Domain independent planning for space: Building a bridge from both shores

Marcelo Oglietti
CONAE - Argentine National Space Agency
Paseo Colon 751, (1063) Buenos Aires
Argentina
marcelo.oglietti@conae.gov.ar

Abstract.
Most of the effort dedicated to bridging the gap between real life planning and domain independent planning has been put in enhancing the latter. In this paper we follow the opposite direction: what can we do in space engineering to narrow this gap? We present some of the problems that make impractical the use of domain independent planning when following the standard architectural design schemes used for space systems. We present a different architectural scheme that avoid these problems turning easier the use of domain independent planning. We briefly describe a real life application based on this new approach: a multi-mission multi-sensor ground station service that manages many Tracking, Telemetry & Command, and science data downloading antennas.

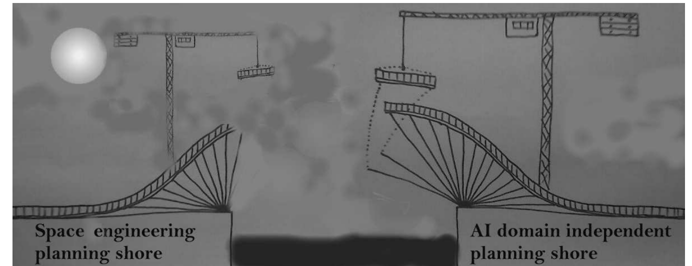

Figure 1: Bridging the gap between AI domain independent planning and space engineering planning

1 Introduction

During last years, the AI planning research community has been enhancing various domain independent planning frameworks in different complementary directions in order to manage more complex domains, e.g. (Laborie 2001; Petrick and Bacchus 2002; Muscettola 2002; Fox and Long 2003). In spite of all this effort, domain independent planning is usually not considered mature enough for space mission planning. A tacit agreement inside the space engineering community is that current planning domain representation languages are not enough for the complexities of space mission planning. Hence, the solutions adopted in order to implement planning tools for these missions are mostly *ad-hoc* highly domain dependent solutions.

In this paper we analyze the standard architectural design approach for space missions. We confront it with current assumptions about the modeling of the world in AI domain independent planning, in order to detect which are the main characteristics that turn its planning problem not tractable with a domain independent planning approach.

We detect that a main issue is the lack of a unified methodology that determines how the state of each unit in the system can be known and represented. This is necessary to guarantee that we can know and represent the state of the whole system that is simple the aggregation of these units.

In addition, we notice that in space engineering there is not an established criteria to manage the possible sources of change of the state of a unit. This change might happens by random or predictable phenomena, and as a product of both exogenous or internal events, and not only by commanding it.

These two characteristics of current standard space engineering practice are a severe impediment when trying to use automated domain independent planning. For this kind of planning is necessary to know how to query and represent the state of the system, and to understand and model all the causes that change its state.

We present a simple general architectural design approach that overcome these problems. An extra positive byproduct of this approach is that by using it we can automate the detection of conflicts between distinct partial configurations of a system.

Finally, we presents an application of this new approach to a real life space mission. We implement the CONAE Ground Stations Service (CGSS), the service that manage all the CONAE's ground stations, using this proposed common architectural design.

2 Current Space Engineering Architectural Design Methodology

There is a established space engineering methodology described in many books, e.g. (Fortescue *et al.* 2003; Larson and Wertz 1999; Boden and Larson 1996), and underlined in some international standardization efforts like the European

Cooperation for Space Standardization (ECSS).

Space Missions are divided in two main segments: the Flight and the Ground Segment (GS). The Flight Segment (FS) might be composed by more than one spacecraft. Each spacecraft is designed by following a functional division in its main subsystems: thermal control, attitude control, communication and data handling, power control, etc. In turn, these subsystems are decomposed in various units that are a lower layer in the architectural hierarchical subdivision of FS. Most GS's are composed by various services like the ground station service, the user ground segment, the individual spacecraft and payload control centers, etc. Again, these services are decomposed in various units that are a lower layer in the architectural hierarchical subdivision of GS. For the purposes of this paper, the decomposition can be stopped at this level, both for FS and GS.

In what follows, in order to keep a reasonable level of abstraction, we will ignore the presence of the hierarchical decomposition of the space mission, focusing on its lowest layer: the units that compose each subsystem of both FS and GS. We assume that the space mission is composed by N units identified as U_i, with $i = 1, \ldots, N$. We will proceed as if the whole space mission is given by the aggregation of these units. Sometimes, we will refer to the space mission simply as *the system*.

A unit might have many interfaces: electrical; mechanical; for communications, both for Telemetry, Tracking and Commands (TT&C) and for science data downloading; networking; for the monitor and control (M&C); etc. These various links between the units are entangled, and many of them depend on the others. For example, M&C depends on the most other interfaces to be operative, it depends on the electrical interfaces to get the power and signals, on the communication and networking interfaces to transmit all valuable data, etc. For planning we need to consider only the M&C interfaces because is through these interfaces that the plan execution is run. The other interfaces can be ignored as if they are just as part of the equipment that need to be monitored and controlled, independently that M&C itself depends on their proper function to be operative. For simplicity, from now on, we will ignore completely the existence of all other interfaces but those for M&C, and refer them simply as the unit interfaces dropping the M&C keyword.

