A DOMAIN-INDEPENDENT PATTERN RECOGNITION SYSTEM TO SUPPORT SPACE MISSION PLANNING

Juan Manuel Delfa Victoria[1], Yang Gao[1], Nicola Policella[2], and Alessandro Donati[2]

[1]*Surrey Space Centre, University of Surrey, Guildford (UK)*
J.Delfa@surrey.ac.uk
Yang.Gao@surrey.ac.uk
[2]*European Space Agency, Darmstadt (Germany)*
Nicola.Policella@esa.int
Alessandro.Donati@esa.int

ABSTRACT

In the history of space exploration, the complexity of spacecrafts and missions have never stopped growing. This makes manual operations very challenging if not impossible. As a result, space agencies have started to move toward autonomous operation and mission planning systems for both on-ground and on-board scenarios. A planetary rover represents a typical complex mission, as the rover needs to directly interact with a dynamic environment that will introduce errors to any pre-designed plan while direct telecommanding from human operators is not possible due to communication delays, demanding more intelligence from on-board software. In future scenarios like a permanent base on the moon or a manned mission to Mars, robots and humans will need to collaborate closely using a *common natural language*. The operators should be in charge of defining high level orders to be executed by the robots, which will have to *coordinate* their actions in order to achieve all the goals with the best possible performance. The planner on-board each robot is the final responsible of the generation and maintenance of a valid plan, taking into consideration the set of goals assigned and the changing conditions of the robot and environment. The planner should as well be capable of generating its own high level goals in case of unexpected events.

This paper presents a planner, called QuijoteExpress, which takes into consideration these concepts. QuijoteExpress uses a mixture of theoretical planning solutions (like Hierarchical Task Networks or Temporal Planning) and soft-computing techniques (like Pattern-Matching or Fuzzy Logic). In particular the paper discusses the use of pattern-recognition algorithms within QuijoteExpress that can improve the planning performance as well as synthesize robust solutions.

1. INTRODUCTION

In 1997, NASA conducted its Mars Pathfinder mission with the Sojourner rover on-board. Above all, Sojourner demonstrated the need for more autonomy in future missions. Currently MER[1] and upcoming MSL[2] and ExoMars[3] missions represent a step forward in terms of autonomy, but they still lack “intelligence”.

There are several reasons why more sophisticated autonomous systems are desirable in space missions:

- No direct communication: Space missions use to suffer frequent eclipses as the spacecraft goes behind a celestial body. At the same time, communication delays for deep space missions make it impossible to operate the spacecraft from Earth. As an example, the round-trip of a signal to Mars takes between 6 and 40 minutes.
- Performance: In order to increase the science return, the time a spacecraft spends in idle mode waiting for instructions must be reduced. More autonomy on-board would provide the spacecraft with the intelligence to make its own decisions, therefore improving the performance.
- Cost: As missions are increasing in number and longevity, autonomy can play a key role to decrease costs. MER mission for example has been extended from its nominal 90 days to far more than 2000 and counting, with a cost around 1.2 Millions/month.
- Criticality: A lot of satellites have been lost because they had no FDIR system on-board or it was incorrectly designed. It is important to endow a satellite with autonomous capabilities to analyse its status and initiate autonomous recovery procedures in case of problems.

[1]http://marsrovers.jpl.nasa.gov/home/index.html
[2]http://marsprogram.jpl.nasa.gov/msl
[3]http://www.esa.int/esaMI/Aurora/SEM1NVZKQAD_0.html

In terms of planning, real life problems (in contraposition with synthetic problems) present some features that make them particularly hard to be solved:

- Partial knowledge of the world: Some characteristics of the world in our system might not be modelled or have an insufficient representation. This will affect the accuracy of the system estimation.
- Dynamic environments: The conditions of the environment might change with time, introducing uncertainty in the execution of the plan.
- Complex search spaces: Many real life problems deal with infinite search spaces. Therefore, it might not be possible to guarantee the soundness and completeness of the planner.
- Solution: Sometimes, it is difficult to specify what is a good solution or even what is a solution.

