Artificial Intelligence Planning and Scheduling Aboard INPE’s Satellites

F. N. Kucinskis¹, M. G. V. Ferreira²
¹ Aerospace Electronics Division – DEA
² Satellite Control Center – CCS
Brazilian National Institute for Space Research – INPE
C. Postal 515 – 12245-970 – São José dos Campos - SP
BRAZIL
E-mail: fabricio@dea.inpe.br, mauricio@ccs.inpe.br

Abstract

The experiments aboard the Brazilian scientific satellites are currently thought to execute in a repetitive way, collecting, storing and sending data in a cycle that does not suffer great alterations. There are, however, scientific phenomena of random occurrence which demand a fast reconfiguration of the system to collect and process data adequately. Due to the short duration of these events and the great number of states in which the satellite subsystems can be at the moment of the detection, the use of classical programming techniques or the ground team intervention is not enough, and the opportunity to analyse the phenomenon is lost. RASSO, a Resources Allocation Service for Scientific Opportunities, makes use of Artificial Intelligence Planning & Scheduling techniques to modify onboard the current plan of operations and allow an experiment to use more resources when a scientific event of this kind occurs, thus improving the scientific return. This paper describes the RASSO architecture and compares some of its characteristics to the ones present in other spacecraft onboard planners.

Keywords: artificial intelligence, planning, scheduling, autonomy, satellite.

1. INTRODUCTION

The satellites of the Brazilian Program for Scientific Satellites and Experiments run by INPE carry as payload a set of scientific and technological experiments, which have pre-defined quotas of resources (mass memory, power, communication channels, etc) allocated by the onboard computer as the scientists and mission engineers had decided it, still in the phase of systems specification. As a result, the experiments collect data through operation plans that generally follow a repetitive pattern.

This way of dealing with the experiments operation fits perfectly to long-term scientific observation. There are, however, short-duration scientific phenomena of which occurrence, although predictable, are random – an ionospheric disturbance, for example, can happen at any time and last from minutes to hours. To better analyze these phenomena it may be important to increase the acquisition rate and / or the precision of the data collected by an experiment. This increases the consumption of memory and power beyond the originally predicted. Due to the short duration and the difficulty to specify exactly when a phenomenon of this kind will occur, it is not enough to leave the ground operations team in charge of the system reconfiguration. The necessary time for the phenomenon to be reported and for the ground team to send a new operation plan to the satellite is in general much longer than the duration of the phenomenon, and in this case the scientific opportunity to adequately analyze it will have been lost.

There is then the need for allowing the experiment, when detecting the occurrence of a short-duration phenomenon, to request from the onboard computer the temporary reallocation of resources to be able to carry out a more detailed analysis, affecting the least possible the operation of the other experiments and the satellite itself. As the number of states in which the several satellite subsystems and experiments is huge, it becomes difficult the use of classical programming techniques to handle it. In this context the Artificial Intelligence (AI) Planning and Scheduling techniques is presented as a potential solution to be explored.

RASSO, a Resources Allocation Service for Scientific Opportunities (Kucinskis & Ferreira, 2006), is being
developed at INPE, and uses AI Planning and Scheduling to allow a scientific satellite to temporarily modify its current operation plan in order to better analyze short-duration phenomena.

2. THE EQUARS SCIENTIFIC SATELLITE AND THE RASSO SERVICE

INPE has a Program for Scientific Satellites and Experiments (SCE, in the Portuguese acronym), of which were part the Scientific Application Satellites (SACI 1 and 2) and the French-Brazilian Microsatellite (FBM). The goal of these missions is to serve as test platforms of new technologies and equipment, as well as providing frequent and low-cost access to space to the Brazilian scientific community. The next satellite in this series is EQUARS. It is a project in partnership with research institutions from many countries. The embarked experiments are proceeding from institutions from Brazil, United States, Canada and Japan, and have as goal to make a global scale monitoring of the Earth’s equatorial low, middle and upper atmosphere and ionosphere, with a special emphasis in dynamical and photochemical energy transport processes. Being the next satellite in the INPE’s scientific satellites program, EQUARS is the natural case of study for the application of onboard planning to respond to the occurrence of short-duration phenomena.

