

Supporting Complex Astronaut Tasks: The Right Advice at the Right Time

A. Bos, L. Breebaart, T. Grant
S&T BV
Delft,
The Netherlands
bos@science-and-technology.nl

M. Neerincx
TNO-D&S
Soesterberg,
The Netherlands

M. Wolff
ESA-ESTEC
Noordwijk,
The Netherlands

U. Brauer
EADS ST,
Bremen,
Germany

A. Olmedo Soler
OK-Systems,
Valencia,
Spain

Abstract

A challenge for future human planetary exploration missions will be to empower the astronauts with sufficient cognitive support so that they can make decisions in much more autonomous fashion than in current missions. Future astronauts will be told the goal they need to realize; they have to work out for themselves how to realize these goals. This paper describes a planning-based system that based on an accurate assessment of the current state is able to support the astronaut in realizing the mission goals. As for the realization of the mission goals the use of complex equipment is crucial, the support includes also the operation of such systems.

1. Introduction

This paper describes a flexible concept for astronaut support during human planetary missions. This concept is being developed in the context of the *Mission Execution Crew Assistant (MECA)* project [1]. The MECA project aims to define requirements and develop a proof-of-concept demonstrator for advanced informatics support to exploration teams engaged in future planetary surface operations. Such teams have to operate in a complex, dynamic, hostile, and uncertain environment. In this environment the team will have to solve many problems and will continuously have to decide what to do, all in the knowledge that a wrong, or simply a late decision may result in Loss of Life (LoL) or Loss of Mission (LoM). This decision-making proc-

ess can be characterized as follows: the problems that the astronaut faces (i) are ill-structured and described in uncertain terms, (ii) are the result of competing goals, (iii) need to be solved under high stress levels, and (iv) require cooperation of multiple parties.

Once they have left Earth orbit, the crew members are basically on their own. The long distance traveled by the team impedes frequent consultation with ground control. Therefore, the team needs to be prepared to make decisions autonomously, even during critical phases of the mission. Earth-based support will shift from detailed astronaut guidance on an operational level (as is currently done), to a more strategic guidance allowing the crew sufficient freedom to construct and implement their own short-term plans. That is, the team will be continuously asked by mission control to realize a set of *mission goals*. How these mission goals are to be realized, however, will be the responsibility of the crew, aided by the MECA software.

The team will be committed to completing their mission with only a limited supply of resources (e.g., food, fuel, and oxygen) that have been carried from Earth or generated at the destination. Each decision made by the astronauts must take into account resource usage and a projection of the activities that still need to be performed using these resources.

Particularly during the initial missions, a large portion of crew time will be devoted to checkout and to the maintenance of complex equipment. After arrival at their destination, the astronauts will have to gain confidence in the ability of the MECA system to support them during their stay and their work. This requires that the astronauts have sufficient insight into the

workings of the equipment to determine system health and to conduct repairs.

In order to deal with these task, context and resource dynamics, the MECA system will act in a ubiquitous computing environment as an “electronic partner”, helping the crew to assess the situation, to determine a suitable course of actions to solve a problem and to safeguard the astronaut from failures.

The SCOPE2 system is an application framework developed in order to tackle many of the capabilities a MECA system needs to offer when it comes to providing support for the execution of mission goals. It is based on the concepts of semantically augmented operational procedures and model-based fault diagnosis.

Section 2 introduces context and specific definitions of terms used in the remainder of this paper.

Section 3 describes the concepts of plan-based control and model-based reasoning as they apply to the astronaut activities MECA intends to support.

Section 4 further describes SCOPE2 as a first-generation version of some of the concepts that will be required by the MECA demonstrator.

Section 5 summarises the lessons learned and describes some of the avenues we intend to explore during the remainder of the MECA project.

2. Background

In this section we will introduce a number of definitions and concepts that will be used in this paper.