To generate robust plans is particularly important in the case of planetary rover missions. On the one hand, the surface of a planet represents a dynamic, unknown environment that introduce high levels of uncertainty in the previsions of resources consumption, execution time and even in the goal achievability, representing one of the main factors that cause a plan to fail. On the other hand, the complexity inherent to the systems and missions represent a challenge that might be solved with the help of autonomous systems.

As previously stated, one of the main concerns in planetary missions is the plan robustness. This paper presents an approximation to increase it using the knowledge acquired during previous plan executions. Once a plan is successfully executed, its context is extracted together with some information about the execution, and stored as a pattern. In case the planner is requested to produce a new plan, the knowledge database (KDB) is queried. If some plans stored in the database have a similar execution context, they are extracted and iteratively inserted in the present problem until a solution is found. Whenever the execution finishes, the plan (which might differ from the original one due to replanning) is again compared with the database. If no similar template is found, it is stored. In case there is a matching, the matched template rating is updated taking into account the execution performance of the plan.

A planner called QuijoteExpress, is under development using this technique. Its aim is twofold: to decrease the planning time, reduced to find templates in the KDB and fix the dependencies between templates; to increase the plan robustness as the system is gathering more information. QuijoteExpress is a problem-independent, timeline-based planner. Although it has been designed for on-ground space missions planning, it can be applied in any scenario where time and constraints satisfaction are key aspects of the problem. It is based on two fundamental pillars: (1) APSI Cesta and Fratini [1], Donati et al. [3], a framework oriented to the development of Artificial Intelligence Planning and Scheduling technologies for space missions and (2) the Hierarchical Timeline Networks theory (HTLN) intended to represent timelines as hierarchies of goals and constraints.

Following sections present a short introduction to APSI and two different concepts, hierarchy structures and machine learning, integrated in QuijoteExpress in order to address the problems of complexity and robustness respectively. Finally some conclusions are presented.

2. APSI REVIEW

QuijoteExpress extends the APSI framework to represent the elements of the planning system. Even though a deep analysis of APSI is out of the scope of the paper, it is important to understand the main concepts that APSI introduces, as they will be used along the whole paper.

APSI is composed of a set of plug-ins developed in JAVA based in the Simple Temporal Problems (STP) theory Dechter, Meiri, and Pear [2]. It is based on three principles: model-based design Fesq et al. [6], goal-based Dvorak et al. [4] and timeline planning.

A problem in APSI is decomposed in components over which the system has to accomplish the P&S. Examples of components in the rover scenario might be simple elements as the camera of the rover, complex as the locomotion system (composed of wheels, motors, etc), external elements as a rock in the surface, etc. The process of prototyping a planning application is essentially composed of the following steps:

1. Modelling (Components): The planning and scheduling problem is modelled by identifying a set of relevant features called *components*. APSI has 3 classes of components:
   - StateVariable: Described by a set of allowed values. Decisions assumed over a state variable are *value choices* in specific instants or intervals of time.
   - Reusable Resource: Described by its maximum capacity, which limits the maximum consumption of the resource. Decisions assumed are *activities* that represents the utilisation of certain amount of this resource in instants or intervals of time.
   - Consumable Resource: Described by the minimum and maximum capacity. The decisions imposed are *consumption* or *production*.

   The components can be described by means of the DDL3 (Domain Description Language). A component is described as a finite automaton (see 1) which represents the valid state transitions. Each component has a *timeline* associated which represents the

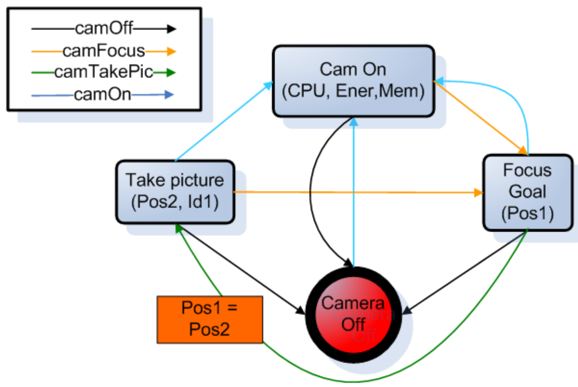

Figure 1. Automaton describing the state transitions of the camera on-board a rover