In addition to the satellites programs – data collecting, scientific and remote sensing –, INPE maintains many technological development projects, managed by different teams spread over the Institute. Among these projects is the COMAV (from the Portuguese acronym for “Advanced Computer”), which is being developed by the Onboard Data Handling Group of the Aerospace Electronics Division (DEA). COMAV is a project for the development of a new onboard computer for future INPE’s satellites. Among the COMAV’s attributions is the operation control and communication with scientific and technological experiments.

The COMAV’s software is being developed in the C language over the RTEMS operating system, distributed under the GNU General Public License terms. The onboard replanning has been being implemented as part of a service that provides more resources than originally programmed for experiments that detect the occurrence of scientific short-duration phenomena. To this service was given the name “Resources Allocation Service for Scientific Opportunities”, or just RASSO. Figure 1 shows RASSO’s architecture. The arrows indicate the data flow between the modules during the planning process. Follows a brief description of the service functioning.

![Figure 1 - The RASSO Architecture](image)

An experiment that detects the occurrence of a short-duration phenomenon sends a request for more resources to RASSO, to make a better observation of the phenomenon (arrow number 1, to the left). When receiving the request, RASSO composes a well-defined problem in the form of a draft operation plan. This draft plan is created
consolidating information from several sources inside the onboard software (arrows 2 to 5), and then is directed to the Planner module (arrow 6), responsible for working out the conflicts that were inserted in the operation plan because of the request from the experiment, respecting a set of constraints and goals that were imposed to it. When succeeding in creating a new operation plan that takes care of all these requirements, RASSO sends it to the TTTC queue, turning it in the new plan of the satellite’s experiments (arrow 7). The next chapters detail the components, concepts and the dynamics of the service.

3. THE SATELLITE MODEL

AI Planning and Scheduling are generally associated to a huge processing volume and memory consumption. This does not present a great problem when working with PC computers capable of run with clock frequencies in the order of Gigahertz and have several hundreds of Megabytes of RAM memory. This, however, is exactly the opposite of what is found in embedded systems. COMAV is based on an ERC32 processor (RISC, SPARC 32 bits architecture) running at 12MHz, and have 2Mb of program memory. The computational power limitation and the quickness necessary to reconfigure the satellite in response to a phenomenon force us to take a special care with the planner performance – the total replanning time has to be kept in the order of some few minutes. In these conditions, the cost to analyze runtime a model described in a language such as PDDL (Edelkamp & Hoffmann, 2004) and transfer its elements to adequate data structures in memory starts to be considerable. To deal with these limitations we decided that the satellite model to be used by the planner should be described in the proper C language. The model elements would be stored directly in the data structures in which they would be worked, preventing the model’s description analysis stage.

The C language, if used in the right way, can describe a model adequately. However, a model described in C would lose in clarity. An engineer or scientist not familiar with the programming language and with the software structure would not understand what is being represented. This way, to allow the direct storage in the right data structures and still keeping the model readable, we decided to create a model description language over C, through the use of macros, hiding behind these macros all of the structures, pointers and function calls used by the planner. The use of macros to implement this language – that we called “RASSO_ml” – makes the task of convert a model instruction to data structures being a responsibility of the GCC compiler’s pre-processor. With this, the model elements are always ready to be used by the planner, eliminating a great part of the initialization process.

3.1. The Model Static Description

The static part of the model is composed basically by objects and resources. Objects are elements instantiated from classes defined by the modeler, which have attributes whose values are changed by the commands execution or the occurrence of exogenous events. The set of attribute values of an object at a specific moment in time is generally called “object state”. Resources are the consumable elements of the model.

