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1) \\
&\qquad : P >= 193 : P+K-1 <= MP-TD3+13\}1 \\
&:\text{-}\ task(Tid), taskType(Tid, \text{"}post\text{-}sleep\text{"}), duration(\text{"}post\text{-}sleep\text{"}, K), \\
&knownDay(Tid, 3), daytwo(TD2), daythree(TD3), maxperiod(MP).
\end{aligned} \tag{3.60}
$$

Rules (3.59) and (3.60) generate *taskTime* predicates for post-sleep tasks of days 2 and 3. Post-sleep tasks in day 1 already have a *taskTime* predicate assigned to them in the problem instance, so they do not need a generator rule.

$$
\begin{aligned}
&1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1) \\
&\qquad : P >= TD1-40 : P+K-1 <= MP-TD2-TD3\}1 \\
&:\text{-}\ task(Tid), taskType(Tid, \text{"}sleep\text{"}), duration(\text{"}sleep\text{"}, K), knownDay(Tid, 1), \\
&dayone(TD1), daytwo(TD2), daythree(TD3), maxperiod(MP).
\end{aligned} \tag{3.61}
$$

$$
\begin{aligned}
&1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1) \\
&\qquad : P >= TD2+57 : P+K-1 <= MP-TD3\}1 \\
&:\text{-}\ task(Tid), taskType(Tid, \text{"}sleep\text{"}), duration(\text{"}sleep\text{"}, K), \\
&knownDay(Tid, 2), daytwo(TD2), daythree(TD3), maxperiod(MP).
\end{aligned} \tag{3.62}
$$

$$
\begin{aligned}
1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1) \\
: P >= TD3 + 153 : P+K-1 <= MP\}1 \\
\text{:-}\ task(Tid), taskType(Tid, "sleep"), duration("sleep", K), \\
knownDay(Tid, 3), daythree(TD3), maxperiod(MP).
\end{aligned}
\tag{3.63}
$$

Rules (3.61) to (3.63) generate *taskTime* predicates for sleep tasks.

$$
\begin{aligned}
1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1) : P >= 14 \\
: P+K-1 <= MP - TD3 - 40\}1 \text{ :- } task(Tid), taskType(Tid, T), duration(T, K), \\
knownDay(Tid, 0), taskDeadline(Tid, 2), daythree(TD3), maxperiod(MP).
\end{aligned}
\tag{3.64}
$$

$$
\begin{aligned}
1\{taskTime(Tid, P, P+K-1) : period(P) : period(P+K-1) : P >= 14 \\
: P+K-1 <= MP - 40\}1 \text{ :- } task(Tid), taskType(Tid, T), duration(T, K), \\
knownDay(Tid, 0), taskDeadline(Tid, 3), maxperiod(MP).
\end{aligned}
\tag{3.65}
$$

Rules (3.64) and (3.65) generate *taskTime* predicates for tasks with deadlines. The limits for deadline tasks are actually a combination of rules (3.56) to (3.58): the minimum limit is common to (3.56), and the maximum limit is the same in each rule as (3.57) and (3.58), respectively.

The new generation rules greatly improve the grounding process, with much smaller ground files. However, that is not translated into more effectiveness when solving problems. The program is still unable to tackle medium and large instances. In chapter 4 – Experimental Evaluation we will show that the lack of success in this case is related to the solver instead of the grounder.

## 3.7 Competition model: Encoding based on choice rules

In this section we will present an encoding based on choice rules, adapted for the competition model. This encoding combines the changes to the basic model described in sections 3.5.1 and 3.6.1. That means it uses the *taskPerform* predicate and a variable *period* domain. The development of this encoding is relevant because many of the constraints of the competition model can be expressed as choice rules, and a similar encoding for the standard model was successful.

### 3.7.1 Rules

$$
\begin{aligned}
1\{taskPerform(Tid, C, P, P+K-1, D) : crewMember(C) \\
: period(P) : period(P+K-1) : P > 96 * (D-1)\}1 \\
\text{:-}\ task(Tid), taskType(Tid, T), duration(T, K), taskDay(Tid, D).
\end{aligned}
\tag{3.66}
$$

$$
\begin{aligned}
1\{taskPerform(Tid, C, P, P+K-1, Dx) : crewMember(C) : period(P) \\
: period(P+K-1) : day(Dx) : Dx <= D : P > 96 * (Dx-1)\}1 \\
\text{:- } task(Tid), taskType(Tid, T), duration(T, K), taskDeadline(Tid, D).
\end{aligned} \tag{3.67}
$$

As explained in section 3.6.3, there are several ways to design the generator rules. We chose to present a version in which we have two rules, (3.66) and (3.67), where one generates *taskPerform* predicates for tasks with a fixed day and the other does it for tasks with a deadline. By using the constraint $P > 96 * (D-1)$, we limit the domain of the *period* predicate. That way, the program cannot assign a task scheduled for day $D$ to a period in which day $D$ cannot have started yet. It is possible to design different rules for each particular case, but ultimately their impact in the program's performance was not relevant enough.

$$
1\{busy(E, T) : equipment(E)\}1 \text{ :- } taskType(T, \text{"}exercise\text{"}). \tag{3.68}
$$

Rule (3.68) generates the *busy* predicate and is identical to rules with the same role in other encodings.

$$
\begin{aligned}
\text{:- } crewMember(C), period(P), \\
2\{taskPerform(T, C, P1, P2, \_) : task(T) : P1 <= P : P2 >= P\}.
\end{aligned} \tag{3.69}
$$

Rule (3.69) eliminates the possibility of overlapping tasks in the schedule of a crew member. This rule is similar to rule (3.41), but with a less restricted domain. It now generates one restriction for each possible combination of period $P$ and crew member $C$.

$$
\text{:- } execute(C1, T), taskPerform(T, C2, \_, \_, \_), C1 \mathrel{!=} C2. \tag{3.70}
$$

Rule (3.70) guarantees that tasks which are assigned to a crew member are performed by that same crew member.

$$
\begin{aligned}
\text{:- } isRPCM(T1, R), stageRPCM(T1, S1), taskPerform(T1, \_, \_, P2, \_), \\
1\{taskPerform(T2, \_, P3, \_, \_) : isRPCM(T2, R) : T1 \mathrel{!=} T2 \\
: stageRPCM(T2, S2) : S2 > S1 : P2 >= P3\}.
\end{aligned} \tag{3.71}
$$

Rule (3.71) establishes an order among RPCM tasks of the same sequence. It is similar to rule (3.42), but without the day restriction, so that it encompasses all possibilities in one single rule (as it was done in rule (3.51)).

Table 3.6: ASP result of example instance (Competition Model: Choice-Rules Encoding)

| Name | Performance |
|---|---|
| taskName(1,"post-sleep") | taskPerform(1,c1,1,13,1) |
| taskName(6,"remove-sleep-station-rpcm1") | taskPerform(6,c1,14,17,1) |
| taskName(5,"reconfigurate-thermal-loops-1-rpcm1") | taskPerform(5,c1,18,21,1) |
| taskName(4,"meal") | taskPerform(4,c1,22,25,1) |
| taskName(3,"exercise") | taskPerform(3,c1,26,29,1) |
| taskName(10,"payload-pa1_1") | taskPerform(10,c1,30,33,1) |
| taskName(7,"replace-rpcm-rpcm1") | taskPerform(7,c1,34,45,1) |
| taskName(8,"reconfigurate-thermal-loops-2-rpcm1") | taskPerform(8,c1,46,49,1) |
| taskName(9,"assemble-sleep-station-rpcm1") | taskPerform(9,c1,50,53,1) |
| taskName(2,"sleep") | taskPerform(2,c1,57,96,1) |

$$\begin{aligned} &:\text{-}\ taskType(T1, "post-sleep"), taskPerform(T1, C, P1, \_, D), \\ &1\{taskPerform(T2, C, P3, \_, D) : T1 \mathrel{!=} T2 : P3 < P1\}. \end{aligned} \tag{3.72}$$

$$\begin{aligned} &:\text{-}\ taskType(T1, "sleep"), taskPerform(T1, C, P1, \_, D), \\ &1\{taskPerform(T2, C, P3, \_, D) : T1 \mathrel{!=} T2 : P3 > P1\}. \end{aligned} \tag{3.73}$$

Rules (3.72) and (3.73) guarantee that the post-sleep and sleep tasks are the first task and last task of each day, respectively.

$$\begin{aligned} &:\text{-}\ busy(E, T1), busy(E, T2), T1 < T2, taskPerform(T1, C1, P1, P2, \_), \\ &taskPerform(T2, C2, P3, P4, \_), C1 \mathrel{!=} C2, P3 <= P2, P1 <= P4. \end{aligned} \tag{3.74}$$

Rule (3.74) prevents exercise tasks performed in the same equipment from overlapping. It is similar to rule (3.44), but without the constraint of both tasks happening on the same day. That is due to the possibility of two crew members being on a different shift, so they may perform exercise tasks scheduled to different days at the same time.

**Solution Example**

The result of this encoding for the example instance is shown in Table 3.6 and Figure 3.6. Looking at the figure, the obtained solution is visibly similar to the solution presented in section 3.6.2 (other than the different order of some tasks), despite using different internal representations, as Table 3.6 and Table 3.3 show.