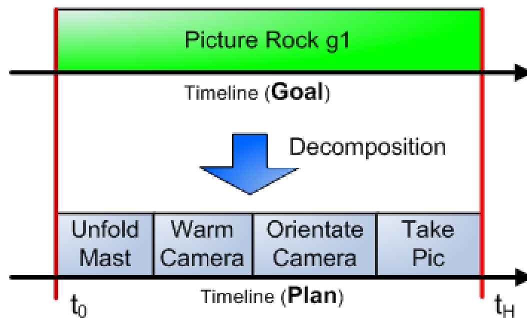

Figure 2. Timeline with the sequence of activities of the camera on-board to take a picture

evolution of the state of the component along the time, limited by a time horizon (see 2). Decisions are posted along the timeline of the components: *choices* over the set of values of the state variable or *consumption/production* activities on a resource.

2. Synchronizing components (Domain Theory): Once all the components are created, *synchronisations* between them are created. A synchronisation indicates the way in which the different components should collaborate. A component may require other components to be in a specific state in order to change its state.

3. Problem description: A problem description represents a specific instance of the domain with the initial state of the world (values of the components) and the definition of the goals as a set of values that some components must have in specific instants or periods of time. A problem is described as a partial *Decision Network* or DN. A DN is a graph composed of nodes and edges:

   - *Node*: Decision (valuechoice, activity, consumption or production) assumed for a component in some time instant/interval of its timeline.
   - *Edge*: Relation (parameter constraints, temporal constraints or value constraints) between two or more components.

In the solving process, the domain and problem represent the inputs for the planner and scheduler. The planner will be in charge of completing timelines that justify the initial goals. To achieve it, new decisions should be posted in the timelines of the different components until they have no gaps (time periods where the action to accomplish is still undecided). A decision is *consistent* if it fulfil the constraints defined in the internal model (component automaton) and the external model (synchronizations between components). If the timelines of all components of the problem present no gaps and all the decisions are consistent, then the plan represents a solution.

3. HIERARCHICAL TIMELINE NETWORKS - EXTENDING APSI THEORY

In the context of space missions, it is not sufficient to merely generate valid plans: experts need to understand the reasons leading the planner to generate a specific solution.
This complexity can be addressed using high level goals, that is, instructions easily understandable by human experts. A goal is then decomposed into a hierarchical structure of sub-goals until all of them are operators understandable by the executive system. This hierarchical structure is also used as learning units, as it is explained in next section.

3.1. Hierarchical Timeline Networks (*HTLN*)

HTLN redefines a decision network as a graph where each node is a tree of graphs (see description of Method bellow) as depicted in Fig. 3. A DN evolves by decomposing each node into sub-networks. A sub-network represents a possible route in which the goal might be decomposed. A goal also has an associated context with information about the situation of the goal within the plan and about the expected execution conditions. This context help the planner to decide which is the best sub-network for an specific plan. The sub-network selected is declared *active*, being the only one taken into consideration during the planning phase. This

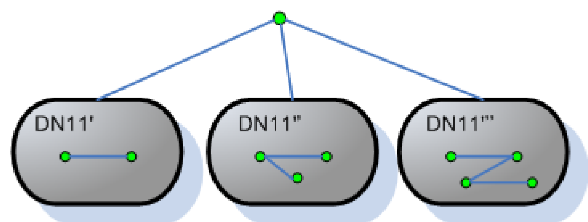

Figure 3. Tree of graphs that represent the different options in which a high level goal might be decomposed

representation is similar to those based on hierarchies of finite state machines, like XABSL Loetzsch, Risler, and Jungel [8] Risler and Von Stryk [9], a framework

used to develop the behaviour of the TU Darmstadt Robocup team, which won the world championship in 2010. The main difference between them is that APSI is more oriented to constraint satisfaction problems than XABSL. This is the reason why relations between nodes of the same component (states of the automaton) or between different components (synchronizations between components) are directly represented as the edges of a graph. Therefore, HTLN use a tree of graphs instead of a tree. At the same time, XABSL seems to be more reactive oriented. Once the system has checked the status of the world, it applies an action of the corresponding automaton. In this case, planning is reduced to the minimal expression, as it is assumed that the system will have an automaton modelling the present situation. In case there is no such automaton, or it has errors, it can be reprogrammed without assuming any risk for the system. In space missions, these dependence on the models is inadmissible, as the consequences of an error might lead to the loss of the spacecraft. That's the main reason why automaton does not play such an important role. The templates of the KDB are used as guides to help the planner, which might modify them (with the help of human experts) to fulfill the requirements of the mission. The automaton of each component is used to check whether the plan fulfils the constraints, but never as a plan to follow during execution time.