Interfaces and Information Hiding. Current practice emphasizes the importance of a clear definition and control of all the interfaces between the units. From any project management point of view this is done by means of the Interface Control Document (ICD). The ICDs should register every important specification of all the interfaces of each unit. These documents are used as contracts between the various teams that design and develop the units. The use of these documents is the single mechanism that guarantee the feasibility of the labor division inside every space mission. Being most of the M&C done by software, and considering that nowadays the Object-Oriented (O-O) methodology(Meyer 1988; Abadi and Cardelli 1996) is the most ubiquitous used for the design of software, it is not surprising that the M&C's ICDs are resembling more and more the specification of an O-O object interface (i.e. the object's class interface). To simplify even further this analysis, we will adopt this approach and consider that the interface of each unit U_i is completely specified by two types of features: variables and methods. We have two types of variables: read-only variables, whose values can only be read; and read-write variables, whose values can be read and written. Also we have two main types of methods: procedures, that do not return any data; and functions, that return some data about the unit. The methods might have parameters that need to be instantiated when calling them. The variable and the parameters of the methods can be of many distinct types: integer or real numbers, strings, time points values, bytes, boolean, etc. Frequently, only some restriction of these basic types are allowed (e.g. only positive integers greater than 10). It is current practice that to control a unit we execute an instantiation of the appropriate procedure method in the unit interface or assign a new value to any of its read-write variables. To monitor a unit we read the values of the variables in its interface and/or call the adequate function method in the unit interface. Both procedures and functions methods might change the state of the unit. The fact that the function method might change the internal state of the unit is considered a negative *side effect* that can create difficulties when developing programs, and so, consequently it is discouraged by some authors (Meyer 1988). There is no established methodology that avoid this side effect.

The design of any unit interface follows many well established practices in order to make the M&C more robust and reliable; e.g. to minimize the number of interfaces between the units, to define each unit's interface explicitly, to minimize the size of information interchange through it, etc. Most of these practices are based in the *Information Hiding* principle that establishes that all internal data of a unit must be private to the unit itself unless it is necessary to interchange it, and in this case, it should be specifically declared as part of the unit interface, i.e. as one of the unit interface variables.

Knowing and representing the state of the system. At this point, it is important to highlight that the state of a unit is completely determined by the status of its internal private data. This is usually hold in a set of internal private variables, whose values *represent* the state of the unit. Sometimes these values come from the commanding of the M&C system, sometimes they come from the sensing of some physical properties of the unit. Following the information hiding principle, we hide all unit internal data behind its interface. Notably, by agreeing with this *auto-limitation*, we

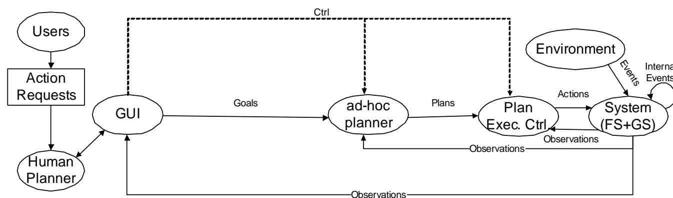

Figure 2: Automated *ad-hoc* on-line planning for space

gain a lot in terms of engineering robustness and and reliability, as explained in the previous paragraphs. In the process we lost the *potential* capability of clearly know the state of the unit by accessing the unit internal data directly[1]. Nowadays, what can be known about the internal state of a unit is in the hands of the designer of its interface. The designer choose what variables appears in the interface and what can be queried through the methods in the interface. A *drawback* of the information hiding principle in its current version is that it do not establish any criteria about this selection.

The main point we want to state in this section is that in space engineering there is not a unified methodology that guarantees that the state of each unit can be known. Consequently, there is even a greater lack of methodology about how to represent the state of each unit. Hence, it is usually impossible to know and represent the state of the whole system.

It is assumed that the internal state of a unit can change by itself, following exogenous events or some internal logic of the unit, and not only by commanding it by executing any of the method in its interface. These events can be random or predictable. There is not an established criteria to distinguish these possible sources of change in a unit state.

What the last two paragraphs state is a severe problem when trying to use automated domain independent planning for space. It is vital for this kind of planning to know how to query and represent the state of the system, and to understand and model all the causes that change this state. We explain this in the next sections.

3 Automated *ad-hoc* on-line planning

Figure 2 shows a broad conceptual view of what is usually done for the planning of space missions. Usually the automated on-line planner is developed *ad-hoc* for each particular mission. The objective of automated planning is to automatically generate *plans* [2] and execute them to control a *system* (including both FS and GS) to make it behave in a certain desired way.

[1]The word *potential* highlights that this was not an established practice before current wide incorporation of the information hiding principle.

[2]In this section we write in italic mostly to mark the concepts that have a counterpart in the figures.

The *users* request actions to be done in order to get from the *system* the products they need (e.g. the request of certain image products). These actions are in general conflicting, need coordination, and priorities are used to resolve them. The set of action requests plus some desired behaviour constrains are specified by a human *planner* by setting some *goals* through a Human Machine Interface tool (called *GUI* in the figure). These *goals* can be of various types. Current practice refers to reachability goals [3] when the request is only to act on the *system* until its internal *state* is such that it satisfies all the requested *goals*. In this case usually it is said that the system has reached a goal state and the plan terminates. It is common to use the terms extended goals when the request goes beyond this, and the goal conditions are requested onto any or some intermediate *states*. For example, in addition to a subset of goal states, it might be requested to avoid certain dangerous states. So, *extended goals* include, as a particular case, *reachability goals*. Also the cases of *plans* that might run forever are included as valid *extended goals*, i.e. *plans* that that might never terminate, always keeping the required *extended goals*. Clearly, these *plans* might be changed for other *plans*, thus terminating them. If some new *goals* are requested by the *user*, the *planner* might produce a new *plan* and replace it for the previous *plan* under execution. The *goals* are passed to the *planner*, a program in charge of generating the *plans* to satisfy the desired *goals*. The generated *plans* are passed to the *controller* which is in charge of executing them. The *controller* controls the *system* by sending it basic commands usually called *actions*. Usually it is said that an *action* is executed whenever the *controller* commands the *system* by sending it that *action*. The *system* can have several internal physical *states*. The set of all the internal *states* of a *system* is called the *State Space*. The execution of an *action* at a given *state* might produce a change of *state*. This new state is usually referred to as the result of the execution of the *action* at that *state*.