The performance of this encoding is somewhat disappointing. Even though it performs better than every other encoding for the competition model, it is beaten by every encoding for the standard model. We

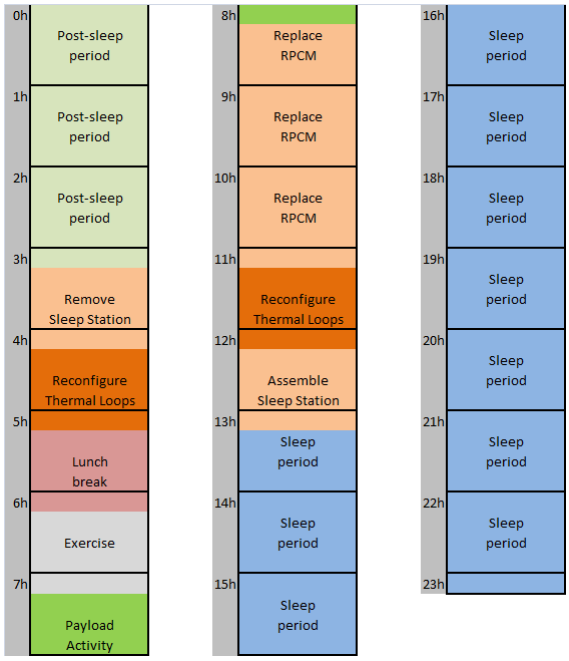


Figure 3.6: Solution of example instance

can conclude that the standard representation is better suited to be modelled in ASP than the competition representation. We will develop this opinion further when discussing the experimental results.

## 3.8 Other approaches

In this section we will mention two other encodings which were attempted to solve the problem, but that were unsuccessful. We will describe the reasoning used in each encoding and explain why they failed.

### 3.8.1 Order-based encoding

An order-based encoding consists in solving the problem by establishing an order relationship among the tasks, instead of defining a time. One possibility is to have a *taskOrder(T,C,N)* predicate, which states that the task $T$ performed by the crew member $C$ is the $N$th task performed by that crew member. The plan is then obtained by gathering all the *taskOrder* predicates and ordering them according to $N$.

Such an encoding eliminates the overlap problem, simply because it is not possible to overlap tasks when the calculations are made using an order number. There are as many order positions as the number of tasks performed by a crew member, and that number is normally much smaller than the number of periods, used in other encodings. Because of this, any necessary checks are made easier, by using a smaller domain.

However, there is a fundamental problem with this approach: the possible existence of several crew members performing tasks simultaneously. With more than one crew member, there are several "order chains" (one for each crew member), and the plan does not establish a temporal relation between tasks performed by different crew members. This is a problem because we need to know whether tasks

performed by different crew members overlap in two situations: when handling RPCM sequences and exercise tasks.

The different tasks in a RPCM sequence can be performed by different people, and need to respect an order relationship amongst themselves. If two of those tasks are performed by different people, then it is impossible to know which one actually is being performed first, because the duration of each task is different. The only way of doing so is to calculate how much time has passed until the task in question, that is, we need to know exactly in which period in time a RPCM task is being performed. We can obtain the time period of a task in two ways:

- By generating the time upfront (which is what we do in the encodings we presented before)
- By computing the time recursively, that is, for each task the time is based on the time of the task before it.

In practice, this means we sum the duration of our task to the time of the task before, and generating predicates using sums is extremely inefficient in ASP. It would mean checking every possible ordering of the tasks, and for each one calculate what the resulting sums would be. Doing the calculations this way is not only much less efficient, but it also renders the basic information about the task order useless after the calculations. Note that it is not required to force any constraints of the problem (they can all be done using only the time period information). The result would be a program with extremely big ground files, full of redundant information and outclassed in every way by all other encodings.

### 3.8.2 Iterative encoding

In section 2.3.4 – iClingo we presented the iClingo tool as a way to optimize problem solving when dealing with variable boundaries. This can be applied to the plan creation, by making the range of the period domain a variable over which we can iterate. It guarantees that the solution has an optimal length.

The main problem with this idea is that, in order to conclude that it is impossible to generate a valid plan on an iteration, the solver needs to travel the whole search space, which takes a very long time. The encodings presented work faster because we treat the problem as a satisfaction problem, which means the program stops searching after finding the first solution. By transforming the problem into an optimization problem, the solver suddenly has much more work to do, and consequently takes more time to find a (now optimal) solution.

However, even if we were already working on an optimization problem, an iterative approach would still be inefficient compared to a standard approach. This is because most of the constraints have to be entirely rewritten in order to work properly. Most of the program would be composed of volatile rules, which cannot be used in further iterations, and most of the work done by the solver would be wasted.

# 4 Experimental Evaluation

In this chapter we will evaluate the encodings developed on the course of this work. We start by introducing the instances used in the evaluation process, and then we present the results of several tests involving the developed encodings, manipulation of some variables and comparing to other planners.

The evaluation of this work is intricately linked to the International Planning Competition (IPC). The IPC is a competition held under the International Conference on Planning and Scheduling (ICAPS), that tests state-of-the-art planners and encourages the development of automated planning technology. In the editions of 2008 and 2011, the crew planning problem was used as a test domain. That domain served as a guideline to the work we developed.

Our evaluation adopts the method of the temporal satisfaction track of the IPC. The planners are submitted to the competition, so that the organizers can perform the evaluation of each planner. In the temporal satisfaction track, each planner is given 30 minutes to solve each instance. The instances are spread across several problem domains. The results for each planner are then calculated according to whether they solved each instance. The quality of the solutions obtained is also taken into account.

## 4.1 Problem Instances

In the evaluation of our encodings, we use 30 instances. Each instance can require a plan that lasts from 1 to 3 days, and have 1 to 3 assigned crew members. Table 4.1 presents the characteristics of each instance, according to the number of days and crew members. The 30 instances were obtained from the temporal satisfaction track of the IPC held in 2008[1]. The instances are named from *p01* to *p30*.

### 4.1.1 Instance Conversion

The instances used by the IPC use the PDDL language. We went through a conversion step in order to translate the instances to ASP. In the following page we present an instance of the crew planning problem in the PDDL language. Then we will explain the steps required to transform the PDDL instance into an ASP instance.

[1] The instances can be downloaded in http://ipc.informatik.uni-freiburg.de/Domains

```
(define (problem CrewPlanning_2crew_1day_100utilization)
(:domain CrewPlanning)
(:objects
        d0 d1 d2 - Day
        c1 c2 - CrewMember
        mcs1 mcs2 - MedicalState
        spaceshipFilter - FilterState
        rpcm1 - RPCM
        pa1_1 pa1_2 pa1_3 pa1_4 pa1_5 pa1_6 pa1_7 pa1_8 - PayloadAct
        e1 e2 - ExerEquipment)
(:init
        (currentday c1 d0)
        (done_sleep c1 d0)
        (available c1)
        (currentday c2 d0)
        (done_sleep c2 d0)
        (available c2)
        (initiated d1)
        (next d0 d1)
        (next d1 d2)
        (unused e1)
        (unused e2))
(:goal
(and
        (done_sleep c1 d1)
        (done_sleep c2 d1)
        (initiated d2)
        (mcs_finished mcs1 d1)
        (changed spaceshipFilter d1)
        (done_rpcm rpcm1 d1)
        (payload_act_completed pa1_1 d1)
        (payload_act_completed pa1_2 d1)
        (payload_act_completed pa1_3 d1)
        (payload_act_completed pa1_4 d1)
        (payload_act_completed pa1_5 d1)
        (payload_act_completed pa1_6 d1)
        (payload_act_completed pa1_7 d1)))
(:metric minimize (total-time)))
```

Table 4.1: Instance classification by dimension

| Instance type | 1 day | 2 days | 3 days |
|---|---|---|---|
| 1 crew member | 1 to 4 | 13 to 15 | 22 to 24 |
| 2 crew members | 5 to 8 | 16 to 18 | 25 to 27 |
| 3 crew members | 9 to 12 | 19 to 21 | 28 to 30 |

Line 4 contains information about the number of days of the instance, in a list of string of the form $dx$, in which $x$ is the number of the day. The first and last day of the list are disregarded. For all the others, we write a line $day(x)$ in the ASP instance. For this particular instance, the result is:

```
day(1).
```

Line 5 contains information about the crew members, in a list of strings of the form $cx$. Each string represents a crew member, and we write a line $crewMember(cx)$ for each one. In this case, the result is:

```
crewMember(c1).
crewMember(c2).
```

Also, for each crew member, we generate information about the tasks of the basic plan: post-sleep, sleep, exercise and lunch. In this case, there is one day and two crew members, so 8 tasks are generated. The ID used to identify each task is obtained from a counter. It starts on task 1 and increments by 1 for each task added. The result is that the tasks of an instance have sequential IDs. The exact form of the information generated depends on the encoding, and has been presented before, particularly in section 3.3 – Instance Encoding Example.

Line 10 contains information about the exercise equipments available, in a list of strings of the form $ex$. We write a line $equipment(ex)$ for each one. For this instance, this step results in the lines:

```
equipment(e1).
equipment(e2).
```

Lines 28 to 37 contain information about additional tasks. In this instance, these lines cover all the possible types of tasks. The lines are of the form *(string1 string2 string3)*, which mean the following:

- *string1* indicates the type of task that is being created.
- *string2* may contain additional information about the task.
- *string3* has information about the day of execution of the task, or its deadline, depending on the type of task.