In the process of planning, the initial problem is represented as a DN which evolves until it represents a solution. The initial DN is defined by the user and composed by the initial conditions and goals represented as *methods*. Each decomposition of a goal into sub-networks represents an evolution of the DN in terms of granularity. This process continues until the DN is *fully decomposed*, that is, all the leafs of the network are *operators* (see bellow) and all constraints are satisfied. In other case, the DN is *partially decomposed*. Picture 4 shows a representation of an HTLN.
Some important concepts related to HTLN are detailed in the following paragraphs:

Definition 1 (Task (*T*)) *Set of primitive (operators) and non primitive actions (methods) available in the domain.*

Definition 2 (Operator (*O*)) *Primitive task that represents a single value choice for a state variable (concept equivalent to the operators in classical planning) or a resource activity, which can be a production or consumption.*

Definition 3 (Method (*M*)) *Non primitive task which can be decomposed into a HTLN. A method is represented as a tree of graphs. The root of the tree is a node (a decision) of the Decision Network. Each leaf is itself a Decision Network containing a graph that represents a possible way to execute the method. Each of the nodes of this graph could represent again a method (to be further decomposed) or an operator (see 4). This decomposition continues until all the nodes of the Decision Network are operators.*

Definition 4 (Relevant) *An operator o_1 is relevant for a method m if it belongs to any of the sub-networks in which the method can be decomposed. A task t_1 is relevant for other task t_2 if it satisfies any of the constraints of o_2. In this case, t_1 represents a supporting task of t_2.*

4. SUPPORTING PLANNING THROUGH PATTERN RECOGNITION

Learning represents a flexible technique that allows to "smooth" the weaknesses of planners. The only flexibility provided in classical planning comes from non-deterministic selections. If we run a deterministic planner several times, or we explore the whole search space with a non-deterministic planner providing always the same inputs, the outputs will be always the same. In case the procedure can not find a solution due to a "weakness" in its conceptualization or implementation, it will never get the solution, doesn't matter how much time we give to the planner. Learning helps to solve this problem. Pattern recognition techniques are exploited in QuijoteExpress in two main areas:

- Learning: Knowledge is gathered from the execution of a set of training examples. This technique is known as EBL (Exampled-Based Learning). This step is subdivided in another two:
  - First of all the system is trained with a subset of the examples in order to populate the KDB. The training process consists of the execution of plans (either in real or synthetic environments) and measuring the performance of the system. In our rover example, some heuristically-based metrics are provided to do it (see Heuristics section). It is also important to generate a wide spectrum of examples covering all the possibilities to make the database as complete as possible.
  - Some noise might be present in the first set of examples. To eliminate it, a second set is executed to check the validity of the information in the database. In this case, each example should match any of the templates already stored in the database. In case of error, the corresponding template of the database should be checked as it might contain wrong information.
- Exploitation: During planning, the KDB is queried for coincidences either in the whole plan or just in some goals. To speed up the process, the matching algorithm only compares the contextual information. In case some similarities are detected, the coincident parts of the current plan are replaced by the template stored in the database.