The physical state of the *system* can change for several reasons. The commanding of some *action* might change its state. The *environment* into which the *system* is embedded can change its state through *events*. Finally, the *system* can change its state by its own internal *events*. It is not possible to execute an *action* in every state of the *system*. When this is possible we say that the *action* is executable in that *state*. An *action*, even when executed in an executable state, not necessarily produces the desired change of the *system state*, because of the synchronous occurrence of either *environmental events*[4] or *internal events*. In such cases, it is common to say that the *system* is not deterministic. Notice that in these cases the result of the execution of an *action* in a given *state* is not unique. It depends on the synchronous

[3]It is also common to find the term *attainment goals* instead of *reachability goals*.

[4]Sometimes the *environmental events* are called *exogenous events*.

occurrence of the *events*.

There are various types of *plans*. They range from simple ordered lists of *actions*, to entire complex programs (with control structures like `while` or `if`) to be executed by the controller. Sometimes *actions* can be time-tagged, some other times the whole *plan* operates in real time. So, in general, the fact that the *controller* is executing a *plan*, does not mean that it is sending commands to the *system* permanently, it uses part of the execution time in looping, waiting until a particular set of *observations* occur, etc. It is usual to name sequential plan any not time-tagged ordered list of *actions*. In general, the *controller* can gather *observations* about the *system*, and these *observations* are used to change the path of the *plan* execution, e.g. choosing between two paths, depending on the value of a certain *observation*. If it is not always possible to deduce the *state* of the *system* from the *observations* we say that there is partial observability, and planning under this condition is referred to as planning with incomplete information and sensing.

It is common to say that a planner is *on-line* when the *plan* under execution might be permanently modified by the *planner* to adjust it to any unexpected new status of the *system*. This is frequently done by means of a step-by-step generation of parts of the plan time line. The *planner* continuously gathers *observations* about the *system* and might react in real-time generating new *plan* updates. Clearly, in this schema the *planner* is a program that contains hardcoded a model of the *system*, and uses it to produce *plans* given the *goals* requested to it. In fact, in this schema also the representation of the *goals*, *observations* and *plans* is hardcoded in the *planner*, *controller* and *GUI*. In case we need to automatically generate plans to control a different *system*, it is likely that the *planner* program cannot be used without being extensively rewritten. Clearly, *reusability* in this approach is very low.

The *system* state need to be defined *ad-hoc* to overcome the lack of a proper methodology to query and represent the state of the *system* units. This fact does not add any significant new problem to the development of the *planner* because it is an *ad-hoc* planner developed from the scratch for the particular *system*. As we explain in the next section, this is is not the case when we use domain independent planning.

4 The Automated Domain Independent On-line Planning paradigm

Figure 3 shows a broad conceptual view of what is intended as *domain independent* automated *on-line planning* in AI. Basically the situation is the same that the one described in the previous section, but for the fact that now the *planner* does not have hardcoded any information about how the *system* is. The idea is that the *planner* does not know anything a priori (hardcoded) about the *system*. The information about the *system* is passed to the *planner* by means of a written description. A special modeling language is needed for

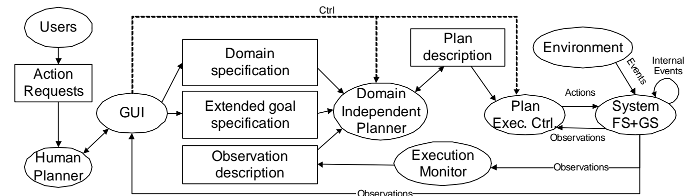

Figure 3: Automated *domain independent on-line planning*

this purpose. We say that the *planner* is *domain independent* when it knows about the *system* only through a model described by modeling language. When describing a *system* using a modeling language, it is common in planning to say that we have specified a *planning domain*.

In domain independent planning, sometimes a unique language is used for all the modeling (e.g. the case of the Situation Calculus (McCarthy 1968)), but it more frequent to use at least two specialized modeling languages: A more basic language to model the internal *state* of the *system*; and another language to model the *actions*, i.e. to model how they change the *state* of the *system*. Usually the definition of the latter relies on the former, because it needs to describe *system states* to specify how actions alter them. This latter approach is the one used in STRIPS (Fikes and Nilsson 1971) and in most modern planning frameworks, mainly heirs of STRIPS in this aspects.

Any language used to specify a *planning domain* is usually called a domain specification language. We call any language used to specify the *state* of the *system* a state description language. A single element of the language represents a *state* of the physical *system*, while the whole language represents its complete *state space*. It is common practice in planning to refer to the state description language as the *state space* and to its elements as *states*, thus, blurring somehow the distinction between representation and reality. It is not a big problem to follow this slightly confusing practice if the reader is aware of the distinction. In case we need to refer to the real physical system, we will state that explicitly. Otherwise, in the rest of this dissertation, whenever we refer to a *state* and a *state space* we mean its corresponding representation in some state description language and the whole language itself. Sometimes the modeling language leaves certain parameters free, with the purpose of representing a family of similar *systems*. Also it is usual to refer any specification of a family of similar systems as a *planning domain*.

So far we have characterized domain independent planning, nevertheless, it is important to highlight that in order to maximize the reusability of the *planner* program, the representation of *goals*, *observations* and *plans* should not be hardcoded. We need languages to describe them: a goal specification language to specify any extended goal; observation description language, to describe the observations;

and a plan description language for representing the *plans*. In Figure 3 we assume the presence of a program, called *execution monitor*, able to both gather the *observations* and provide the planner with a description of them in the corresponding observation description language.