Lines with the string $mcs\_finished$ are transformed into medical conference tasks. It is followed by a string $mcsx$, in which $x$ indicates the crew member $cx$ is assigned to this task. In the case of line 28, the

task is assigned to crew member *c1*. The string *d1* tells us the task must be performed on day 1. The obtained task may look as the following:

```
task(9).
taskName(9,"medical-conference-mcs1").
taskType(9,"medical-conference").
taskDay(9,1).
execute(c1,9).
```

Line 29 creates a filter change task, as indicated by the string *changed*. The only information associated with that type of task is the day of execution. In this case, the task is performed on day 1.

```
task(10).
taskName(10,"change-filter").
taskType(10,"change-filter").
taskDay(10,1).
```

Line 30 tells us that there is a RPCM sequence that should be completed until day 1. This line generates 5 tasks that compose a RPCM sequence. The string *rpcm1* is used as an ID to the RPCM sequence, used in the following predicate (and similar predicates for the other tasks in the sequence):

```
isRPCM(11,rpcm1).
```

Lines 31 to 37 represent payload tasks. For this instance, all the tasks must be performed until the end of day 1. Each line contains a string of the form *pax_n*, in which:

- $x$ is the day of the deadline for this task. It will always contain the same number as the $dx$ string afterwards.
- $n$ tells us this is the $n^{th}$ payload task with a deadline on day $x$.

The string is also used to identify the payload task in its name. As an example, the task from line 33 will be translated to the following lines:

```
task(18).
taskName(18,"payload-pa1_3").
taskType(18,"payload").
taskDay(18,1).
```

### 4.1.2 Instance Modification

The instances we use to test the programs are compatible with the competition model. In their original form, some instances are impossible to solve in the standard model. In order to assure all instances were solvable, we trimmed the unsolvable instances by removing some tasks from them.

The list of tasks removed from the instances is the following:

- Instance p08: one payload task with deadline on day 1.
- Instance p12: two payload tasks with deadline on day 1.
- Instance p18: one payload task with deadline on day 1 and two payload tasks with deadline on day 2.
- Instance p21: two payload tasks with deadline on day 1 and two payload tasks with deadline on day 2.
- Instance p23: one filter change task scheduled to day 2 and one filter change task scheduled to day 3.
- Instance p24: one payload task with deadline on day 2 and one payload task with deadline on day 3.
- Instance p27: one payload task with deadline on day 1, two payload tasks with deadline on day 2 and one payload task with deadline on day 3.
- Instance p30: two payload tasks with deadline on day 1, two payload tasks with deadline on day 2 and two payload tasks with deadline on day 3.

## 4.2 Results

In this section we will present the experimental results collected during the evaluation stage of this work. All results were collected using Intel Xeon 5160 machines (dual-cores with 3.00 GHz of clock speed, 4 MB of cache, 1333 MHz of FSB speed, and 4 GB of RAM each), running 64-bit versions of Linux 2.6.33.3-85.fc13. The tests have an imposed limit of 30 minutes per instance and a memory limit of 3,8 GB. We used the program runsolver[29] to control the evaluation process. The values collected refer to either time (which is always expressed in seconds) or file size (which is always expressed in megabytes), and are rounded to two decimal places. The values are presented in tables and charts. All the charts presented use a logarithmic scale.

We use the following abbreviations in this chapter:

- SMB - Standard Model: Basic
- SMBO - Standard Model: Basic Optimized

- SMCR - Standard Model: Choice Rules
- CMB - Competition Model: Basic
- CMBO - Competition Model: Basic Optimized
- CMCR - Competition Model: Choice Rules
- CMO - Competition Model: Order
- T.O. - The solver ran out of time.
- O.O.M - The solver ran out of memory.
- Grnd T - Grounding time
- Slv T - Solving time
- Total T - Total time (grounding time + solving time)

### 4.2.1 Comparing ASP encodings

We start by comparing the performance of the ASP encodings presented in chapter 3 – Solution Development. In this comparison, we ran gringo and clasp with the default options.

Table 4.2 shows which instances each encoding was able to solve in time, and how much time they took. The time displayed is the sum of the time spent by gringo and clasp. Looking at the table, we can distinguish a progression in performance from each basic encoding to the more optimized versions. The optimized versions solve more instances and generally take less time to solve them. Also, there is a notorious difference between the performance of encodings that follow the standard model and encodings that follow the competition model. The latter have trouble solving any instance with plans of two or more days, and all of them are unable to solve any instance after *p13*.

Table 4.2: Solved instances by time (s)

| | SMB | SMBO | SMCR | CMB | CMBO | CMCR |
|---|---|---|---|---|---|---|
| p01 | 0,71 | 0,16 | 0,09 | 1,36 | 0,21 | 0,32 |
| p02 | 0,3 | 0,08 | 0,05 | 0,55 | 0,33 | 0,16 |
| p03 | 0,71 | 0,15 | 0,08 | 1,36 | 0,2 | 0,32 |
| p04 | 0,73 | 0,16 | 0,08 | 10,33 | 3,46 | 0,29 |
| p05 | 1,62 | 0,31 | 0,16 | 3,53 | 0,25 | 0,72 |
| p06 | 2,79 | 0,53 | 0,2 | 6,39 | 1,35 | 1,07 |
| p07 | 4,75 | 0,92 | 0,25 | 89,48 | 15,87 | 1,63 |
| p08 | 6,2 | 1,22 | 0,32 | 44,55 | 1,47 | 5,23 |
| p09 | 7,24 | 1,32 | 0,38 | 16,31 | 1,07 | 2,78 |
| p10 | 11,14 | 2,11 | 0,45 | 84,47 | 10,71 | 12,14 |
| p11 | 19,55 | 3,47 | 0,7 | - | 3,83 | 6,83 |
| p12 | 54,37 | 115,2 | 0,81 | - | 43,91 | 7,79 |
| p13 | 1,74 | 0,38 | 0,17 | - | - | 908,61 |
| p14 | 3,01 | 0,57 | 0,19 | - | - | - |
| p15 | - | - | 0,2 | - | - | - |
| p16 | 29,61 | 3,52 | 0,39 | - | - | - |
| p17 | - | - | - | - | - | - |
| p18 | - | 1654,83 | 0,73 | - | - | - |
| p19 | 533,74 | 18,93 | 0,81 | - | - | - |
| p20 | - | - | - | - | - | - |
| p21 | - | - | 0,95 | - | - | - |
| p22 | 8,27 | 1,7 | 0,33 | - | - | - |
| p23 | 7,38 | 0,54 | 15,94 | - | - | - |
| p24 | - | 0,92 | 1,79 | - | - | - |
| p25 | - | 16,81 | 1,29 | - | - | - |
| p26 | - | - | - | - | - | - |
| p27 | - | - | - | - | - | - |
| p28 | - | 336,25 | 2,25 | - | - | - |
| p29 | - | - | 1,5 | - | - | - |
| p30 | - | - | - | - | - | - |
| Solved | 18/30 | 22/30 | 25/30 | 10/30 | 12/30 | 13/30 |

Table 4.3: Time spent on grounding (s)

| | SMB | SMBO | SMCR | CMB | CMBO | CMCR |
|---|---|---|---|---|---|---|
| p01 | 0,48 | 0,11 | 0,08 | 0,65 | 0,13 | 0,28 |
| p02 | 0,25 | 0,07 | 0,04 | 0,33 | 0,06 | 0,15 |
| p03 | 0,48 | 0,11 | 0,07 | 0,65 | 0,12 | 0,28 |
| p04 | 0,51 | 0,12 | 0,07 | 0,68 | 0,17 | 0,27 |
| p05 | 1,17 | 0,22 | 0,16 | 1,81 | 0,17 | 0,65 |
| p06 | 1,94 | 0,34 | 0,2 | 2,79 | 0,34 | 0,98 |
| p07 | 3,23 | 0,54 | 0,24 | 4,56 | 0,67 | 1,49 |
| p08 | 3,54 | 0,59 | 0,3 | 5,71 | 0,81 | 2,11 |
| p09 | 4,89 | 0,78 | 0,36 | 7,2 | 0,56 | 2,48 |
| p10 | 7,3 | 1,14 | 0,44 | 10,62 | 1,17 | 3,56 |
| p11 | 10,69 | 1,68 | 0,65 | 16,01 | 1,93 | 5,48 |
| p12 | 12,28 | 1,91 | 0,68 | 21,02 | 2,99 | 6,59 |
| p13 | 1,24 | 0,26 | 0,16 | 5,55 | 1,1 | 1,49 |
| p14 | 1,79 | 0,36 | 0,18 | 7,78 | 1,76 | 2,09 |
| p15 | 1,94 | 0,38 | 0,19 | 9,01 | 1,91 | 2,49 |
| p16 | 10,92 | 1,79 | 0,37 | 51,42 | 8,24 | 9,87 |
| p17 | 15,54 | 2 | 0,59 | 68,86 | 14,02 | 15,69 |
| p18 | 14,75 | 2,4 | 0,45 | 80,03 | 15,96 | 16,45 |
| p19 | 38,3 | 4,88 | 0,77 | 177,92 | 27,41 | 32,64 |
| p20 | 54,07 | 5,81 | 1,33 | 237,92 | 46,71 | 49,47 |
| p21 | 54,67 | 6,91 | 0,9 | 280,36 | 57,47 | 52,5 |
| p22 | 3,9 | 0,71 | 0,32 | 31,23 | 6,52 | 7,06 |
| p23 | 3,88 | 0,35 | 0,46 | 38,43 | 9,5 | 12,03 |
| p24 | 4,36 | 0,51 | 0,42 | 41,94 | 10,85 | 11,96 |
| p25 | 32,68 | 5,23 | 1,24 | 257,8 | 47,69 | 46,42 |
| p26 | 40,96 | 6,47 | 1,34 | 319,08 | 66,87 | 59,94 |
| p27 | 36,05 | 4,44 | 1,67 | 345,95 | 75,88 | 72,9 |
| p28 | 103,61 | 11,77 | 1,76 | 862,01 | 151,34 | 161,64 |
| p29 | 149,64 | 18,84 | 1,25 | 1101,13 | 233,3 | 186,08 |
| p30 | 134,61 | 17 | 1,3 | 1232,09 | 265,38 | 207,74 |

Tables 4.3 and 4.4 contain information about the time spent on grounding and the size of the ground file, respectively, for running each instance on each program. The same information is displayed in Figure 4.1 and Figure 4.2 in chart form. Although the growth in time and file size is exponential, the charts show that, for the most part, the grounder spends more time when creating larger ground files. The charts also show that the performance of the encodings for the competition model is progressively worse on larger instances. This is most visible on the CMBO encoding: from instances *p1* to *p12* its performance is close to the SMBO encoding, but starting on instance *p13*, which is the first instance with a plan of two days, the performance level becomes similar to the SMB encoding. This is explained by the larger *period* domain on instances with plans of multiple days.