2.1. Process Under Control

For its success and safety, a planetary exploration mission must rely on the correct functioning of large numbers of complex equipment. The MECA system will play a major role in the supplying the crew with support for the control and operation of these systems. We will use the term *Process Under Control* (PUC) to denote the environment in which the astronauts have to operate, or, putting it more technically, a configuration of systems over which the exploration team has to reason in order to fulfill the mission goals. In general, a PUC is considered to be a *system-of-systems*, where the behavior of one system may depend on the state of other systems. The different elements constituting the PUC may be complex systems such as spacecraft, facilities, habitats, and rovers, all with intricate behaviors in their own right.

2.2. Mission goals

Mission control will communicate the objectives that have to be realized by the exploration team in terms of *mission goals*. Our technical usage of the term “mission goal” in this article will be closely related to the *state space* of the PUC. We will assume that there is a simple procedure that decides whether a PUC state *satisfies* the mission goals. The task of the exploration team is to find a sequence of control actions that will bring the PUC into a state that will satisfy the mission goals.

In general, the exploration team will have to realize multiple, simultaneous mission goals. These multiple goals may be ordered according to desired completion time (i.e., goal *A* has to be realized before goal *B*), and may be mutually inconsistent. In order to decide how to deal with inconsistent goals, mission goals may have a preference order or weighting factor associated with them.

2.3. PUC state

We will assume that the behavior of the PUC can be sufficiently described using a discrete state-based formalism. This means that we will be able to use the concepts of, e.g., the *current state* of the PUC, or the desired *end state* of the equipment. Furthermore, an action by an astronaut or by a machine (e.g. the opening or closing of a valve) may bring the system from one state into another.

We will consider both nominal states and fault states of a PUC. A fault state is a state representing a situation in which one of the PUC’s components is at fault.

2.4. Operational procedures

Operational procedures can be viewed as the atomic tasks or *actions* that, when executed in a certain sequence by the astronaut, will implement a state change on the Process Under Control.

2.5. Resources

Resources play an important role in describing the necessary conditions for the realization of a mission goal. In a technical sense, resources can be represented by a predicate defined over the state space of the PUC, stating how many instances of a particular resource type are available to that state. Resources can be used in the context of an operational procedure. For example, an operational procedure may have an associated

precondition that states the number of resources that should be available before the action can be executed.

Resources may be (partly) “consumed” by an action so that they cannot be used by subsequent actions. We call these *consumable* resources. Resources may also be “used” by an action so that once the action has ended the resource is “released” for future usage by other actions. We will call these *reusable* resources.

3. Main reasoning methods: planning and diagnosis

The support given to astronauts is targeted at boosting their cognitive abilities. In order to realize this objective, the astronaut will be supported with automated reasoning methods. The main reasoning methods that we consider are: (i) planning to derive a sequence of action to bring the PUC in such a state that it will satisfy the mission goals, and (ii) diagnosis as an example of a reasoning method to improve the astronaut’s situational awareness.

In this section, we will demonstrate that these reasoning methods will take place at various levels of abstraction. The system supporting the astronaut will have to be able to support reasoning at all of these levels as well. Next we will describe the planning and diagnosis reasoning process in more detail.

3.1. Reasoning over different levels of abstraction

In general, satisfying mission goals will involve reasoning over many levels of abstraction. For example, in order to control oxygen production, the astronaut will have to incorporate current and future resource demands relative to the tasks that need to be performed. Depending on these resource demands, the astronaut will control the equipment involved in the oxygen production Process Under Control. During this resource-level type of reasoning, the astronaut is likely to ignore the possibility that the equipment might contain a failure, and assume nominally functioning components. Once the control plan is refined, however, this failure state becomes important, as in reality failures and exceptional cases *do* occur.

For example, one of the PUC components may not be working as intended. In such a case, the PUC details cannot be ignored anymore, and the astronaut has to repair the situation. The repair action may be restricted to a “simple” reset of a PUC component, but may also require system diagnosis and repair of the equipment. In the latter case, the astronaut needs to

shift his reasoning from the domain covering the logic of the experiment to the domain of the PUC.