Therefore, to build a *domain independent planner* program it is necessary to choose the following four languages: a *domain specification language*, an *observation description language*, a *goal specification language* and a *plan description language*. The first three languages are used for writing the planner's input, the latter, for its output. Each *planner's* input is called an instance of the planning problem. The *planner* core, named the planning algorithm, is the algorithm that actually computes a solution *plan* for any instance of the planning problem. So, the *planner* needs to be able to interpret the first three languages, to understand its input, and to write the computed solution *plan* in the latter. Usually the definition of the three languages used to specify the *planner*'s input are based on that of the *state space* (the language used to specify a *state* of the *system*).

The representation of the state of the *system* is in the core of domain independent planning, it is necessary for the definition of the current state, the goal, the domain and the actions (we need to express how the actions change the state). The previous paragraphs underline how essential is to have a common established methodology for the representation of the state of the *system*, i.e. of that of all its units. We need this representation methodology in order to incorporate inside the domain independent planner the representation of the state in a truly domain independent way. Otherwise, it is impossible to develop a really domain independent planner because we will need to adapt the planner in order to make it understand the particular ad-hoc form of representing the state of the *system*.

Furthermore, we need also to have an established methodology for the representations of the observations about the current state. We need this in order to process in a domain independent way the observations about the current state and transform them into proper constrains on the representation of the *system* current state. Again, without this, we will need always some ad-hoc processing of the information that comes for the sensing in order to translate them inside the planner.

5 Towards a Pro-Planning Architecture for Space Systems

Now we present a general architectural design for space systems that allows the designer to avoid the problems described in the previous sections. This methodology establishes a unified approach for the representation of the state of the system and for the acquisition of the observations about this state.

The proposed methodology exclusively dictates certain rules about the design of the interfaces of the units of the system. It do not alter the current well established space engineering practices of follow a hierarchical functional decomposition of the system (both FS and GS) that terminates with a lowest layer described as a group of units interconnected between them. It only dictates how these units interfaces shall be defined. As before, for simplicity, we assume that the space mission is composed by N units identified as U_i, with $i = 1, \ldots, N$, and proceed as if the whole space mission is given by the aggregation of these units. Again, and for the same reasons, we will ignore completely the existence of all other interfaces but those for M&C, and refer them simply as the unit interfaces dropping the M&C keyword.

In this common architecture, the interface of a unit U_i, denoted by $I(U_i)$, is specified by an ordered pair:

$$I(U_i) \equiv (\mathbb{V}_i^{\mathrm{ro}}, \mathbb{V}_i^{\mathrm{rw}})$$

with:

- $\mathbb{V}_i^{\mathrm{ro}} \subset \mathcal{V}$, a set of read-only variables identifiers
- $\mathbb{V}_i^{\mathrm{rw}} \subset \mathcal{V}$, a set of read-write variables identifiers

Both sets are subsets of $\mathcal{V}$, the set of all accepted variable identifiers. The complete set of variables that define the interface of the unit is denoted by $\mathbb{V}_i$, i.e. $\mathbb{V}_i \equiv \mathbb{V}_i^{\mathrm{ro}} \cup \mathbb{V}_i^{\mathrm{rw}}$.

The control of the unit is exclusively done through the setting of the values of the $\mathbb{V}^{\mathrm{rw}}$ variables. It is enough to read the values of all the variables in $\mathbb{V}_i$ to determine the state of the unit U_i. The values of the $\mathbb{V}^{\mathrm{ro}}$ variables offer all that can be observed about the unit change of state by exogenous or internal events (i.e. changes that happens independently of the values of the unit control read-write variables $\mathbb{V}^{\mathrm{rw}}$). We will formalize this in the following paragraphs.

As usual, we require these sets of variable identifiers to be mutually disjoint in order to avoid name clashes[5]. This property simplifies the definition of the the sets of all variables in the interfaces of the units:

$$\mathbb{V} = \mathbb{V}^{\mathrm{rw}} \cup \mathbb{V}^{\mathrm{ro}}$$

with:

$$\mathbb{V}^{\mathrm{rw}} = \bigcup_{i=1,\ldots,N} \mathbb{V}_i^{\mathrm{rw}} \qquad \mathbb{V}^{\mathrm{ro}} = \bigcup_{i=1,\ldots,N} \mathbb{V}_i^{\mathrm{ro}}$$

As in Section 2, these variables can be of distinct basic types: integer or real numbers, strings, time points values, bytes, boolean, etc. or they can be restrictions of these basic types on some subset of values. Hence, in general, for each variable $v \in \mathbb{V}$ we consider that the corresponding set with the values that can be assigned to it is given. We name this set the domain of the variable v and denote it by $\mathcal{D}_v$.

The State Space for the whole system is defined by:

$$\mathcal{S} = \prod_{v \in \mathbb{V}} \mathcal{D}_v$$

Its dimension is given by the total number of variables in the unit interfaces of the system. We are interested in the

[5]We require: $\neg\exists i, j \in \{1, \ldots, N\} \,.\, i \neq j \wedge (\mathbb{V}_i^{\mathrm{rw}} \cap \mathbb{V}_j^{\mathrm{rw}} \neq 0)$, $\neg\exists i, j \in \{1, \ldots, N\} \,.\, i \neq j \wedge (\mathbb{V}_i^{\mathrm{ro}} \cap \mathbb{V}_j^{\mathrm{ro}} \neq 0)$, and finally $\neg\exists i, j \in \{1, \ldots, N\} \,.\, i \neq j \wedge (\mathbb{V}_i^{\mathrm{rw}} \cap \mathbb{V}_j^{\mathrm{ro}} \neq 0)$.

finite case, so we have: $\dim(\mathcal{S}) = \bmod \mathbb{V} = m$, and we can assume $\mathbb{V} = \{v_1, \ldots, v_m\}$. The state of the system will be determined by the current value of these m variables. We denote val the function that returns the current value of any variable $v \in \mathbb{V}$, i.e.:

$$\text{val} : \mathbb{V} \to \mathcal{D} \qquad \text{with} : \mathcal{D} = \bigcup_{v \in \mathbb{V}} \mathcal{D}_v$$