Table 4.4: Size of ground file (MB)

| | SMB | SMBO | SMCR | CMB | CMBO | CMCR |
|---|---|---|---|---|---|---|
| p01 | 1,77 | 0,31 | 0,05 | 2,73 | 0,28 | 0,43 |
| p02 | 0,68 | 0,12 | 0,02 | 1,26 | 0,11 | 0,19 |
| p03 | 1,77 | 0,31 | 0,05 | 2,73 | 0,28 | 0,43 |
| p04 | 1,77 | 0,28 | 0,03 | 2,83 | 0,4 | 0,37 |
| p05 | 4,14 | 0,65 | 0,05 | 9,11 | 0,57 | 0,93 |
| p06 | 7,19 | 1,08 | 0,07 | 14,24 | 1,29 | 1,36 |
| p07 | 12,03 | 1,92 | 0,09 | 22,73 | 2,73 | 2,1 |
| p08 | 14,32 | 2,23 | 0,22 | 28,76 | 3,28 | 3,17 |
| p09 | 17,78 | 2,95 | 0,15 | 39,28 | 2,48 | 3,61 |
| p10 | 28,26 | 4,57 | 0,2 | 56,86 | 6,04 | 5,21 |
| p11 | 44,21 | 6,99 | 0,49 | 83,55 | 9,24 | 8,55 |
| p12 | 50,73 | 8,07 | 0,46 | 108,41 | 14,4 | 9,87 |
| p13 | 4,52 | 0,8 | 0,06 | 23,16 | 2,69 | 2,04 |
| p14 | 6,81 | 1,29 | 0,07 | 32,13 | 4,79 | 2,9 |
| p15 | 7,65 | 1,43 | 0,1 | 37,37 | 4,7 | 3,69 |
| p16 | 43,39 | 8,29 | 0,25 | 242,45 | 31,49 | 16,04 |
| p17 | 66,14 | 9,12 | 0,79 | 319,78 | 56,6 | 29,56 |
| p18 | 62,02 | 11,47 | 0,38 | 368,98 | 61,19 | 28,57 |
| p19 | 156,52 | 30,01 | 0,55 | 941,22 | 117,58 | 54,93 |
| p20 | 231,15 | 35,91 | 1,85 | 1250,26 | 203,13 | 94,27 |
| p21 | 227,87 | 43,12 | 0,65 | 1464,63 | 243,74 | 90,33 |
| p22 | 15,24 | 3,04 | 0,13 | 121,8 | 15,86 | 9,63 |
| p23 | 16,79 | 1,01 | 0,53 | 152,67 | 26,96 | 22,1 |
| p24 | 18,26 | 1,83 | 0,41 | 170,34 | 30,93 | 20,15 |
| p25 | 136,35 | 27,17 | 0,48 | 1260,2 | 165,12 | 77,14 |
| p26 | 170,88 | 33,99 | 0,56 | 1522,81 | 243,71 | 97,54 |
| p27 | 154,41 | 22,26 | 1,6 | 1622,09 | 282,95 | 136,03 |
| p28 | 445,26 | 77,92 | 2,53 | 4400,5 | 595,85 | 304,59 |
| p29 | 653,58 | 127,73 | 1,45 | 5544,92 | 963,45 | 340,88 |
| p30 | 581,62 | 113,28 | 1,35 | 6033,05 | 1066,19 | 379,61 |

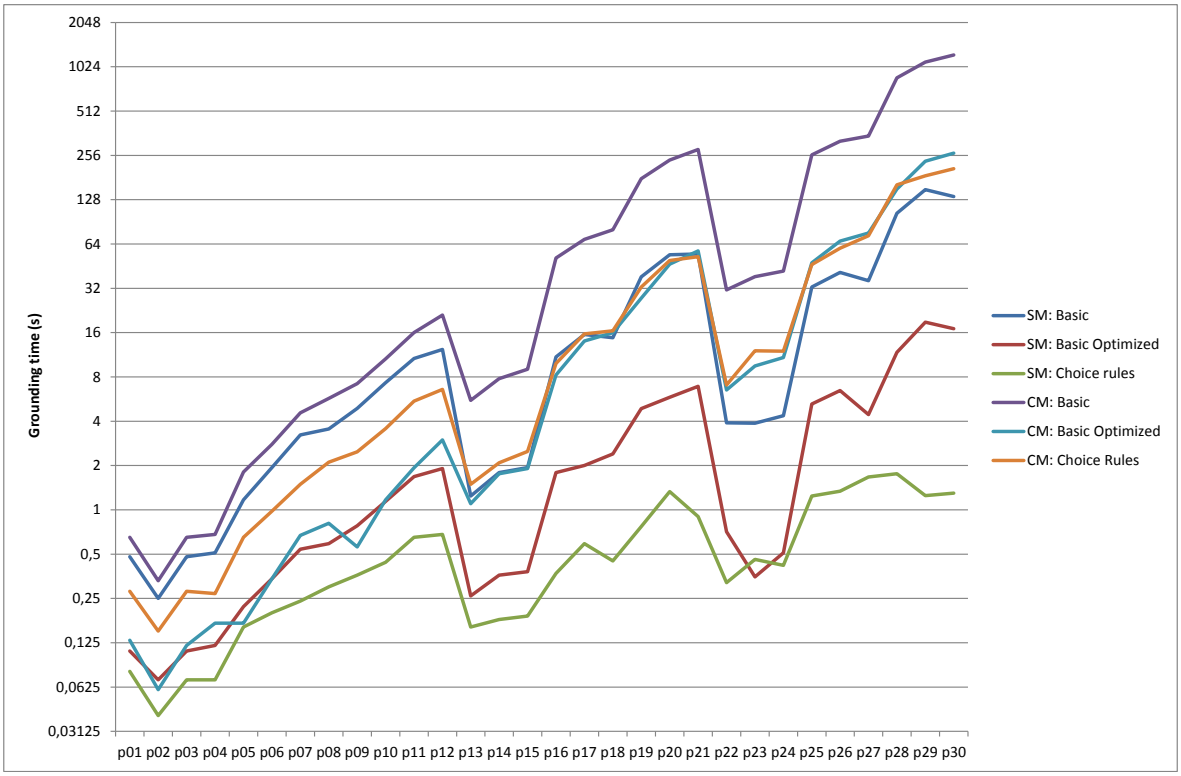


Figure 4.1: Time spent on grounding

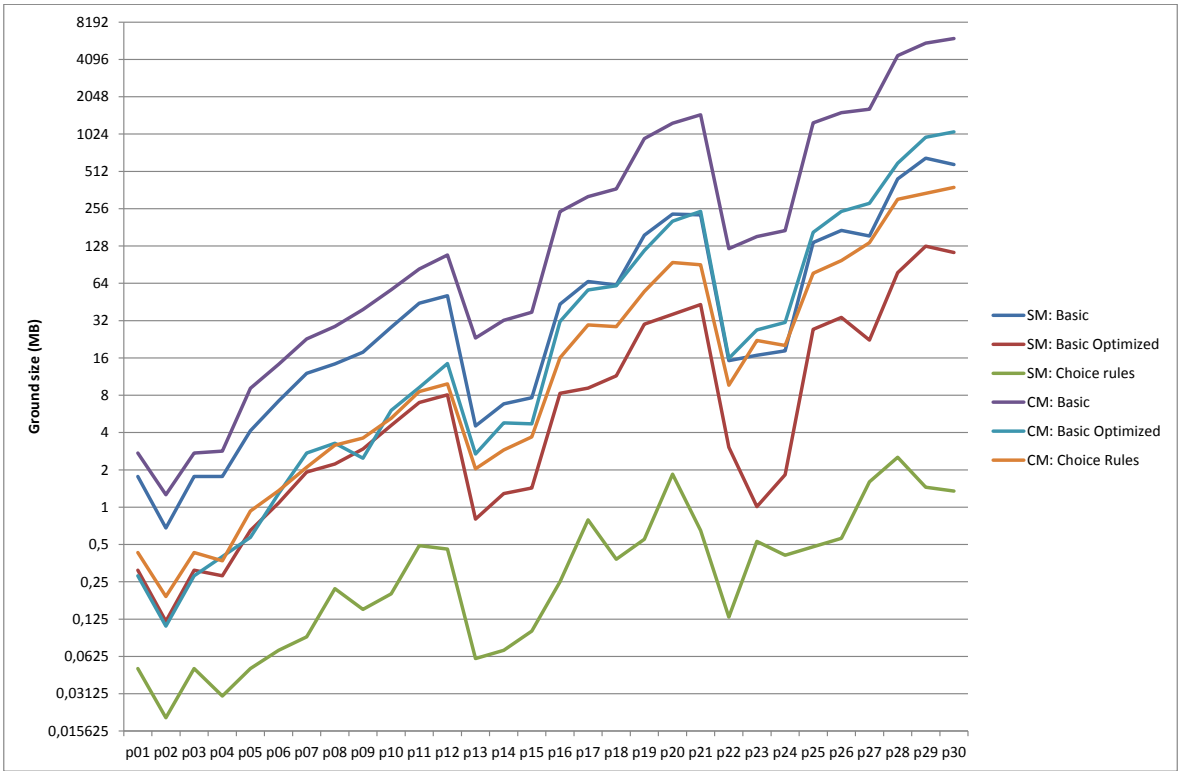


Figure 4.2: Size of ground file

Table 4.5: Time spent on solving (s)