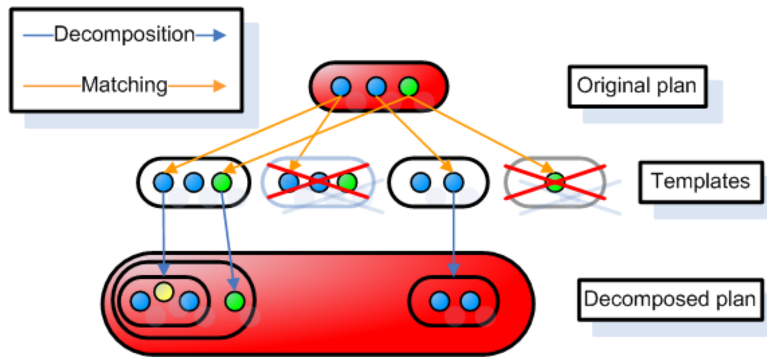

Figure 4. Decision Network generated after the decomposition of its HLGs into sub-goals. DNs are presented in red, HLGs in blue and primitives in green

Learning has been widely used in search optimization problems. However, planning problems present important differences requiring different approximations. Some planning knowledge systems have focused in learning heuristics, like Yoon [11] applied to FF planner or LaSO Xu, Yoon, and Fern [10], a more advanced approach based on discriminative-learning. But these approximations might fail in real problems, where the performance of an heuristic and the planner must be analysed taking into account the context in which the plan has been executed.

In other relevant cases like Fox and Long [7], the system tries to identify similarities between objects (taking into account their initial and final state) and between actions, based on the similarity of the objects involved. Even though this approximation seems more appropriate for real problems, it could be difficult to define manually all similar objects if we deal with a large domain. In QuijoteExpress, similarities are analysed not only between objects, but between contexts of execution for plans and goals. These contexts are automatically discovered and stored using a matching algorithm. Following three subsections give a more detailed description of this process in QuijoteExpress.

4.1. Problem Abstraction, Learning and Application

In QuijoteExpress, all the elements involved in the learning process, problems (decision networks) and goals (component decisions and relations), share the same structure. This structure has been designed taking into account the requirements of both the planner itself and the KDB (relational database). This fact represent an important advantage in terms of performance, as conversions from and to the database are reduced to the minimum.

The components of the KDB are the following::

- HTLN: The hierarchical network contains a template of a plan that has been learnt by the system. The fact that it is mandatory for a plan to be successfully executed in order to be stored in the database assures that it will work next time the system faces a similar context execution, being this similarity heuristically determined. A plan might be composed of several goals representing common tasks that use to be executed together. These "building blocks" can be reused to replace whole parts of the problem, therefore saving time during the planning process. The size and number of plans stored in the KDB is only limited by memory constraints. There are special types of templates, called *units* which contain the HTLN of a single task (a method). At the moment it is mandatory to provide a fully decomposed HTLN for each method in the system, although it is foreseen to make the system in the future able to autonomously discover such HTLN's. In fact, one goal or (more likely) one problem might present several HTLN's which represent different ways to achieve the same objective. This is the reason why a HTLN is represented as a tree of graphs, where each leaf for a given node represents a different execution option to achieve this node. One node of the tree can be decomposed in just one network at a time, that means, only one leaf can be selected.
- Context: Decision Networks and goals (component decisions and relations) are provided with contextual information. This information will allow the system to determine which template in the database better match the problem at hand. The contextual information is user-defined. Picture 5 shows the contextual information for a component decision, which is distributed in two classes:
  - Persistent: Contain all contextual information needed during the process to find the appropriate template in the database. It also contains a copy of the real object, with the HTLN structure that will replace the part of the plan in case the template match the plan. This information is only available in the objects stored in the database as it is not needed during planning or execution.
  - Context: Information relevant for both learn-

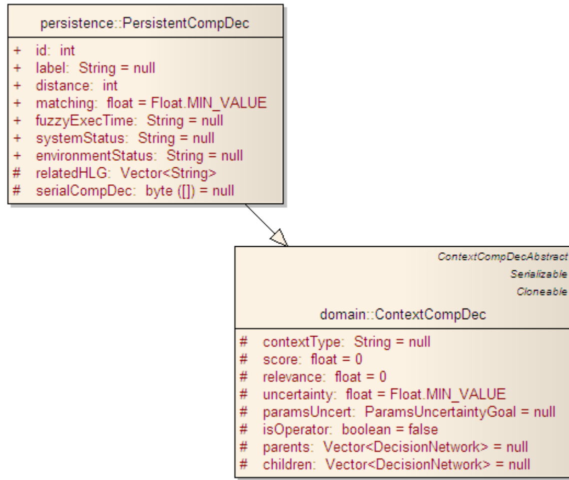

Figure 5. Contextual information of a component decision to support learning and planning

ing and planning system.