Using this function it is easy to define formally the state of the whole system in the following way. If $\text{val}(v) \in \mathcal{D}_v \subset \mathcal{D}$ returns the current value of v, we have that the state of the system is the tuple of all the values of the variables in the system:

$$\vec{s} = (\text{val}(v_1), \ldots, \text{val}(v_m)) \in \mathcal{S}$$

We are describing the problem of planning with incomplete information and sensing, where $\mathbb{V}^{\text{ro}} \neq \emptyset$, because this is the standard situation when planning for space missions, we have both exogenous environmental events and internal events that change the state of the various units. If $\mathbb{V}^{\text{ro}} = \emptyset$ we have the particular case of planning with complete information, when there is not need to query the state of the system more than to know which is the initial state.

The following list describes the set of rules to follow when designing the interface of each unit U_i in the system:

1. The complete control of the unit functions shall be implemented through the assignment of appropriate values to the variables in the $\mathbb{V}_i^{rw}$ set.
2. The unit shall not modify values of the variables in the $\mathbb{V}_i^{rw}$. These values shall be set by the system monitor and control, and shall not be change by the unit later.
3. The reading of the values of the variables in the $\mathbb{V}_i^{ro}$ set, together with the values of the variables in the $\mathbb{V}_i^{rw}$ set, shall be enough to determine the internal state of the unit.
4. Only the unit can modify the values of the variables in the $\mathbb{V}_i^{ro}$ set.
5. It shall be stated if there are or not random exogenous or internal events that might change the values of the variables in the $\mathbb{V}_i^{ro}$ set.
6. A partial model of the environment behaviour should be provided, when some parts of it follows a predictable behaviour (e.g. sun/eclipse patterns on a orbit) that produce regular exogenous events that change the values of the variables in the $\mathbb{V}_i^{ro}$ set.
7. A partial model of the unit internal logic should be provided, when this logic follows a predictable behaviour that produce regular internal events that change the values of the variables in the $\mathbb{V}_i^{ro}$ set.
8. These models should be stated in the same domain independent language used to describe the planning domain.
9. When the unit is a consumable resource, it should be included in its interface a variable that can be set in order to mark which is the activity that is consuming it.

The first four rules complement the information hiding principle in order to keep track, and impose on the various designers of the system's units a common method that guarantee that will be possible to query the state of the system (simply by querying the values of the variables in the $\mathbb{V}_i^{ro}$ set) and to represent the state of the system, i.e. $\vec{s}$. The next four rules establish a general criteria for the treatment of the exogenous and internal events that can change the status of the units. They separate the case of random behaviour (e.g. failure of an equipment) from that of predictable behaviour. They require for the latter, models that should be incorporated into the planning domain definition and possible used by the domain independent planner for the generation of plans (e.g. the planner needs to take into account the sun/eclipse patterns on a orbit because this will appear as the precondition of some actions and in the goals).

The last rule is useful only for the automation of the conflicts between partial configurations of the CGSS as we explain now. An additional advantage of this approach is that it is possible to automate the conflict between configurations. It is easy to define partial configurations. A partial configuration of a Unit U_i is defined as any set of ordered pairs $(v, d) \in \mathbb{V}_i \times \mathcal{D}_i$. We can add this units partial configurations into partial, but composed, configuration of the whole CGSS. A partial configuration of the CGSS is simply any union of any set of unit partial configuration.

The technique to automate the detection of conflicts between any pair of partial configurations of the CGSS by analyzing the values of the variables that appear in both partial configurations. If there is any different value we have a conflict. Otherwise, both configuration are compatible and can be applied to the CGSS.

This completes the description of the proposed engineering methodology that overcomes the problems described in the previous sections.

6 CONAE Ground Station Services

This section briefly describes an on going effort of a real life application of the common architectural methodology described in the previous section.

We present the architecture design concept of: the CONAE Ground Segment (CGS); the CONAE Ground Stations Service (CGSS); and that of the Ground Station Operations Center (GSOC), the CGSS's subsystem in charge of the execution of the plan and all M&C activities.

The CGSS manages three antennas: one X-Band down link, S-Band up+down link 13 meters antenna; one X-Band down link 7,3meters antenna; and one S-Band up+down link 4,3meters antenna. A fourth 10 meters antenna is in the process of being integrated and a fifth is in the planning phase.

The CGSS download data from several satellites (AQUA, Landsat 5 and 7, Terra, RADARSAT, EROS-A1, etc.) and operates as the main station for the SAC-C Argentine mission, both for TT&C and science data downloading. Starting