| | SMB | SMBO | SMCR | CMB | CMBO | CMCR |
|---|---|---|---|---|---|---|
| p01 | 0,23 | 0,04 | 0,01 | 0,72 | 0,08 | 0,04 |
| p02 | 0,05 | 0,02 | 0 | 0,23 | 0,26 | 0,01 |
| p03 | 0,23 | 0,04 | 0,01 | 0,71 | 0,08 | 0,04 |
| p04 | 0,22 | 0,04 | 0 | 9,64 | 3,3 | 0,03 |
| p05 | 0,45 | 0,09 | 0 | 1,72 | 0,08 | 0,06 |
| p06 | 0,85 | 0,19 | 0 | 3,61 | 1,01 | 0,09 |
| p07 | 1,52 | 0,38 | 0,01 | 84,92 | 15,2 | 0,14 |
| p08 | 2,66 | 0,63 | 0,02 | 38,84 | 0,65 | 3,12 |
| p09 | 2,35 | 0,55 | 0,01 | 9,11 | 0,51 | 0,3 |
| p10 | 3,84 | 0,97 | 0,02 | 73,85 | 9,53 | 8,58 |
| p11 | 8,85 | 1,79 | 0,05 | T.O. | 1,89 | 1,34 |
| p12 | 42,09 | 113,3 | 0,13 | T.O. | 40,92 | 1,21 |
| p13 | 0,5 | 0,12 | 0 | T.O. | T.O. | 907,12 |
| p14 | 1,22 | 0,21 | 0,01 | T.O. | T.O. | T.O. |
| p15 | T.O. | T.O. | 0,01 | T.O. | T.O. | T.O. |
| p16 | 18,69 | 1,73 | 0,02 | O.O.M | T.O. | T.O. |
| p17 | T.O. | T.O. | T.O. | O.O.M | T.O. | T.O. |
| p18 | T.O. | 1652,43 | 0,28 | O.O.M | T.O. | T.O. |
| p19 | 495,44 | 14,04 | 0,04 | O.O.M | T.O. | T.O. |
| p20 | T.O. | T.O. | T.O. | O.O.M | T.O. | T.O. |
| p21 | T.O. | T.O. | 0,05 | O.O.M | T.O. | T.O. |
| p22 | 4,37 | 0,99 | 0,01 | O.O.M | T.O. | T.O. |
| p23 | 3,5 | 0,18 | 15,48 | O.O.M | T.O. | T.O. |
| p24 | T.O. | 0,41 | 1,37 | O.O.M | T.O. | T.O. |
| p25 | T.O. | 11,59 | 0,05 | O.O.M | O.O.M | T.O. |
| p26 | T.O. | T.O. | T.O. | O.O.M | O.O.M | T.O. |
| p27 | T.O. | T.O. | T.O. | O.O.M | O.O.M | T.O. |
| p28 | T.O. | 324,48 | 0,49 | O.O.M | O.O.M | T.O. |
| p29 | T.O. | T.O. | 0,24 | O.O.M | O.O.M | T.O. |
| p30 | T.O. | T.O. | T.O. | O.O.M | O.O.M | T.O. |

Table 4.5 shows the time each program spends on the solving stage for each instance. The information in the table is also displayed in a chart. Figure 4.3 represents the solving times of each encoding.

Overall, the solving performance is more chaotic than the grounding performance. Particularly, in encodings for the standard model, the performance levels hit two extremes: either a program solves an instance very quickly (in under 5 seconds), or it does not solve the instance at all. There are few cases in between, and they do not follow any visible pattern. In the encodings for the competition model, the problem shifts to the large ground files created in the grounding process. In many occasions, clasp runs out of memory while performing the search, and in some cases the program cannot even finish reading the ground file. However, this is not the only factor contributing to the lack of success of the competition model encodings, as the CMBO and CMCR encodings are unable to solve some instances with relatively small ground files (less than 10 MB).

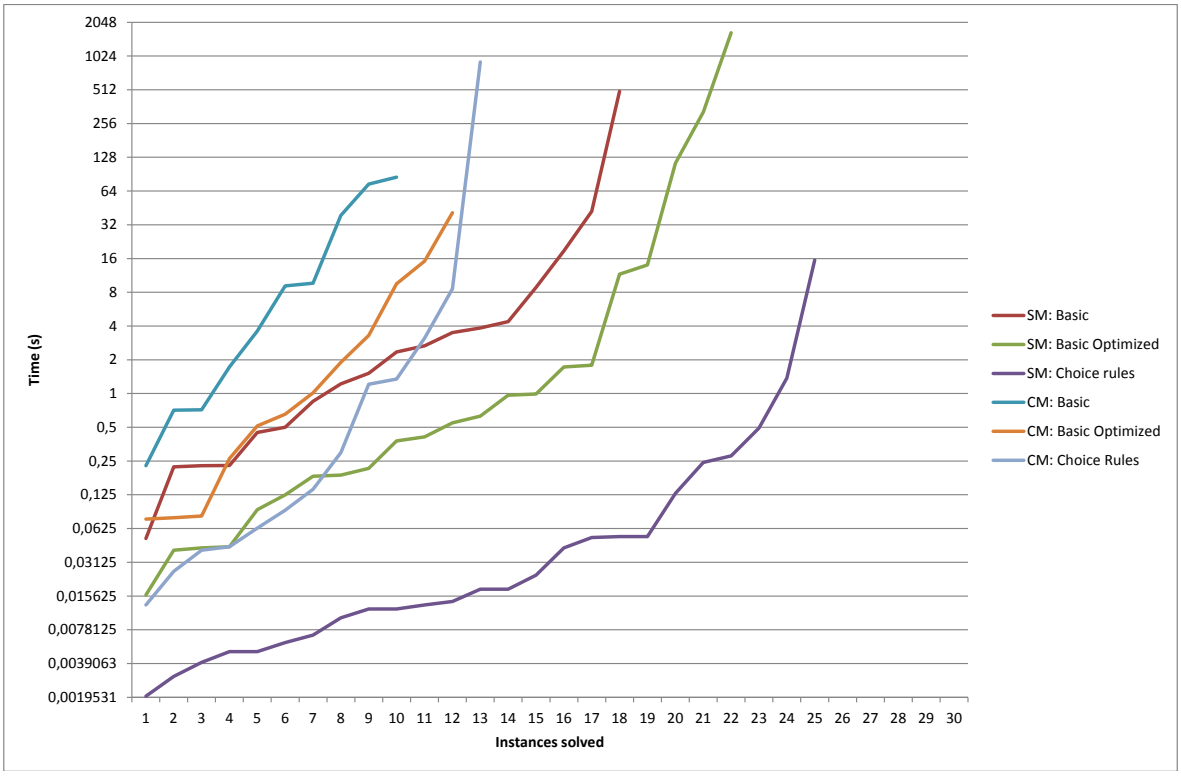


Figure 4.3: Time spent on solving

### 4.2.2 Evaluating clasp options

In an attempt to improve the program's performance, we tested different configurations of clasp's parameters. For this experiment we used the choice-rule based encodings, as they provided the best optimization with respect to the grounding files. The parameters[10] manipulated in this test were the following:

- *−−seed=n* - changes the seed of the random number generator to *n*.
- *−−trans-ext=all* - transforms all choice rules into boolean rules.
- *−−sat-prepro* - enables SAT preprocessing.
- *−−rand-prob* - enables random probing.

Tables 4.6 and 4.7 show the results of configurations tested in the SMCR encoding and the CMCR encoding, respectively. These results are also represented in Figure 4.4 and Figure 4.5 in chart form. The configurations tested are the following:

- Default - uses clasp default options.
- Pre-Pro - uses options *−−trans-ext=all* and *−−sat-prepro*.
- Trans-Ext - uses option *−−trans-ext=all*.
- Random - uses options *−−seed=n* and *−−rand-prob* to run the program 30 times, giving 1 minute to each run.

Because the encodings we are testing are based on choice rules, we ran configurations *Pre-Pro* and *Trans-Ext* to test whether transforming choice rules is beneficial to clasp's performance. The idea behind the *Random* configuration came from analysing 4.5: we observed that, for the SMCR encoding, clasp assumed its most extreme behaviour. The larger amount of time it spent successfully solving an instance was around 15 seconds. Therefore, we were led to believe the performance of the solving process depends strongly on clasp searching the right region of the search tree upfront. With the *Random* configuration, we explore different parts of the search tree on each run, assuming that some regions are closer to valid solutions than others.

From this results, we can conclude that the random approach is able to improve success rate by solving 28 of the 30 instances of the SMCR encoding. In both cases, using pre-processing helps to solve new instances. However, it can also stop clasp from solving instances that it solved successfully using the default configurations. Overall, the time spent on pre-processing increases the total time spent on the solving stage.

We also tested the claspfolio[13] tool with both encodings, but there was not an improvement in performance compared to the default configuration.