There are three kinds of resources in RASSO_ml: exclusive, depletable and reservable. The exclusive resources do not have a quantity (they are unique) and can be used by only one object at a time. The depletable and reservable ones have a minimum and maximum quantities defined in its creation, and these quantities cannot be exceeded in any moment. The difference between them is that, while the depletable resources “accumulate” its consumption in time until being completely depleted, the reservable ones are controlled in a momentary way, without accumulation. Examples of depletable resources are fuel and memory; power, on its turn, is a reservable resource. Resources are controlled in “units”, no mattering to the planner if they are watts, kilos or bytes. Figure 2 shows a simplified RASSO_ml code snippet with instructions to the planner for the creation of a domain, a class, objects and resources.

```c
create_domain(Exp_Name, ionex, grom, certo, tip, mltm);
create_class(Experiment, 5,
    Exp_Name name;
    Boolean on;
    int sample_rate;
```
3.2. Dealing with Time

An operation plan is a set of actions (satellite’s internal commands) that affect the satellite state as the time goes by. So that to the planner be successful in dealing with the changes of the modeled satellite, it has to manage not only the objects and resources, but all of the states they assume over the plan, that is, all of its “moments”. Thus, whenever an object is created, it is being created, in fact, a timeline for the object. This timeline stores all of the states an object assumes during the plan period. An object is not directly manipulated in RASSO_ml; it is necessary to declare in which “moment” of the object one wants to work. Figure 3 shows some of the ways of dealing with objects in time.

```c
int  priority;
);
create_object(IONEX,   Experiment);
create_object(MLTM,    Experiment);
create_resource(Power, reservable, 0, 60);
create_resource(Mass_Memory, depletable, 0, 128);
```

**Figure 2 - Creating Classes, Objects and Resources**

```c
current_state(TIP).on = false;
if (initial_state(TIP).on == goal_state(TIP).on) return(success);
```

**Figure 3 - Working with Timelines**

In a similar way of what happens with the objects, when creating a resource, a resource consumption profile is generated. This profile controls how much of each resource is being consumed or generated at each moment of the time, and is used to detect resource overuse.

3.3. The Model Dynamic Description

The main elements of the dynamic part of the model are the actions and the behaviors. An action corresponds to one or more internal satellite commands. When composing a problem, RASSO converts commands extracted from the TTTC queue in its corresponding actions to generate an initial draft plan. When the planning process is finished the planner sends to the TTTC queue the commands corresponding to the new plan, overlapping the old one (this will be explained in more detail in the next chapters). The action is implemented as a C language function and describes how the model is affected by its execution. Figure 4 shows a simple action, which is described as follows.

```c
RASSO_Action Turn_On (with_parameters)
{
    parameter(exp); // experiment

    when_planning
    {
        // the gps cannot be turned on / off !!!
        condition(current_state(exp).name != grom);
        condition(current_state(exp).on == false);

        // effects described here
    }
```

**Figure 4 - A Simple Action**
current_state(exp).on = true;

switch(exp.name)
  case ionex:
    { 
      consumes(exp, Power, 7 per_hour);
      consumes(exp, Mass_Memory, 1.5 per_min);
      break;
    }
  case mltm:
    { 
      consumes(exp, Power, 4 per_hour);
      consumes(exp, Mass_Memory, 0.8 per_min);
      break;
    }
  }
when_running
    
      Send_to_TTTC_Queue(Turn_On_Exp(exp));
    
action_success;

Figure 4 - An Action in RASSO_ml

The “Turn_On” action receives as parameter an object of the type “Experiment” (the action parameters, its types and range of possible values are informed to the planner in an apart routine, where the structures to control the actions are initialized). Each action is called by the planner at two different moments in the planning process. The blocks “when_planning” and “when_running” indicate which code fragment must be executed in each one of these moments. The first moment (“when_planning”) happens when the planner is testing actions to apply in an incomplete plan, trying to achieve the goals and satisfy the constraints that were imposed to it. The second moment (“when_running”) happens when the plan was already gotten, and the planner is sending the commands corresponding to the actions to the TTTC queue. The function “Send_To_TTTC_Queue” called here is an internal satellite pseudo-command. Inside the “when_planning” block the pre-conditions necessary to the execution of this action are described and, in case of all of the informed conditions are true, which are the action effects over the model.