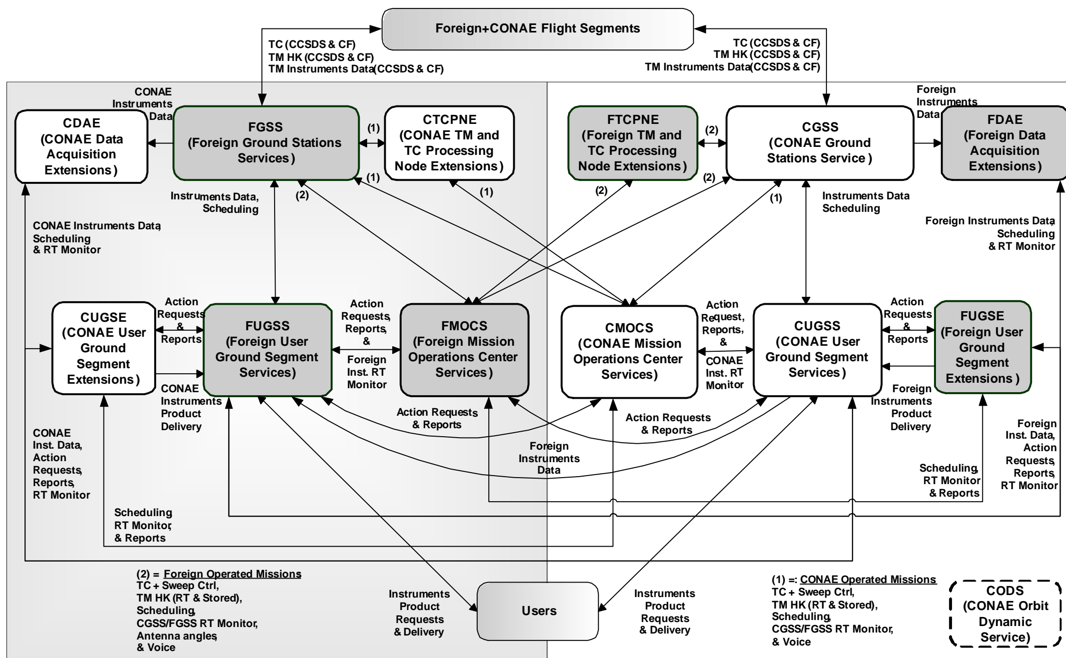

Figure 4: CONAE Ground Station concept

in a few months, it will serve the SIASGE (Italian-Argentine Satellite System for Emergency Management) mission, a constellation of four Italian SAR-X satellites and two Argentine SAR-L satellites. CGSS will operate as a support station for the Italian satellites (both for TT&C and SAR-X data downloading) and as the main station for the Argentine satellites. In a few years it will be also the main station for the Argentine/NASA SAC-D/Aquarius mission.

Currently, the general architectural methodology presented in the previous section has been applied to a subset of the whole system including the 13 meter antenna. The project is successfully finishing the testing campaign, it is at two weeks of the Acceptance Review. The incorporation of the rest of the CGSS into the pro-planning architectural approach is projected for next year.

CGS Design Concept. Figure 4 shows the top level design concept of the CGS. It is a functional diagram of considerable level of abstraction and do not represent the actual architecture of the CGS. The purpose of this functional diagram is to present the main functional components of the CGS and those of CONAE's foreign partners. In the figure, these components are represented by the boxes. The white boxes represent CONAE's components. The grey boxes represent all foreign components. The components depicted over the light grey background are located on foreign facilities. The components located over the white background are located on CONAE facilities. The box representing the flight segment is depicted in both colors due to the typical cooperative ownership of this segment. This diagram allows us to clearly identify the main interfaces between the components of the CGS and those of the Foreign Ground Segment. These interfaces are depicted with arrows. The CONAE Orbit Dynamic Service (CODS) has a dotted border meaning that all ground segment components might interface with it (both the foreign and CONAE components).

We briefly describe the CONAE's ground segment components:

- The CONAE Ground Stations Service (CGSS) groups all CONAE ground stations both for TT&C (S-Band) and for science data downloading (X-Band). Basically it consists of all CONAE's antennas (both for S and X-Band), its accompanying RF equipment, Data Acquisition equipment, and the Monitor and Control system.
- The CONAE User Ground Segment Services (CUGSS) are the services for both Argentine and foreign users interested in products generated by CONAE, independently of being they products from its own instruments or not.
- The CONAE Telemetry (TM) and Telecommand (TC) Processing Node Extensions (CTCPNE) allows CONAE to receive TM and to send TC in a foreign location from and to CONAE's missions. This equipment is only necessary when the facilities at the FGSS do not support the TM and TC formats of CONAE's missions (e.g. CONAE equipment at INPE's facilities at Alcantara, Brazil, for the SAC-C mission).
- The CONAE Data Acquisition Extensions (CDAE) allows a foreign partner to acquire some of CONAE's in-

struments data at a foreign location using the FGSS.

- The CONAE User Ground Segment Extensions (CUGSE) allows a foreign partner to generate some of CONAE's instruments products at a foreign location.
- The CODS include orbit determination, orbit propagation, two line elements generation and distribution, ground station contacts generation, maneuver computation, etc. for both Argentine and foreign, services and users.

CONAE Ground Stations Service Design Concept. Figure 5 is a functional diagram that shows the top level design concept of the CGSS. It presents the main subsystems of the CGSS. As before, the white boxes represent CONAE's components and the grey boxes represent all foreign components. This diagram not only allows us to clearly identify the main interfaces between the CGSS subsystems and the other components of the CGS and the Foreign Ground Segment with more detail, but also, it shows the internal functional subsystems of the CGSS.

Again, the figure is a functional diagram of a high level of abstraction that does not represent the actual architecture of the CGSS.

Now we briefly describe the CGSS subsystems:

- The GSP (Ground Stations Planning) subsystem receives action requests and generates the plans for the operation of the whole CGSS.
- The GSOC (Ground Stations Operations Center) subsystem executes the plans generated by the GSP. The execution of these plans embraces both the control of the units and the monitor of some variables in order to choose between various path. The CGSS also monitor the units in order to detect failures and change execute the corresponding contingency plan.
- The ANT subsystem includes all CONAE S/X-Bands antennas, as well as the Antenna Control Units, High Power Amplifiers, etc.
- The RF S/X-Band Subsystem includes all the RF equipment for each antenna, both the equipment used for tracking and data reception.
- The DA (Data Acquisition) subsystem consists of all acquisition systems.
- The TCPN (Telemetry & Commands Processing Node) subsystem includes any system used for the synchronization of the Telemetry and the encoding of the commands in real time.
- The NET (Network) subsystem provides the connection between all subsystems and with other services.
- The PE subsystem is the Power and Environment subsystem.