Table 4.6: Configurations - standard model (s)

| | Default | Pre-Pro | Random |
|---|---|---|---|
| p01 | 0,01 | 0,07 | 0,07 |
| p02 | 0 | 0,02 | 0 |
| p03 | 0,01 | 0,07 | 0,06 |
| p04 | 0 | 0,03 | 0,01 |
| p05 | 0 | 0,07 | 0 |
| p06 | 0 | 0,1 | 0 |
| p07 | 0,01 | 0,15 | 0,01 |
| p08 | 0,02 | 0,47 | 0,17 |
| p09 | 0,01 | 0,24 | 0,02 |
| p10 | 0,02 | 0,34 | 0,02 |
| p11 | 0,05 | 1,64 | 0,21 |
| p12 | 0,13 | - | 2,72 |
| p13 | 0 | 0,09 | 0,01 |
| p14 | 0,01 | 0,12 | 0,04 |
| p15 | 0,01 | 0,14 | 0,04 |
| p16 | 0,02 | 0,5 | 0,11 |
| p17 | - | 2,39 | 0,2 |
| p18 | 0,28 | 0,91 | 0,39 |
| p19 | 0,04 | 1,28 | 0,42 |
| p20 | - | 8,57 | 71,61 |
| p21 | 0,05 | 1,6 | 0,54 |
| p22 | 0,01 | 0,27 | 0,08 |
| p23 | 15,48 | 10,88 | 1,24 |
| p24 | 1,37 | 2,87 | 0,64 |
| p25 | 0,05 | 1,19 | 0,2 |
| p26 | - | - | 227,38 |
| p27 | - | - | - |
| p28 | 0,49 | 11,05 | 1,13 |
| p29 | 0,24 | - | 61,77 |
| p30 | - | - | - |
| Solved | 25/30 | 25/30 | 28/30 |

Table 4.7: Configurations - competition model (s)

| | Default | Pre-Pro | Trans-Ext |
|---|---|---|---|
| p01 | 0,04 | 1,63 | 0,16 |
| p02 | 0,01 | 0,46 | 0,03 |
| p03 | 0,04 | 1,64 | 0,15 |
| p04 | 0,03 | 1,87 | 68,55 |
| p05 | 0,06 | 1,67 | 0,13 |
| p06 | 0,09 | 4,93 | 0,2 |
| p07 | 0,14 | 55,15 | 0,33 |
| p08 | 3,12 | 36,2 | 0,51 |
| p09 | 0,3 | 12,4 | 0,6 |
| p10 | 8,58 | 32,02 | 0,93 |
| p11 | 1,34 | 126,57 | 1,64 |
| p12 | 1,21 | 214,34 | 2,03 |
| p13 | 907,12 | 940,85 | - |
| p14 | - | - | - |
| p15 | - | - | - |
| p16 | - | - | 2,92 |
| p17 | - | - | - |
| p18 | - | - | - |
| p19 | - | 17,52 | 16,16 |
| p20 | - | - | - |
| p21 | - | - | - |
| p22 | - | - | - |
| p23 | - | - | - |
| p24 | - | - | - |
| p25 | - | - | - |
| p26 | - | - | - |
| p27 | - | - | - |
| p28 | - | - | - |
| p29 | - | - | - |
| p30 | - | - | - |
| Solved | 13/30 | 14/30 | 14/30 |

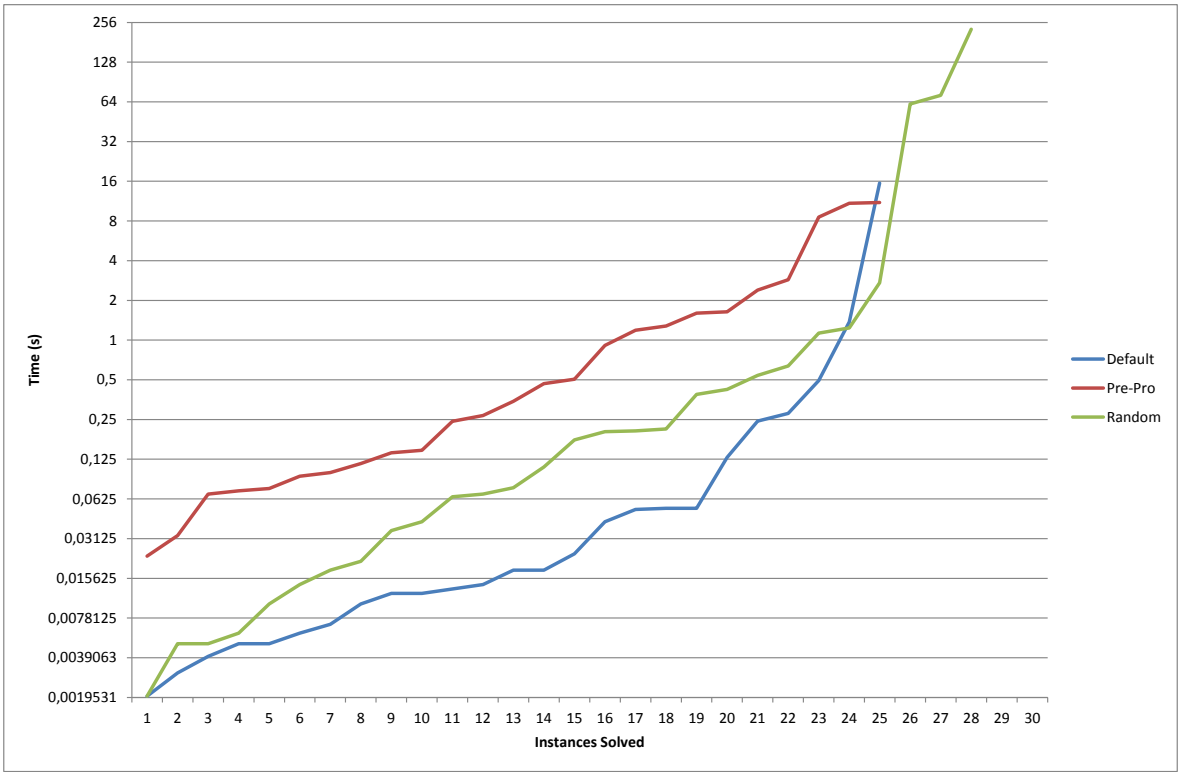


Figure 4.4: Configurations - Standard Model

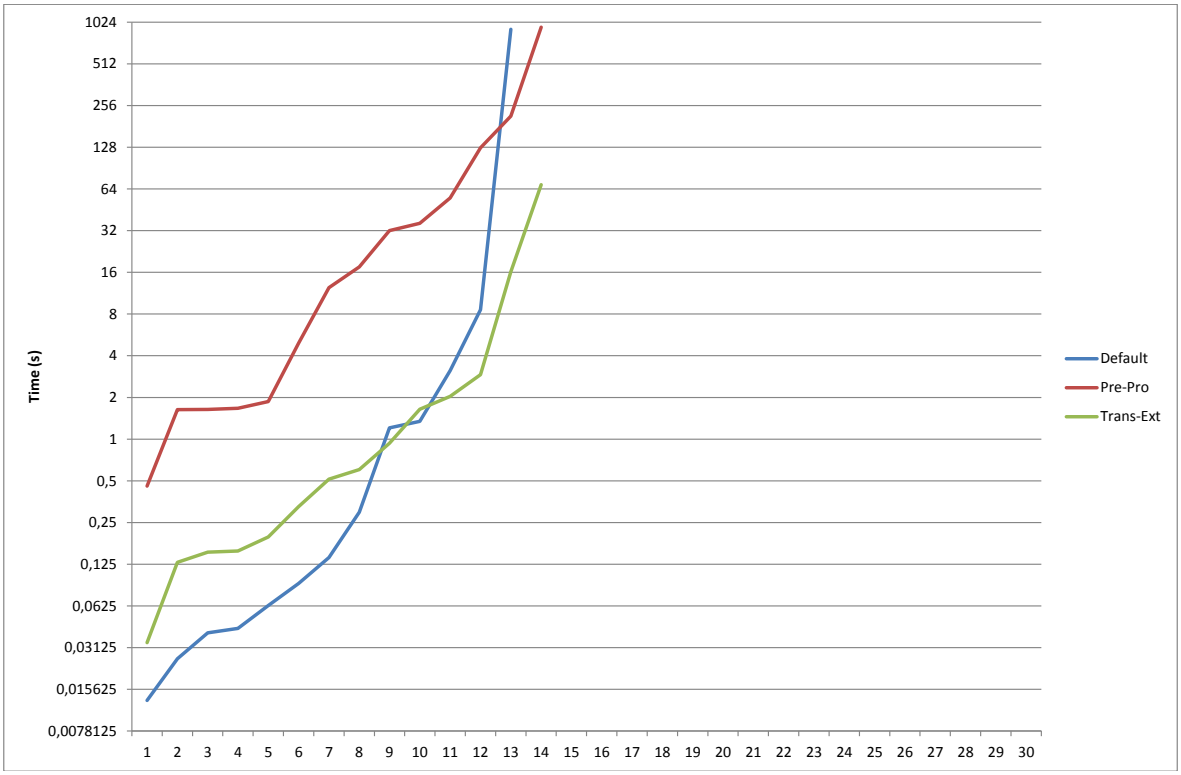


Figure 4.5: Configurations - Competition Model

Table 4.8: CMBO - Optimal Instances (s)