- Evaluation function: This function heuristically determines which are the templates that better fit the plan or some parts of the plan, according to the special context of the plan and each of its goals. Heuristic functions are discussed in section Heuristics.

Learning. Once the plan has been successfully executed, some contextual information is extracted. In this process, numeric variables will be ungrounded using fuzzy logic, while the enumeration variables like $f\varsigma$ or $f\epsilon$ (see description bellow) are defined by the user and used by the heuristic to identify similar plans or goals. As they are problem dependent, heuristics are defined in a separate layer from the planner, but implement standard interfaces defined in the planner layer that describe the inputs they will receive and the outputs to be generated. Afterwards, the learning system (using the heuristic) compares the plan with those already stored in the database. If there is no coincidence, the plan is stored as a DN (called *template*). If there is a coincidence, then the contextual information is updated. It is important to notice that both Learning and Exploitation use the same functions (presented bellow) in order to compare plan and templates.
From the point of view of performance, this technique helps to decrease the search space in cases where the system has been properly trained. Once the template is applied, only the constraints between high level goals must be propagated to the sub-networks.
From the point of view of robustness, as the templates are rated, the system helps to discover the “building blocks”, that is, the strategies proved to give the best results for a specific context in a similar way as genetic algorithms.

Exploitation. The process is depicted in Fig. 6.

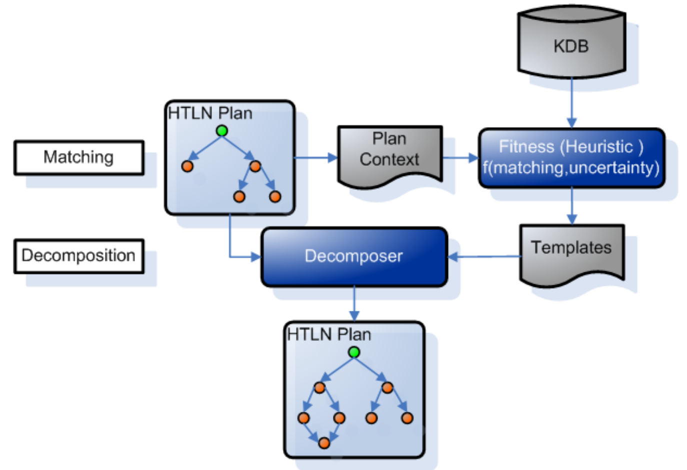

Figure 6. Structure of the learning system

Given a partially decomposed problem, the contextual information from the decision network and each of its subgoals is extracted. This information is used to query the KDB to retrieve a sorted vector with all coincident templates, each of which could cover the whole set of goals of the problem, parts of them or just one specific goal. The score function (see Heuristics section) is:

$$score_{template} = f(\mu, \upsilon) \tag{1}$$

being μ the matching and υ the uncertainty. In order to make heuristics computationally efficient, all of them use progressive functions. The score of the template is calculated as a function of the score of each of its goal. Therefore, in case something changes in the plan, the template score can be easily recalculated adding/subtracting the score of the goals added/deleted respectively. As a result, a vector of templates is retrieved, ordered by their score.
The approximation used to apply the templates to the problem is presented in Algorithm 1.

Algorithm 1 applyNextBestTemplate(problem, templates, affected)

$best = getNextBest(templates)$
while $(\neg problem.isFullyDecomposed$ **or** $\neg best = null)$ **do**
  if $(best \subset affected)$ **then**
    $recalcScore(problem, best)$
    $sort(templates, templates_{best})$
  else
    $applyTemplate(problem, best)$
    $affected = getAffectedTemplates(best)$
  end if
  $best = getNextBest()$
end while
if $(\neg problem.isFullyDecomposed)$ **then**
  return error
end if
return problem