The required change in the reasoning domain stresses the cognitive abilities of the user, as the reasoning on, say, the PUC level requires knowledge that is unrelated to the reasoning on the experimental level. The software architecture of the system needs to support the astronaut also on the different levels. These levels will be reflected in the architecture.

3.2. Plan-based control

We will view *mission goal satisfaction* as the process of transforming the PUC’s initial state into a state that satisfies the mission goal. These states will be representations of the PUC, and will, e.g., describe situations such as: “*The LSS is producing O2 at 80% of its nominal rating*” or: “*Valve V003 is stuck closed*”. The transition from initial state to a final state can be realized by issuing control commands, e.g., “*Set the desired (set point) value of O2 production to 100% of its nominal rating*”, or “*Isolate Valve V003 from the rest of the circuit by, e.g., enabling redundancy circuit XY2*”. Due to the complexity of the environment (PUC) it is not possible that all the sequences of possible actions can be pre-computed. Instead these actions must be derived from the actual situation, the desired situation, and the available capabilities of the astronauts and the equipment. Deriving such a sequence is called *planning* [4].

The planning process can be described as follows: The planning problem concerns the derivation a sequence of actions that transforms the current PUC state (s_0) into a goal state (s_g). That is, a planning agent reasons over the state space of the environment in terms of the current and goal state, and the operators that induce a state transition (also called actions). The problem of planning is usually defined in terms of three items:

- A description of the current state.
- A description of the goal state.
- A description of the operators, i.e., the description of how an operator transforms a state into another.

The resulting sequence of actions should be such that the resulting state changes transform the current state into the goal state.

Furthermore, in general, the execution of an action is not cost-free, but requires the consumption of a cer-

tain number of resources, such as e.g. various forms of energy carriers (i.e., fuel), people, and time.

We assume that the planning process is resource-constrained, that is: the total resource consumption of an action sequence $\langle a_1, \dots, a_n \rangle$ will not exceed a certain boundary value. The process of assigning actual resource instances to action is called *scheduling*.

Cooperation between two or more reasoning agents (such as a human astronaut and a SCOPE2-based MECA Unit) may be motivated by at least two reasons: (i) the plan of one reasoning agent requires a resource (or an action) that is in the possession of another agent, or (ii) the joint plan is more economical than the sum of the two individual plans [5].

The idea is to allow human and automatic planners to cooperate in producing a plan. There are several aspects to this interaction. First, this way of mixed-initiative planning simplifies the problems facing the automatic planning, since difficult choices in plan construction can be passed to a human, while the human can benefit from not having to manage the bulk of easier planning decisions and simple bookkeeping tasks. Furthermore, the automatic planning task must also support tasks such as plan repair and iterative plan improvement besides complete plan construction.

3.3. Reasoning over system behavior

Besides reasoning with the aim to establish the basic steps that will satisfy the mission goals, the astronaut will also have to reason at a much lower level of detail. For example, if the main plan says that a certain piece of equipment must be brought in a certain state, but the equipment is in a fault state, then the astronaut has to derive the main reason for this failure, and fix it if possible (see also Figure 2). There are various techniques to reason over device behavior. One of the most promising is Model-Based Reasoning. In this section, we will investigate this type of reasoning in more detail.