We avoid to depict the TCP/IP interfaces and the power interfaces for the sake of simplicity.

GSOC Architectural Design Concept. Figure 6 shows the various units of the GSOC subsystem and how they are interconnected, and connected with other subsystems of the CGSS and other services of the CGS and of the Foreign Ground Segment.

Now we briefly describe the several units that compose the GSOC subsystem:

- The CGSS Plan Execution Controller unit executes the plans send to it by the GSP subsystem.
- The CGSS Plan Execution Console unit (GUI) allow the monitoring of the CGSS Plan Execution Controller execution of the plans.
- The CGSS Configuration Manager (CfgMng) unit. This unit allow defining new configurations for commanding of the CGSS.
- The CGSS RT TM Mng unit distributes the real time TM that comes from the various CGSS units.
- The CGSS Stored TM Mng unit consolidates all the stored TM in the CGSS stored TM database.
- The CGSS TM Analyzer unit computes variables of higher temporal granularity and various standard queries using the consolidated CGSS stored TM.
- The CGSS Stored TM Mng Console unit (GUI) allows to display various queries of consolidated CGSS stored TM.
- The CGSS Product Mng unit generates all the data files needed by the various subsystems. It uses both, the Stored TM variables and those computed by the TM Analyzer.
- The CGSS TMView unit (GUI) displays the GSCC RT TM variables.
- The CGSS Alarm Handler unit associates contingency plans and informative actions with any alarm.
- The CGSS Alarm View unit (GUI) displays the CGSS RT TM alarms. It is the GUI of the CGSS Alarm Handler.
- The CGSS LogbookSrv unit (Logbook server) distributes the digital logbook entries.
- The CGSS LogbookClt unit (Logbook client GUI) displays the digital logbook entries.

All exogenous or internal random events (usually failures) are managed by the Alarm Handler in order of keep the whole system as safe and operative as possible. This is managed by having a failure tree with the associated contingency plans, with the best responses to each failure in the tree.

All exogenous or internal periodic/predictable events (usually the normal behaviour of the units) are managed by the GSP subsystem and incorporated directly inside the generated plans.

The pro-planning architecture approach in practice. The architecture of all the subsystems of the CGSS is based on the architecture presented in the previous section.