| | Grnd T | Solv T | Total T | Size |
|---|---|---|---|---|
| p01 | 0,13 | 0,08 | 0,21 | 0,28 |
| p02 | 0,06 | 0,26 | 0,33 | 0,11 |
| p03 | 0,12 | 0,08 | 0,2 | 0,28 |
| p04 | 0,17 | 3,3 | 3,46 | 0,4 |
| p05 | 0,17 | 0,08 | 0,25 | 0,57 |
| p06 | 0,34 | 1,01 | 1,35 | 1,29 |
| p07 | 0,67 | 15,2 | 15,87 | 2,73 |
| p08 | 0,81 | 0,65 | 1,47 | 3,28 |
| p09 | 0,56 | 0,51 | 1,07 | 2,48 |
| p10 | 1,17 | 9,53 | 10,71 | 6,04 |
| p11 | 1,93 | 1,89 | 3,83 | 9,24 |
| p12 | 2,99 | 40,92 | 43,91 | 14,4 |
| p13 | 1,1 | - | - | 2,69 |
| p14 | 1,76 | - | - | 4,79 |
| p15 | 1,91 | - | - | 4,7 |
| p16 | 8,24 | - | - | 31,49 |
| p17 | 14,02 | - | - | 56,6 |
| p18 | 15,96 | - | - | 61,19 |
| p19 | 27,41 | - | - | 117,58 |
| p20 | 46,71 | - | - | 203,13 |
| p21 | 57,47 | - | - | 243,74 |
| p22 | 6,52 | - | - | 15,86 |
| p23 | 9,5 | - | - | 26,96 |
| p24 | 10,85 | - | - | 30,93 |
| p25 | 47,69 | - | - | 165,12 |
| p26 | 66,87 | - | - | 243,71 |
| p27 | 75,88 | - | - | 282,95 |
| p28 | 151,34 | - | - | 595,85 |
| p29 | 233,3 | - | - | 963,45 |
| p30 | 265,38 | - | - | 1066,19 |

Table 4.9: CMBO - Suboptimal Instances (s)

| | Grnd T | Solv T | Total T | Size |
|---|---|---|---|---|
| p01 | 0,14 | 0,04 | 0,18 | 0,32 |
| p02 | 0,12 | 0,03 | 0,15 | 0,24 |
| p03 | 0,14 | 0,04 | 0,18 | 0,32 |
| p04 | 0,18 | 0,14 | 0,33 | 0,45 |
| p05 | 0,47 | 0,25 | 0,73 | 1,81 |
| p06 | 0,65 | 0,41 | 1,06 | 2,55 |
| p07 | 0,85 | 0,61 | 1,46 | 3,56 |
| p08 | 0,82 | 0,65 | 1,47 | 3,28 |
| p09 | 1,55 | 1,2 | 2,75 | 7,51 |
| p10 | 2,17 | 1,76 | 3,93 | 10,78 |
| p11 | 2,23 | 2,28 | 4,51 | 10,67 |
| p12 | 2,99 | 40,78 | 43,77 | 14,4 |
| p13 | 1,74 | 1,12 | 2,85 | 4,72 |
| p14 | 2,15 | 1,45 | 3,6 | 6 |
| p15 | 1,99 | - | - | 4,95 |
| p16 | 11,98 | 17,74 | 29,72 | 46,32 |
| p17 | 15,27 | - | - | 61,46 |
| p18 | 15,93 | - | - | 61,19 |
| p19 | 39,85 | 102,39 | 142,25 | 171,38 |
| p20 | 50,7 | - | - | 220,5 |
| p21 | 57,7 | - | - | 243,74 |
| p22 | 8,74 | 6,51 | 15,25 | 22,84 |
| p23 | 9,62 | - | - | 27,34 |
| p24 | 10,94 | - | - | 31,36 |
| p25 | 63,69 | 162,29 | 225,98 | 239,64 |
| p26 | 72,95 | - | - | 271,45 |
| p27 | 75,96 | - | - | 282,95 |
| p28 | 192,74 | - | - | 779,7 |
| p29 | 259,48 | - | - | 1067,75 |
| p30 | 265,26 | - | - | 1066,19 |

### 4.2.3 Evaluating suboptimal instances

In an earlier stage of development, the instances used to evaluate the encodings for the competition model were slightly different. The rule to calculate the range of *period* domain was different: the maximum period would take the highest value between the value calculated with Algorithm 1 and the formula $96 * D$, in which $D$ is the number of days of the plan. This method of calculation was changed because we wanted to have a maximum period as close to the optimum value as possible. In this test, we show the implications of working with an suboptimal set of instances. We use the CMBO encoding in this evaluation.

In most cases, the ground files for suboptimal instances are larger than for optimal instances. The exceptions are the instances where the *period* domain is the same in both groups of instances. In those instances the ground files were the same size and the times were similar.

However, the solver was more successful when solving suboptimal instances, despite the larger file size. We believe this happens because, with a larger *period* domain, the tasks can be spread through time

Table 4.10: Time spent on solving (s)

| | POPF-2 | SGPlan-6 | Best SM | Best CM |
|---|---|---|---|---|
| p01 | 81,21 | 0,06 | 0,14 | 1,91 |
| p02 | 68,31 | 0,06 | 0,05 | 0,6 |
| p03 | 80,79 | 0,05 | 0,14 | 1,92 |
| p04 | 71,56 | 0,06 | 0,08 | 2,14 |
| p05 | 65,05 | 0,06 | 0,16 | 2,33 |
| p06 | 59,77 | - | 0,2 | 5,91 |
| p07 | 56,23 | - | 0,26 | 56,63 |
| p08 | 64,7 | - | 0,48 | 38,31 |
| p09 | 61,27 | - | 0,38 | 14,88 |
| p10 | 61,36 | - | 0,45 | 35,57 |
| p11 | 66,46 | - | 0,86 | 132,05 |
| p12 | 71,76 | - | 3,4 | 220,93 |
| p13 | 74,77 | 478,66 | 0,17 | 942,33 |
| p14 | 67,43 | 357,63 | 0,22 | - |
| p15 | 52,29 | 423,71 | 0,22 | - |
| p16 | 50,5 | - | 0,48 | - |
| p17 | 53,27 | - | 0,8 | - |
| p18 | 54,87 | - | 0,84 | - |
| p19 | 57,17 | - | 1,19 | 50,16 |
| p20 | 64,63 | - | 72,93 | - |
| p21 | 63,37 | - | 1,44 | - |
| p22 | 48,69 | 542,38 | 0,4 | - |
| p23 | 76,47 | - | 1,69 | - |
| p24 | 53,75 | - | 1,05 | - |
| p25 | 47,06 | - | 1,44 | - |
| p26 | 48,88 | - | 228,72 | - |
| p27 | 52,77 | - | - | - |
| p28 | 68,14 | - | 2,89 | - |
| p29 | 76,55 | - | 63,02 | - |
| p30 | 80,85 | - | - | - |
| Solved | 30/30 | 9/30 | 28/30 | 14/30 |

with less risk of overlapping. Thus, it is easier to find a solution for a suboptimal instance.

### 4.2.4 Comparing ASP encodings to competition planners

After analysing the different encodings and finding out the best configurations, we compare them to planners that participated in the IPC. We tested the planner with best results in the crew planning domain in two competitions: SGPlan6[35], from IPC-2008, and POPF2[6, 7], from IPC-2011. We compared these planners to the best configurations presented in section 4.2.2: the choice rule encoding for the standard model using random probing, and the choice rule encoding for the competition model using pre-processing of choice rules.

Comparing the different planners was a difficult task, due to the different behaviours each planner exhibited. POPF2 would find an optimal solution in a few seconds, but afterwards it would engage in a depth-first search until it ran out of memory. SGPlan6 solved all instances, but only 9 of them optimally.

For other instances, the solutions presented were not optimal, as the planner could not fit all the tasks in a plan with the required number of days.

In the end, the most important criteria is the number of instances solved optimally, and in this respect our encodings did not beat POPF2. However, both encodings solved more instances optimally than SGPlan6. In terms of time, the standard model encoding got the best performance, with occasional exceptions. SGPlan6 displayed a worse performance for all instances where the number of days was higher than the number of crew members. POPF2 showed a stable performance over all instances.

# 5 Conclusion

In this chapter we will conclude this work. We will discuss our opinion on the solutions developed and the experimental results. We will also point possible directions of follow-up work.

During the development of this work, our goal was set on solving the 30 instances used in the International Planning Competition. We also wanted to, as much as possible, present solutions similar to the ones given by the competition planners. However, there was a conflict of interest between what was the best approach to take in terms of maximizing ASP performance, and present a solution as similar as possible to the competition output. Because of this, our work diverged into two parts, which although very strongly linked, were fundamentally different.

Our best ASP approach solved 28 of the 30 instances, so we did not achieve the proposed goal. Is this a good or a bad result? It may not look good when compared to the best planners of the 2011 planning competition, which correctly solved all the instances. But, after looking at how this thesis work unfolded, we think of this as a great result. The divergence that occurred in our work shows that ASP works in a radically different way from a dedicated planner, as its results were much better under the standard model than under the competition model. ASP is a logic programming language, and its strong point is to solve binary problems. However, there is nothing binary about the crew planning problem. Every task must be in the plan, and the true question is to what time each task is scheduled. That is a question that may have a hundred, or more, possible answers. We have repeatedly shown how the grounding process suffered from having to compile rules for numeric domains, generating a huge amount of ground constraints that were, in several cases, overloading the solver with information. Despite all this problems, that show ASP was out of its comfort zone, we still managed to solve all but 2 instances. And for us, that makes it a great success.

## 5.1 Future Work

We have worked hard to solve the crew planning problem and cover many hypothesis as possible. However, there are still areas of the problem we have not tackled, which can serve as future work. The following is a list with some suggestions of work.