The “consumes” instruction deserves a more detailed analysis. It informs that, from the execution of the action on, determined object (in this case, the experiment passed as parameter) will consume a resource at the rate informed in the instruction – seven units of power per hour, for example. The basic RASSO time unit is the second. When informed of the consumption rate over time, the planner stores it in seconds. The macros per_sec (default), per_min, per_hour and per_day are just multipliers.

Actions allow describing the satellite behavior in response to determined commands executed by software. However, not all of the changes in the satellite state happen in function of software commands. Some of the EQUARS experiments have their functioning tied to the solar incidence, or the lack of it. The TIP experiment, for example, makes observations of lightning and sprites while it is in eclipse (at “night”, during an orbit), what is not possible when it is illuminated (at “day”). These experiments can be activated and deactivated automatically by solar sensors, without turn on / off commands scheduled for execution. Although the software is notified that the experiment was turned on / off, there is no way to predict this with antecedence, what is vital for the planning process. To deal with exogenous events such the ones described above, RASSO makes use of the concept of time windows.

Time windows are periods in which the satellite presents a determined typical behavior. For example, during the period in which the satellite is in contact with a ground control station, it transmits the experiments stored data, thus releasing the resource “memory”. It also consumes more power, since its communication system is active.
Figure 5 shows the creation of three time windows: “Day”, “Night” and “Communicating”. The second instruction parameter is a unique time window identifier, used by the planner.

```c
// time windows related to orbital phases
create_time_window(Day, 0);
create_time_window(Night, 1);
create_time_window(Communicating, 2);
```

**Figure 5 - Creating Time Windows**

Time windows can be consecutive, as it is the case of “Day” and “Night”, or they can overlap each other, as “Communicating” in relation to the other two – no restriction about this is made by RASSO. There is also no obligation concerning the occurrence of a time window in every orbit. “Communicating”, for example, will not occur in every orbit, depending on the orbital characteristics. RASSO has a Time Windows Table, which stores the start and end times of the occurrence of each time window for every orbit in the next days. These data are updated regularly by the ground operations team via telecommands. Using this information, it is possible to tie behaviors to the occurrence of the windows.

The behavior is part of the model dynamic description. While an action must have one or more commands related to it in the TTTC queue, the behavior describes activities that happen at the beginning and / or the end of a time window, independently of the current operation plan. These activities can be described directly in the behavior, through instructions for the manipulation of objects and resources, or through calls to actions – in this case, only the block “when_planning” of the action is executed. Behaviors, in contrast to actions, do not generate commands in the plan generated by RASSO. Figure 6 brings some behavior examples.

```c
RASSO_Behavior Day_Bhv (happens_at Day)
{
    at_start
    {
        // load battery when enlightened
        generates(Power, 0.5 per_min);

        // "don’t mess with MLTM memory!!!"
        keep_resource_untouched(MLTM, Mass_Memory);
    }

    at_end
    {
        generates(Power, 0); // stop loading battery
    }
}

RASSO_Behavior Night_Bhv (happens_at Night)
{
    at_start // sun sensor turns on experiments
    {
        Turn_On(IONEX);
        Turn_On(CERTO);

        // guarantee at least 4 Watts for half an hour to IONEX
        guarantee_resource(IONEX, Power, 4, 0, 1800);
    }

    at_end // sun sensor turns off experiments
```
The “happens_at” clause ties a behavior to a time window. The blocks “at_start” and “at_end” of each behavior indicate which activities are related to the beginning and end of the time window. It is not necessary to have both blocks declared, just that that will be used. In the above example, the instruction “generates(Power, 0.5 per_min)” indicates that the battery is being loaded at the rate of 0.5 units per minute while the satellite is enlightened. At the end of the time window, “generates(Power, 0)” informs that the loading is finished.