3.3.1. Model-based reasoning. Model-Based Reasoning (MBR) uses a technique that resembles that of experienced mechanics (see e.g. [6], [7], [8], and [9]). These mechanics do not have to have any experience with a particular piece of equipment that they need to repair. All they need is generic knowledge of how basic components function under nominal conditions, a structural description of how the system is composed out of the basic components (often in terms of a schematic of the system), and the observations telling them how the system actually behaves. These ingredients are

in general enough for an experienced mechanic to find, e.g., the root cause of a failure of many different types of equipment. MBR uses a qualitative (an abstract, causal description) model of a system to infer how to operate the system, diagnose failures and generate appropriate behaviour to repair or reconfigure the system in response. Models are composed out of component descriptions, enabling a relatively low-cost direction towards autonomy. Rather than requiring engineers to envision all possible interactions and failures at design time or perform analysis during the mission, the reasoning engines generate the appropriate response to the current situation, taking into account its system-wide knowledge, the current state, and even sensor failures or unexpected behaviour.

A mechanical reasoning system that has been built according to the principles of MBR consists of two major components:

- A domain-independent generic reasoning system performing the logic inferences in order to, say, find the root cause of failure.
- A domain dependent description of system behaviour, the model, consisting of a description of the basic components that are used in the equipment, and a structural description of how the basic components are connected to each other.

Consider the problem of diagnosing a faulted system. As we are considering automatic means to diagnosis only, the fault must have a distinguishable effect on the sensor values. This effect, of symptoms, may not always be present, but may depend on the actual state of the system. The main objective of the health management system, we are considering in this report, will be the discrimination of the component (or, in general, the components) responsible for the observed symptoms.

The traditional approach towards diagnosis could be said to be symptom-based (this includes methods based on fault trees, filter banks, learning-based methods such as neural nets, etc). In order to explain the concept of symptom-based, we consider the *design effort* to construct the diagnosis system, and the *run-time* behavior of the diagnosis system, during which the diagnosis system actually determines the root cause of a failure based on the sensor values.

During run-time, the symptom-based approach makes use of a precompiled list of symptoms that are linked to the actual causes. A symptom-based diagnosis system searches this symptom-list for a best match with the actual symptom.

The major disadvantage of the symptom-based approach is that during effort required to develop such a diagnosis system. Usually, a human modeler has to work out how a system failure affects the sensor readings. For complex systems, this modelling task quickly becomes too labour-intensive and error-prone.

Furthermore, for each change in the design the fault propagation process has to be performed from scratch.

The model-based approach towards diagnosis automates much of the reasoning. That is, the model-based approach hypothesises system failures and automatically propagates the failure effects. In this way, a fault hypothesis can be tested by comparing the failure effects with the observations. It is said that a failure hypothesis is a fault diagnosis if the effects are consistent with the observations.

3.3.2. Modeling. A simple example system is presented in Figure 1.

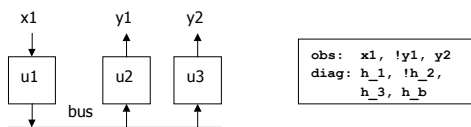


Figure 1. An example system

The system consists of three unit boxes communicating with each other via a bus structure. In the model the three units and the communication box are each modeled as a separate subsystem. For the sake of illustration, the behavior of each of the units is modeled as a simple copy of the input in case the unit is working as intended, and giving no output in case the unit is faulted. If we use \mathbf{x} to represent the input of the unit, \mathbf{y} to represent the output; and \mathbf{h} to represent the health of the unit, then the unit system can be modeled as:

```

system unit(x,y,h: bool) {
  h => y = x;
}

```

The communication bus is treated similarly:

```

system bus(x,y1,y2,h: bool) {
  h => (y1 = x and y2 = x)
}

```

The whole system can then be composed out of the **unit** and **bus** descriptions:

```

system comm(x1,y1,y2: bool) {

```

```

  unit(x1, o1, h1);
  unit(o2, y1, h2);
  unit(o3, y2, h3);
  bus(o1, o2, o3, hb);
}

```

If we observe that an input command ($\mathbf{x1}$ is **true**) is given to **unit1**, reflected by **unit2** ($\mathbf{y1}$ is **true**) but not by **unit3** ($\mathbf{y2}$ is **false**), then the diagnosis engine will search for a health assignment (to the elements **h1**, **h2**, **h3**, and **hb**) such that the observations correspond to the model equations. The diagnosis engine will come up with that an explanation for observed behavior is that **unit3** is faulted, i.e., **h3** is **false**, and other components must be working as intended, i.e., **h1**, **h2**, and **hb** are **true**. Thus a diagnosis is represented by the health assignment for each component (in this example: **h1** and **h2** and **not h3** and **hb**).