- We presented two different sets of constraints for the crew planning problem: one in section 2.1 – Crew Planning, when introducing the problem for the first time, and the other in section 3.1 – Plan constraints, a simplification used by the IPC and that we adopted. Creating a program that solves

the crew planning problem according to the first set of constraints and study its behaviour to see how it would differ from the programs we developed would make an interesting work.

- In the 2008 planning competition, the crew planning problem appeared also as an optimization problem, consisting of making an optimal choice of tasks, according to the utility of each task, to fit into a plan of fixed time. This version of the problem was not solved successfully in the 2008 competition, and was not used in the 2011 competition. We would like to know whether an ASP planning program with the goal of optimizing the utility value would be successful.

- The ASP programs we developed in this work were tailor-made to solve the crew planning problem. It might be possible to take a different approach to planning in ASP, by developing a program that works as a generic planner/scheduler. We are interested to see if such an approach is feasible, and if it is, how it performs compared to our work.

# A Decision system for the Space Shuttle

Nogueira et. al.[27, 1] designed one of the most successful applications of ASP: The USA-Advisor, a decision system to support the flight controllers of the Space Shuttle, particularly the **Reaction Control System** (RCS). We present this work due to the similarities to the work we describe in this paper; it is a space-related planning problem that was solved using ASP.

The RCS is responsible for the maneuvers of the Space Shuttle while it is in orbit. In order to perform a maneuver, the RCS may require the astronauts to perform certain operations in order to prepare the system for the maneuver. The goal of the ASP system is to find a plan with the operations the astronauts need to do in order to prepare the RCS. To achieve so, the ASP system has access to detailed information about the state of the RCS, regarding its plumbing system and its electronic system. The system is divided into several modules, which are described as follows.

The **plumbing module** determines if the current state of the plumbing system allows the execution of the required maneuver. The plumbing system is composed of fuel and oxidizer tanks, jets, pipe junctions and valves which control the flow of fluid through pipes. Fluid flows from the tanks to the jets by passing through connected pipes, as long as the valves are open and there are no leaks. The maneuver can be executed if all the jets it requires are working properly and receiving fuel and oxidizer. The information generated by this module is expressed in the form of the predicate $pressurized\_by(N, Tk)$, which means that node $N$ (a node is either a tank, a jet or a pipe junction) is supplied with fluid from tank $Tk$.

The **valve control module** is responsible for determining the position of each valve. This information is needed by the plumbing module in order to model the fluid network. The position of a valve may be controlled by using either mechanical switches or electronic commands. In order to know the current position of a valve, this module takes as input the information about the initial positions of switches and valves, the actions performed until the current instant, and the faults in the system, which may be stuck valves and switches, or electronic malfunctions. In the most advanced encoding of this module, the switches are connected themselves to the electrical circuits, so that a switch updates the information in the circuit when it is flipped. As such, the circuit always describes the position of the valves accurately, unless the circuit itself is faulty. The predicate $in\_state(V, S)$ indicates that the valve $V$ is in the state $S$ (which might be *open* or *closed*).

The **circuit theory module** describes the functioning of the electrical circuits of the RCS. A circuit is composed of gates (electric components) which are connected by wires. The goal of this module is

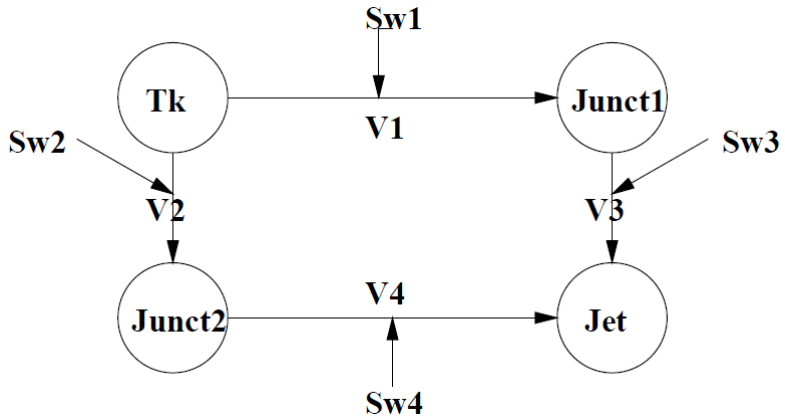


Figure A.1: Simplification of the RCS[1]

to find the values on the *output wires* (wires which send signal to the valves), based on the description of the circuit and the values of the *input wires* (wires which receive signal from diverse components). The electric signal is described as 3-valued, i.e., it may be assigned to the values 1, 0 and unknown. Faults in the circuit consist of gates stuck at a signal value. The predicate $value(C, X)$ indicates that the component $C$ (which may be a wire or a gate) has the signal value $X$.

The **planning module** decides which actions should be performed in the RCS at an instant in time. Possible actions include flipping a switch and issuing a command. The structure of the RCS allows the RCS to be divided in three separate subsystems. Because of this, the planning module generates not one, but three plans – one for each subsystem of the RCS – and executes them simultaneously. As such, the ultimate goal of the system is to achieve the sub-goals designed for each subsystem. The division into subsystems greatly improves the performance of the planning system. The planning module is also responsible for applying heuristics to the planning process. This is done in the form of ASP constraints. The constraints are used to exclude plans that, although fulfilling all the previous restrictions, are undesirable; and at the same time, to improve performance. For example, one of the heuristics says that one node should not be pressurized by two different paths, so it prevents plans that include two actions intended to pressurize the same node.

In [1], it is shown that the addition of heuristic provides better plans and much faster computation of those. The generated plans were minimal for the standard set of instances, and even when faced with more difficult instances, the planner never took more than 15 minutes to find a plan.

This work is an example of the successful use of the concepts of modularity and parallel planning, and showed that ASP is a good tool to model information and solve planning problems.

# Bibliography


[1] Marcello Balduccini, Michael Gelfond, Richard Watson, and Monica Nogueira. The usa-advisor: A case study in answer set planning. In *LPNMR*, pages 439–442, 2001.

[2] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, February 2003.

[3] J. Barreiro, G. Jones, and S. Schaffer. Peer-to-peer planning for space mission control. In *Aerospace conference, 2009 IEEE*, pages 1 –9, march 2009.

[4] Martin Brain, Owen Cliffe, and Marina De Vos. A pragmatic programmer's guide to answer set programming. In *Software Engineering for Answer Set Programming (SEA09)*, pages 49–63, September 2009.

[5] Owen Cliffe, Marina De Vos, Martin Brain, and Julian A. Padget. Aspviz: Declarative visualisation and animation using answer set programming. In *ICLP*, pages 724–728, 2008.

[6] Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *ICAPS*, pages 42–49, 2010.

[7] Andrew Coles, Amanda Jane Coles, Allan Clark, and Stephen Gilmore. Cost-sensitive concurrent planning under duration uncertainty for service-level agreements. In *ICAPS*, 2011.

[8] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

[9] Martin Gardner. Langford's problem. In *Mathematical Magic Show*, page 70. Vintage, 1978.

[10] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to gringo, clasp, clingo, and iclingo. Unpublished draft, 2010. Available at URL[1].

[11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental asp solver. In *ICLP*, pages 190–205, 2008.

[12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Challenges in answer set solving. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, pages 74–90, 2011.


[1] `http://heanet.dl.sourceforge.net/project/potassco/potassco_guide/2010-10-04/guide.pdf`

[13] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

[14] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp* : A conflict-driven answer set solver. In *LPNMR*, pages 260–265, 2007.

[15] Martin Gebser, Orkunt Sabuncu, and Torsten Schaub. An incremental answer set programming based system for finite model computation. *AI Commun.*, 24(2):195–212, 2011.

[16] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In *LPNMR*, pages 266–271, 2007.

[17] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[18] Malik Ghallab, Craig K. Isi, Scott Penberthy, David E. Smith, Ying Sun, and Daniel Weld. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[19] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004.

[20] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: algorithms and complexity. In S. C. Graves, A. H. G. Rinnooy Kan, and P. H. Zipkin, editors, *Logistics of Production and Inventory*, volume 4 of *Handbooks in Operations Research and Management Science*, chapter 9, pages 445–522. Elsevier, 1993.

[21] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[22] Yuliya Lierler and Marco Maratea. Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In *LPNMR*, pages 346–350, 2004.

[23] Fangzhen Lin and Yuting Zhao. Assat: computing answer sets of a logic program by sat solvers. *Artif. Intell.*, 157(1-2):115–137, 2004.

[24] Toni Mancini, Davide Micaletto, Fabio Patrizi, and Marco Cadoli. Evaluating ASP and Commercial Solvers on the CSPLib. *Constraints*, 13(4):407–436, 2008.

[25] Victor W. Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. *CoRR*, cs.LO/9809032, 1998.

[26] Ilkka Niemela. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3):241–273, 1999.

[27] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An a-prolog decision support system for the space shuttle. In *PADL*, pages 169–183, 2001.

[28] Edwin P. D. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[29] Olivier Roussel. Controlling a solver execution with the runsolver tool. *JSAT*, 7(4):139–144, 2011.

[30] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[31] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

[32] Stephen J. J. Smith, Dana S. Nau, and Thomas A. Throop. Computer bridge - a big win for ai planning. *AI Magazine*, 19(2):93–106, 1998.

[33] Tommi Syrjänen. Omega-restricted logic programs. In *LPNMR*, pages 267–279, 2001.

[34] Tommi Syrjänen. On the practical side of answer set programming. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, pages 473–489, 2011.

[35] Chih wei Hsu and Benjamin W. Wah. The sgplan planning system in ipc-6.

[36] Claudia Zepeda and David Sol. Evacuation planning using answer set programming: An initial approach. *Engineering Letters*, 15(2):240–249, 2007.