It receives as inputs the partial problem, the set of relevant templates ordered by their score and the set of *affected* templates containing the set of templates which scores are not updated. The algorithm returns a fully decomposed problem or error in case no pattern is found for some goal. This algorithm is sound as the number of templates is finite and complete if our KDB is provided with all the *units* of the domain. It starts selecting the next best template from the list. In case a previous template overlaps some of the goals with the present one ($best \subset affected$), the score should be recalculated, deleting the effect of these goals (which have been already decomposed by the previous template). $recalcScore$ is in charge of this activity. As all heuristics used to calculate the score are progressive, this process can be achieved without high computational cost. Once the template score has been recalculated, the template is ordered and the algorithm starts again selecting the best template (that might be the same one). In case the template has not been $affected$, its decision network is applied to the problem. This action may affect the score of next templates. Each goal has a pointer to all decisions networks in which it is involved. In this way, it is straightforward to generate the list of affected templates. Finally, the algorithm checks whether the problem has been fully decomposed, returning error in other case.
A second algorithm is executed in case the planner is not able to generate a solution. In this case, the planner backtracks to choose other templates that might lead to a valid solution. The algorithm gets as inputs the problem, templates and the flawed goals detected by the planner. The templates used to decompose the failed goals are then eliminated of the templates list. The initial goals in the plan decomposed by these templates are restored and a new list of affected templates is calculated. As a consequence, the templates in the affected list might increase their scores, as there are new goals in the problem available to be decomposed. Finally, it calls back Algorithm 1 to generate a new fully decomposed problem.

Algorithm 2 backtrackStrategic(problem, templates, $goals_{Failed}$)

```
templates_Failed = getFailedTemplates(goals_Failed)
templates = templates − templates_Failed
affected = getAffectedTemplates(best)
return applyNextBestTemplate(problem, templates, affected)
```

Heuristics. Three heuristics are used to calculate the score of a plan, one for each element of equation 1. They are defined by the user in a mission-dependent layer of the software. It helps to keep the planner isolated of the details, as it only needs the list of templates and their scores.

Matching value. The matching value of a template is calculated as the aggregation of the matching value of each goal of the template.

$$\mu_{template} = k * \sum_{i=1}^{n} \mu_{goal_i} + (1-k) * \sum_{j=1}^{m} \mu_{hoper_j} \tag{2}$$

where:

- n is the number of goals in common between the problem and template.
- μ_{goal_i} is the matching value of the $i-th$ matching goal.
- m is the number of "hoper" goals, that is, goals not included in the problem but "nice to have", which are present in the template. In case there are resources available after the planner obtains a solution for the problem, it can add any of this "hopers" to give more value to the plan.
- μ_{hoper_j} is the matching value of the $j-th$ matching hoper goal.
- k is a constant that represents the relevance of goals over hoper goals.

Another constant defined in the algorithm establish a threshold for the number of matching goals of a template. If this number is lower than the threshold, then the template is discarded.
The matching value of a goal uses the variables specified on its context.

$$\mu_{goal} = k_\tau * defuzz(f\tau_{goal_p}, f\tau_{goal_t}) + k_\varsigma * defuzz(f\varsigma_{goal_p}, f\varsigma_{goal_t}) + k_\epsilon * defuzz(f\epsilon_{goal_p}, f\epsilon_{goal_t}) \tag{3}$$

where:

- k_τ: Constants used to adjust the relevance of the different variables.
- $goal_p$, $goal_t$: Goal of the plan and template respectively which matching value is calculated.
- $f\tau$: Fuzzy value of the type of execution requested in terms of *time*. It tries to represent the different time situations in which the user might want to execute a goal. For example, in the rover example, it is completely different for the rover to do an approximation traverse to a science spot, which can take minutes or hours, or doing a long traverse in order to move to a new area, which can take several days.
- $f\varsigma$: Fuzzy value that represents the status of the *environment* in which the goal must be executed. For a rover, it is completely different to charge batteries during the midday or the sunset.

- $f\epsilon$: Fuzzy value that represents the status of the system. For example, if a rover has to take pictures, it might take into consideration the status of energy and memory resources to determine how many pictures it can take and the quality of them.