In general, there can be several possible health assignments that will explain observed behavior. In case the diagnosis cannot determine the diagnosis to be associated with a single unit, the model-based approach will be able to suggest new measurements that will reduce the level of ambiguity. Using information theory concepts, the system then computes the variable that will reduce the ambiguity group most once its value is known (as more information about actual system behavior is known, some of the health mode assignments will be rendered inconsistent with the actual observations). In this way, the system searches for the component responsible for abnormal behavior in an iterative fashion.

3.3.3. Reconfiguration. Besides diagnosis, it is also possible to perform other types of reasoning. An example of an additional reasoning method is planning for *reconfiguring* the system. If the fault state of a system is known, then algorithms similar to diagnosis can be used to derive a sequence of actions that will bring the system in the desired state (see, e.g., [7]). Note that reconfiguration is a form of planning as well; again, we are searching for a set of actions that will transform the system in a desired state. The major advantage of using the model-based approach here is that the same model that is used for diagnosis purposes can also be used for reconfiguration as well.

3.4. Architecture

In order to mimic the multi-level reasoning, we assume that there are at least two reasoning levels over which the system has to give support. A high reasoning level

that describes coarse sequence of actions to realize the mission goals (for example a description to conduct an experiment), and a low reasoning level that describes how the various sub-goals need to be satisfied. Figure 2 depicts this reasoning over different levels of reasoning in a graphical fashion. The top of the figure denotes the high level plan. The hexagons represent the basic actions. Each basic action has a post-condition representing the next state; the final state is such that the mission goals are satisfied. The post-condition of each high-level action is used to control the low-level reasoning in the following way:

- ❑ The post-condition of the action denotes the sub-goal that needs to be satisfied. The sub-goal must be will be satisfied at the lower-reasoning level, e.g. by controlling the PUC equipment.
- ❑ Furthermore, the post-condition can be used to test whether the basic action (controlling the equipment) has been executed correctly. The resulting state should be consistent with the post-condition of the action.
- ❑ If the action was not executed correctly, then the diagnosis functionality of the lower-level reasoning process can be used to derive what is wrong. Furthermore, system reconfiguration reasoning capabilities are applied to restructure the equipment in such a way that the action's post-condition can still be satisfied.

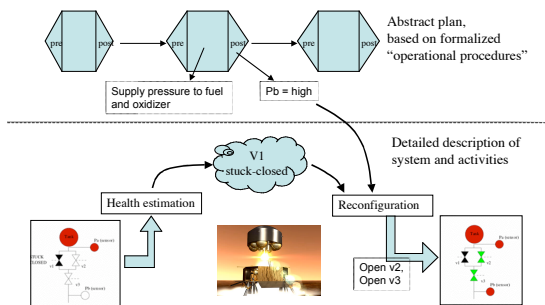


Figure 2. Layered reasoning

Figure 3 depicts the basic reasoning architecture to support astronaut reasoning. The intention is to control the PUC to satisfy mission goals. Central in this representation is the high-level plan. This high-level plan is used to derive actions to control the PUC. The plan is derived from the basic actions available and the safety conditions to obey, and the capabilities delivered by the PUC (which depend on the state of the PUC). The reconfiguration system is used to convert the high-

level action of the plan into a sequence of PUC actions incorporating the PUC's state.