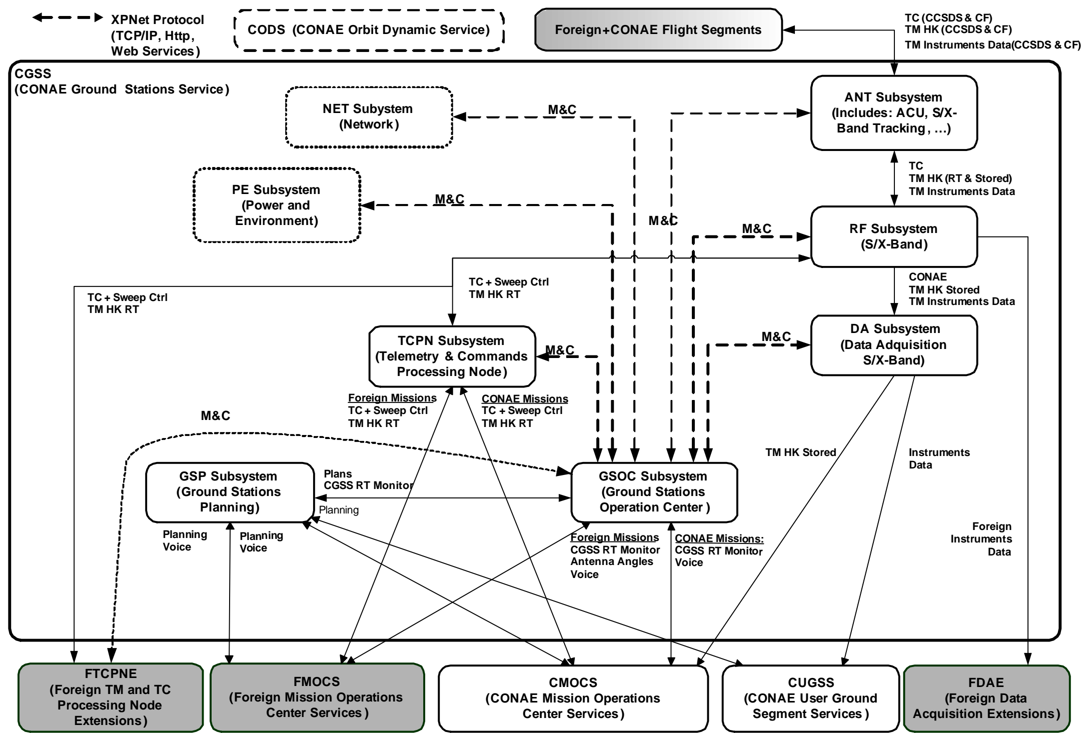

Figure 5: CONAE Ground Stations Service design concept

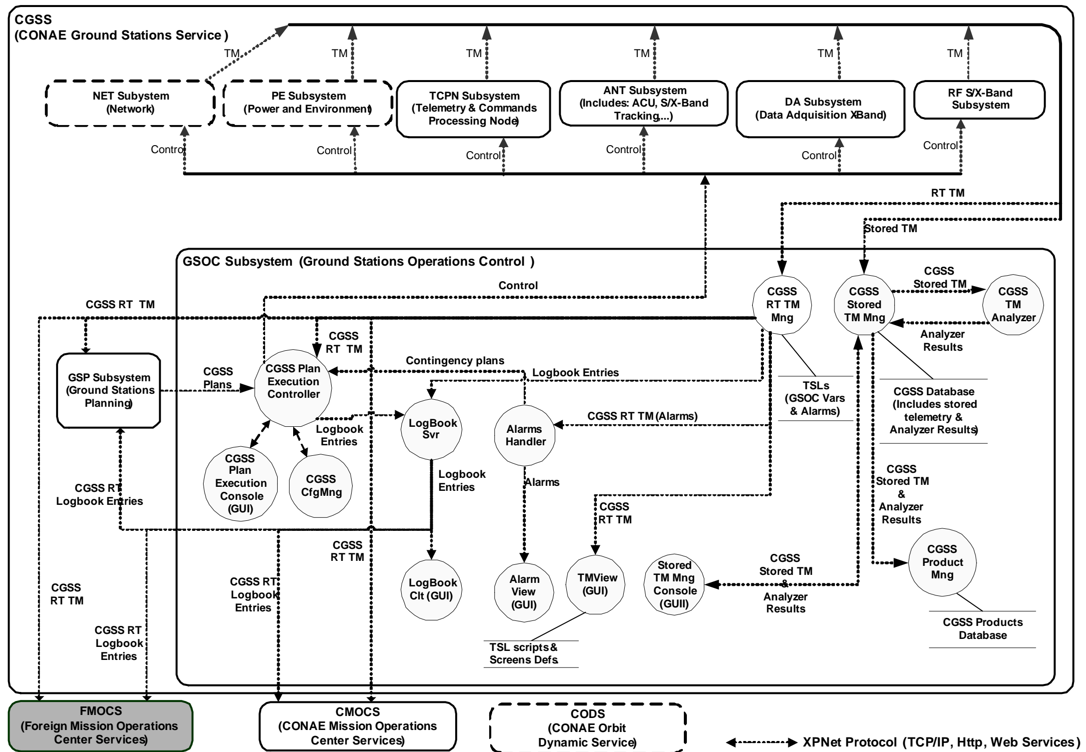

Figure 6: CONAE Ground Stations Operations Center Design Concept

As was explained in the previous section, the proposed methodology simplify and systematize the M&C architecture. The control is carried out by setting the read-write variables in the unit interfaces. All the monitoring is done by reading the read-only variables in the unit interfaces. The state of the CGSS comes out clear as the value of all read-write and read-only variables in the unit interfaces.

This made the whole project easier to project, design, develop, integrate and test. The complete list of equipment goes around one hundred hardware units and seventy software units. The whole set of variables in the unit interfaces goes around one thousand.

The following list summarizes the main hardware components that are currently functioning under this approach in the CGSS: 1 Antenna Control Unit, 3 Redundant Switch High Power Amplifiers, 1 Test Loop Translator, 2 X-Band Down-converters, 2 S-Band Redundant Switch Down-converters, 2 S-Band Redundant Switch Up-converters, 8 Receivers, 2 Demodulators, 2 RF IF Matrix, 1 ECL Matrix, 1 Weather Monitor Station, 2 CORTEX Base Band units, 2 Station Computer units, etc.

Currently, the CGSS's planning subsystem (the GSP) is based in an ad-hoc planner developed for the previous architecture. We decided to first validate the whole new architecture using the old ad-hoc planner during a certain period. This will approach will provide us with statistics about the new architecture compared with the previous one, using the same planning tool. In a first step, we are upgrading the ad-hoc planner with the capability of manage partial configurations and with the automation of the detection of conflicts between them. We are planning to start during the rest of this year the design phase of a new domain-independent planner based on the technology described in (Cesta *et al.* 2004).

7 Conclusion

We confront the standard architectural design approach for space missions with current AI domain independent planning modeling of a system in order to detect what turns the planning problem for space missions not tractable with the a domain independent planning approach. The main issues are: 1) The lack of a unified methodology that determines how the state of each unit and that of the whole system can be known and represented; and 2) That there is not an established criteria to manage the possible sources of change of the state of a unit.

We presented a simple general architectural design based on the exclusive use of read-write and read-only variables for the formal definition of the unit interfaces for their monitor and control, and on eight supplementary design rules. This approach allows us to overcome the previously highlighted problems, and as a byproduct, it allow us to automate the detection of conflicts between partial configurations of a system.

Finally, we briefly presented an application based on this approach: the CONAE Ground Stations Service, the service that manage all the CONAE's ground stations.

References

M. Abadi and L. Cardelli. *A theory of objects*. Springer, 1996.

D.G. Boden and W.J. Larson, editors. *Cost-Effective Space Mission Operations*. McGraw-Hill,Inc, 1996. Space Technology Series.

A. Cesta, S. Fratini, and A. Oddi. Planning with concurrency, time and resources: A CSP-Based approach. In I. Vlahavas and D. Vrakas, editors, *Intelligent Techniques for Planning*, chapter 8, pages 259–295. Idea Group Publishing, 2004. To appear.

R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence Journal*, 2(3-4):189–208, 1971.

P. Fortescue, J. Stark, and G. Swinerd, editors. *Spacecraft Systems Engineering*. J.Wiley & Sons Ltd., 2003. Third Edition.

M. Fox and D. Long. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003. Special issue on 3rd International Planning Competition.

P. Laborie. Algorithms for propagating resource constrains in AI planning and scheduling: Existing approaches and new results. *European Conference on Planning, Toledo, Spain*, 2001.

W.J. Larson and J.R. Wertz, editors. *Space Mission Analysis and Design*. Microcosm Press and Kluwer Academic Publishers, 1999. Third Edition.

J. McCarthy. Situations, actions and causal laws. In M. Minsky, editor, *Stanford Artificial Intelligence Project: Memo 2, 1963. Reprinted in Semantic Information Processing*, pages 410–417. MIT Press, Cambridge, Mas., 1968.

B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

N. Muscettola. Computing the envelope for stepwise-constant resource allocations. In *Principles and Practice of Constraint Programming, Eight International Conference, CP 2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2002.

R.P.A. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, pages 212–221. AAAI Press/The MIT Press, 2002.