The instructions “keep_resource_untouched” and “guarantee_resource” can only be used in the “at_start” block of the behavior, and impose constraints to the planner. The first one prevents the planner to select, when searching for actions to compose a plan, any action that affects the resource consumption by an object (in the example, the memory consumption by the MLTM experiment). The second one imposes that a determined amount of resource must be guaranteed to an object during certain period inside the time window (in the example, it is imposed the maintenance of at least 4 units of power for the IONEX experiment, on the first 30 minutes of the “Night” window).

4. THE PROBLEM COMPOSER MODULE

COMAV will communicate with “intelligent” experiments, that is, experiments that are managed by their own processor and software. Each experiment that can detect short-duration phenomena shall have at least one Trigger Condition in its software that, when true, will send a request for more resources to RASSO, notifying which resources it needs, how much is necessary, from what moment and for how much time. This request for resources will be received by the Problem Composer module, which is responsible for congregating information from several sources (see Figure 1) and supply the Planner with a problem to be solved. The creation of a well-defined problem is as important as the process of searching for the solution. The problem consists of a draft operation plan with goals to be achieved, constraints to be respected and conflicts to be solved. The initial satellite state, the goal states, the telecommands (actions) scheduled for execution and the exogenous events (behaviors) that occur during the plan execution period are all part of the problem.

4.1. Planning Horizons

The request for resources sent by the experiment carry in its parameters the information of two crucial moments for the planning process: the beginning and end of the period in which the experiment needs more resources.
These key moments are called “planning horizons”, and they are used to determine the initial state, the intermediate goals and the final goal (see Figure 7).

![Figure 7 - Planning Horizons and Goals](image)

The beginning moment of the period with more resources allocated for the solicitant experiment is called “horizon h1”, or simply “h1”. The end moment of the period with more resources is “h2”. The horizons h1 and h2 indicate the moments in which the intermediate goal states are in the planning process.

The horizon h1 indicates the moment in which the requested amount of resources must be already reallocated for the experiment. H2 indicates until when these resources must be kept. In function of these, there are two more horizons: the initial horizon “h0” and the final horizon “h3”. H0 is determined as being h1 – ∆1, where ∆1 is the time necessary for the execution of reallocation commands for the solicitant experiment. In a similar way, h3 is determined as h2 + ∆2, where ∆2 is the time necessary to return the resources to the experiments that yielded them, thus placing the satellite in its “normal” operation mode.

It is necessary to define the meaning of “normal operation mode” here: as RASSO is proposed to temporarily modify the satellite’s operation mode, the plan generated shall guarantee that, when reaching the horizon h3, all of the modeled objects will be in the same state as they would if the original plan were executed, and the available amount of resources will be at least equal to that that would be left by the execution of the original plan.

5. THE PLANNER MODULE

Having a draft plan, the Planner module will cover it in chronological order, from the horizon h0 to h3, searching for conflicts to solve. Whenever a conflict is found, it will retrocede in the plan and try to change the actions in such a way to resolve it. Once the conflict is resolved, the planner returns to scroll the plan in direction to h3, searching for the next conflict to solve.

Conflicts can be of two kinds: violation in the resources consumption constraints or inconsistent goal states. A violation conflict in the resources consumption constraints is related to the resource overuse, consuming more than its available amount, or a quantity out of the range imposed by the Problem Composer. To resolve this kind of conflict, the planner has at its disposal the following options:

- Change the operation mode of experiments to make then consume less. To do this it is possible to try an action "Change_Sample_Rate", for example, to modify the data acquisition interval;
- Delay the execution of an instruction to turn on an experiment – since this instruction is an action (a time-tagged telecommand), and not a behavior (an exogenous event);
- Turn off an experiment for the least possible time.
An inconsistent state conflict is related to goal states imposed to the planner by the final goals in h3 or by instructions inside behaviors. When the execution of actions and behaviors does not lead to one of the imposed states, there is a conflict to solve. The planner tries then to find actions that take the model objects to the desired states. In both cases, heuristics guide the search process to choose the action that seems to lead the satellite to a state nearer the goal. However, the insertion, alteration or exclusion of actions in the plan can insert new conflicts, which are treated by the planner in the same way. This process continues until there are no more conflicts to solve. When the plan does not have more conflicts, it is sent to the TTTC queue, to become the new satellite’s operation plan, overlapping the old one.