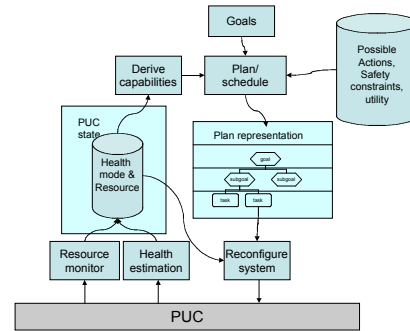


Figure 3. The reasoning architecture

4. Supporting astronaut activities

4.1. SCOPE2

SCOPE2 [1] can be seen as a first iteration implementation of some of the concepts described in the previous sections, specifically those of a simplified Processes Under Control, and of mission goals implemented as a sequence of tasks.

Most current astronaut support systems are based on static operational procedures: pre-computed sequences of basic actions. A major disadvantage of most of these (sequences of) operational procedures is their static structure. For most astronaut support systems it is not possible to dynamically adapt the execution order of the operational procedures, let alone a procedure itself, depending on the state of, say, the equipment being maintained.

For example, when attempting to deal with an equipment malfunction, most current operational procedures contain a fault-tree for use in pinpointing the root cause of failure. Constructing and presenting a fault-tree as an operational procedure has many disadvantages: (i) it is *expensive*, as a lot of human inference is needed to construct such a tree; (ii) it is *incomplete*, as it requires a number of simplifying assumptions to construct (e.g. assume a single failure); (iii) it is *hard to maintain*, as the fault tree consists of complex if-then-else structures; and (iv) it is *inefficient*, as the fault tree requires questions about the state of the equipment to be answered by the astronaut in a fixed order, whereas the known symptoms might already hint at a particular fault — information a smart support

system could exploit in order to steer the diagnostic process.

The SCOPE2 system was designed to remedy these defects and thus improve the situational awareness of its users. SCOPE2 is basically a generic application framework that can be instantiated (via a plug-in mechanism) with operational procedure catalogues, control panels, task sequences, HTML documentation, and diagnosis models all dedicated to a specific real-world system. This mirrors the Process Under Control as seen in MECA. SCOPE2 then presents the astronaut with an integrated, ‘instantiated’ console application containing a hierarchical presentation of the task sequences to be executed, with integrated documentation and control panel functionality (via hyperlinks), but also with integrated model-based diagnostic capabilities.

SCOPE2 keeps track for the user of the states of individual tasks/procedures (e.g. *inhibited*, *active*, *completed*, etc.), and attempts to ensure that these states correctly reflect the state of the process under control (as determined by e.g. sensor readings). Should SCOPE2's diagnostic engine detect a system anomaly, it will initiate an iterative dialogue with the user that may in turn lead to a dynamic adaptation of the primary task sequence (e.g. by inserting a subsequence of repair procedures).

4.2. Traditional operational procedures

As mentioned in section 2, an operational procedure is an abstraction of an atomic action that is meant to change the state of the PUC.

For example, when configuring a piece of equipment, we could start with a world that has an initial state in which all the components are stored. The desired end state of the "set up equipment" mission goal task sequence would be a world in which the astronaut has unpacked all components and mounted/installed the equipment according to a certain sequence. Each procedure step is a representation of an action intended to realise the desired state transition. The end goal of the compound procedure is reached through a series of intermediate goals / state changes, described by each of the operational procedure making up the task sequence.

Traditionally, operational procedures are static pieces of information (hardcopy text paragraphs, in the most literal case). In the SCOPE2 implementation, operational procedures are augmented with the concept of state/lifetime (e.g. a procedure can be *active*, or *completed*). This in turns allows two major improvements, namely (a) the definition of pre- and post-

conditions on the lifetime of the procedure, and (b) the possibility of associating scripts with a procedure.

4.3. Pre- and post-conditions

While the execution of one (or maybe more) sequences of operational procedures will lead to a desired change in world-state, other sequences may not have that effect. In general, we are therefore interested in (and SCOPE2 will assist in enforcing) sequencing constraints that impose a certain order on the actions (e.g. “first unpack, then mount”).