Uncertainty value. The uncertainty (υ) is measured here in terms of time, as it is computationally fast and because taking time into account, we also consider in an indirect way other factors like resource consumption. The equation is presented in 4

$$\upsilon_{template} = \frac{\sum_{i=1}^{n} \upsilon_{goal_i}}{n} \quad (4)$$

Being n the number of matching goals between the plan and the template. Now, the uncertainty of a single goal is calculated as the standard deviation of the execution time. Negative values, which mean that the execution of a goal took less time than expected, are reassigned to zero to make the heuristic more conservative.

$$\upsilon_{goal_i} = \sqrt{\frac{\sum_{i=1}^{n} (time_{goal_i})^2}{n} - \overline{time}^2} \quad (5)$$

Score Finally, the score (σ) is calculated taking into consideration matching and uncertainty values. Many other variables, like the frequency of execution of each goal/template or the relative distance between goals could be applied just modifying the heuristic, but it might imply more computational cost.

$$\sigma_{template} = \frac{\rho_t * \rho_p * \mu}{\upsilon + 1} \quad (6)$$

where:

- ρ_t: Represents the ratio of matching goals respect the total amount of goals in the template
- ρ_p: Represents the ratio of matching goals respect the total amount of goals in the plan

In this way, we give better scores to those templates which best cover the plan goals.

5. CONCLUSIONS

At present, planning theory has achieved sufficient maturity to be in charge of the automatic generation of critical plans for real applications like space missions ECS [5].

One concern might be the limited computational capabilities on-board the spacecraft. However, taking into consideration the long term schedules managed for space missions, we should start now preparing the software to be run in future hardware platforms. Even though it is usually subject of discussion, there exists a big gap between theoretical and practical planners. It is Space Agencies duty to invest efforts in order to bridge this gap.

This paper goes in this direction, providing a fusion of different theoretical technologies from planning and AI worlds. Central ideas in the design of QuijoteExpress like high level goals, parallelism and distributed systems will represent key concepts in future space mission scenarios. At the same time, the use of AI techniques like the pattern matching presented here will play an important role in order to develop more informed planners and heuristics required to address the increasing complexity of plans and search spaces.

REFERENCES

[1] Cesta, A., and Fratini, S. 2008. The timeline representation framework as a planning and scheduling software development environment. In *In PlanSIG-08, Proceedings of the 27th Workshop of the UK Planning and Scheduling Special Interest Group*.

[2] Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

[3] Donati, A.; Policella, N.; Cesta, A.; Fratini, S.; Oddi, A.; Cortellessa, G.; Pecora, F.; Schulster, J.; Rabenau, E.; Niezette, M.; and Steel, R. 2008. Science operations pre-planning & optimization using ai constraint-resolution - the apsi case study 1. In *Proceedings of the 10th International Conference on Space Operations*.

[4] Dvorak, D. L.; Ingham, M. D.; Morris, J. R.; and Gersh, J. 2007. Goal-based operations: An overview. In *Proceedings of AIAA Infotech@Aerospace Conference*.

[5] ECSS-E-70-11c – Space engineering – Space segment operability. ESA Publications, Noordwijk, The Netherlands, 31 Jul. 2008 (available from http://www.ecss.nl/).

[6] Fesq, L.; Ingham, M.; Pekala, M.; Eepoel, J. V.; Watson, D.; and Williams, B. 2002. Model-based autonomy for the next generation of robotic spacecraft. In *In proceedings of International Astronautical Congress*.

[7] Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proceedings of IJCAI-99*, 956–961. Morgan Kaufmann.

[8] Loetzsch, M.; Risler, M.; and Jungel, M. 2006. Xabsl- a pragmatic approach to behavior engineering.

[9] Risler, M., and Von Stryk, O. 2008. Formal behavior specification of multi-robot systems using hierarchical state machines. In *In AAMAS08-Workshop on Formal Models and Methods for Multi-Robot Systems*, 12–16.

[10] Xu, Y.; Yoon, S.; and Fern, A. 2007. Discriminative learning of beam-search heuristics for planning. In *Proceedings of the International Joint Conference on Artificial Intelligence 2007*, 2041–2046.

[11] Yoon, S. 2006. Learning heuristic functions from relaxed plans. In *Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS-06*. AAAI Press.