6. RELATED WORK

RAX (Bernard et al., 1999) was the very first spacecraft onboard planner, having been executed in May of 1999. Aboard the DS-1 probe, it received high-level commands and generated, from this, detailed operation plans for the propulsion and thrust control, navigation and others. Few comparisons can be made between RAX and RASSO due to the difference in their objectives and implementation form – RAX was part of a more complex experiment, involving model-based fault protection and a plan execution agent capable of making deviations in the plan during its execution, based on the comparison between system values predicted in the model and real values verified in the execution moment. The plans generated by RASSO, however, are scheduled and executed so that there isn’t any kind of reasoning subsequent to the planning process.

In 2003 CASPER (Chien et al., 2005) started to operate as part of the Autonomous Sciencecraft Experiment (ASE) in the EO-1 remote sensing satellite. ASE uses algorithms to detect events of scientific interest, such as floods and volcanic eruptions, as well as verify the presence of clouds in the images taken. Based on this information, CASPER generates autonomously operation plans for the next orbits to get, for example, new images of a detected eruption.

RASSO has much in common with CASPER: both have as main goal the onboard replanning to propitiate the increase of the scientific return. While CASPER makes use of a “C-like” language syntax to describe its models, RASSO really uses C. The form of generation and resolution of problems also keeps some similarity between the planners – both have as input a draft plan with conflicts and apply the “iterative planning” to solve them.

There are, however, some basic differences. While CASPER models the elements in terms of “state variables” (the same approach adopted in RAX), RASSO instantiates objects from classes with sets of attributes. Although this is not object orientation in the strict sense of the term, it allows the use of more complex data structures in the modeling and the passing of objects as parameters to actions. While activities such as “Turn_On_TIP”, “Turn_On_GROM” would be necessary in CASPER, RASSO allows creating a unique action: “Turn_On(experiment)”.

But maybe the main difference between them is in the approaches of the CASPER activities and RASSO actions. An activity in CASPER has a start time, end time and a duration. An activity can consume one or more resources, and it is informed the total amount of resource consumed during the activity. In RASSO, the objects consume resources, not the actions – although there are also instructions to model the generation and consumption of resources without link to objects. The RASSO actions do not have duration – they are punctual.

The resource consumption is not informed in an absolute quantity, but in terms of a resource consumption rate by an object, in the time. Instead of informing that an activity “Operating_TIP” with duration of 60 minutes, for example, consumes 6Mb of memory, RASSO uses an action “Turn_On(TIP)” informing that, from this action on, the experiment TIP consumes 0.1Mb per minute. This is nearest to the real satellite operation than the activities description, being necessary even that another action informs to the planner when a resource stopped being consumed through a “Turn_Off(TIP)”, for example. Thus, if it is necessary to get 10Mb of memory from TIP, there is no need for removing the entire activity from the plan; it is enough to turn off the experiment 10 minutes earlier.

One of the most interesting CASPER features is the temporal relationship between activities to a partially ordered plan. RASSO does not have this sophistication level. The tasks ordering happen implicitly by action conditions, resource consumption constraints and goal states imposed by the Problem Composer.
7. FINAL REMARKS

RASSO is an effort to allow the use of onboard planning and scheduling in the INPE’s future space applications. The increase of autonomy provided by planning makes possible the reduction of the operation costs, the optimization of the results gotten from the satellite and opens way for new applications. The insertion of this technology, however, has to be made in a safe and gradual way, based on the gain of confidence of engineers, operators and mission managers.

The use of the C language pre-processor to analyze the model optimizes the processing time and brings the planner near to the rest of the satellite software, allowing that the onboard replanning could become just one more service provided by the onboard computer, as well as housekeeping, diagnosis and others.

REFERENCES