One simple way of implementing a sequencing constraint is to equate presentation order with sequence order, and effectively say: “perform these activities in the order in which SCOPE2 is showing them”.

This is, however, a very limited, inflexible approach. A more useful approach is to define ordering constraints not explicitly as sequences, but rather in terms of *conditions* on the state of the world, both prior to and after executing the procedure.

For example, before a procedure `mount-component-A` can be ‘executed’ by the user, the state of the world must satisfy the precondition `component-A-is-on-table`, which may itself be the result (or post-condition) of an earlier procedure step. This way of specifying constraints induces a desirable partial ordering of operational procedures.

In SCOPE2 procedure pre- and post-conditions are used as a means of encoding world state transitions and thus help accomplish the mission goal associated with the process under control, while decreasing the cognitive load on the astronaut (e.g. by showing only the currently “available” procedures). Similarly, conditions can be used to implement resource-based plan-control, as seen in Section 3.

4.4. Scripting

SCOPE2 can also support the astronaut in controlling real world systems by means of a Virtual Control Panel for that system. By using *procedure scripts*, this connection can be made more explicit and supported by the application.

An operational procedure can have a script attached to it (in the current implementation the script in question can in fact be any arbitrary Java code), which will be executed when the user ‘activates’ that procedure.

For example, if a procedure states that the user must open a valve and wait for a container to drain, this can actually be done for the user by the script — activating the procedure will be sufficient to start the

script code, and SCOPE2 will itself signal when the procedure has finished.

In this way, script- and state-augmented procedures make it possible for the SCOPE2 system to become a much more active partner in the dialog between the astronaut and the process under control.

5. Conclusions

SCOPE2 and its predecessor, SCOPE, have seen a number of usability trials in which various groups of users (ranging from students to actual astronauts) were tasked to run through various scenarios using SCOPE/SCOPE2, as instantiated for a given medical payload [2] From these trials it is clear that the combination of the augmented operational procedures, the automated diagnosis and the various integrated support functions succeed in lowering the cognitive load on the users when effecting state changes on a Process Under Control in order to achieve the mission goal.

For use in the MECA project, we intend to extend the augmented procedures further so that true resource-based plan control and multiple concurrent mission goals (as described in Section 3) can be implemented. Also, MECA will be a far more distributed environment than SCOPE2 has so far been used in, so we will need to develop networked, agent-based versions of SCOPE that will be able to support a far greater variety of systems than has so far been the case.

6. References

- [1] A. Bos, L. Breebaart, M. A. Neerinx, and M. Wolff, "SCOPE: An intelligent maintenance system for supporting crew operations," in Proceedings of IEEE Autotestcon 2004, 2004, pp. 497–503.
- [2] DTTP brochure containing Cardiopres description: www.esa.int/esapub/br/br231/br231.pdf
- [3] Mission Execution Crew Assistant, <http://www.crewassistant.com/>.
- [4] Danial. Weld "An Introduction to Least Commitment Planning," AI Magazine, 15(4), Winter 1994, pp. 27–6.
- [5] Hans Tonino, André Bos, Mathijs de Weerd, Cees Witteveen: Plan coordination by revision in collective agent based systems. *Artif. Intell.* 142(2): 2002 pp 121–145.
- [6] J. Kurien, P. Pandurang Nayak, and B.C. Williams, Model-Based Autonomy for Robust Mars Operations, Founding Convention of the Mars Society, Colorado, 1998.
- [7] Pandurang Nayak, B. Pell, and B.C. Williams, Remote agent: To boldly go where no AI system has gone before, *Artificial Intelligence*, 103(1/2), August 1998.
- [8] B. C. Williams and P. P. Nayak. Immobile Robots: AI in the New Millennium. *AI Magazine*, pp.17–34, Fall 1996.
- [9] B.C. Williams, and P.P. Nayak, A Model-based approach to reactive self-configuring systems, in Proceedings of AAAI-96, 1996, pp. 971–978.