A multiagent architecture for controlling the Palamede satellite

Francesco Amigoni a,∗, Stefano Gualandi a, Daniele Menotti a and Guido Sangiovanni b

a Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy
b Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy

Abstract. The fundamental role of autonomous agents in managing activities of space systems has emerged some years ago with the NASA’s Remote Agent Experiment. However, the possible advantages of employing multiple agents to manage activities on a single space system are largely unexplored. This paper gives a contribution in this direction by presenting the design and the experimental validation of a multiagent architecture intended to run onboard a space satellite. Each agent is associated to a subsystem of the satellite and manages its activities. Each agent is organized in three modules, or layers: a planner, a scheduler, and an executor. Taken together across the agents, these layers form a distributed planner, a distributed scheduler, and a distributed executor, respectively. With a multiagent architecture, a number of benefits, including robustness, easy reuse of agents, and the possibility for the designer to focus on a small portion of the problem at a time, can be exploited. We experimentally validated our system in the scenario provided by Palamede, a low Earth orbit satellite under development at the Department of Aerospace Engineering of the Politecnico di Milano.

Keywords: Autonomous space systems, multiagent architectures, multiagent planning, multiagent scheduling

1. Introduction

An autonomous space system must be capable, to some extent, of identifying the goals of the mission, planning its activities, executing and monitoring the planned actions, detecting the presence of failures and, in these cases, deploying some recovery strategies [7]. The fundamental role of autonomous agents in managing activities of space systems has emerged starting from the NASA’s Remote Agent Experiment [28]. Usually, planning and scheduling are considered as basic building blocks of autonomy. Planning builds effective courses of actions to be undertaken in order to reach some desired or assigned goals; while scheduling selects among alternative plans to appropriately manage resources concurrently needed by different actions in a plan. It is currently accepted that, in real-world applications, these two aspects must be integrated [33]. Moreover, increasing interest has been devoted to systems in which a number of agents cooperatively plan or schedule their actions [13].

In this paper, we view a spacecraft as a collection of subsystems, functionally detached but interconnected and interdependent in order to reach the goals of the mission. Each subsystem (device) is associated to an agent. In the context of this paper, we consider an agent as an independent computer program that controls the associated device. We present a multiagent architecture for controlling Palamede, a low Earth orbit satellite under development at the Department of Aerospace Engineering of the Politecnico di Milano. The control of a satellite implies the management of its activities and, basically, their planning, scheduling, and execution. Hence, each agent is composed of three modules organized in a layered structure [28]: a planner, a scheduler, and an executor (from highest to lowest level). In general, a module interacts only with the adjacent modules in the layered structure. For exam-
The scheduler interacts with the planner and with the executor. The corresponding (i.e., at the same level) modules of different agents create a planner, a scheduler, and an executor, respectively, that are distributed. The multiagent approach appears to be effective in dealing with satellite management. In general, it is widely accepted that a collection of distributed agents that work and act locally can be more robust (in coping with uncertainties, unforeseen events, or failures) than a single centralized manager. For example, “the failure of one or several agents does not necessarily make the overall system useless, because other agents already available in the system may take over their part” [37, p. 8]. The separation of functions, the decentralization of activities and responsibilities, and the parallelism enable each agent to quickly answer to local unforeseen events or faults, without involving the overall system. Moreover, the multiagent paradigm allows the designer to focus on small portions of the problem at a time, simplifying the analysis of complex problems [21]. For these reasons, the multiagent approach is increasingly employed in space applications. However, it is usually applied to formations or constellations of satellites working together [9] or to multiuser scenarios [19]. In this paper, we give an original contribution to investigating the use of multiple agents to manage the activities on a single space system. This topic is, to the best of our knowledge, largely unexplored (with few exceptions, including the IDEA architecture [27] and the reinforcement learning-based proposal of [26]). Note that the multiagent approach is usually applied when the elements of the system are physically detached, which is not the case for the sub-systems of our satellite. Less often, as we do in this paper, agents are associated to functionally detached elements, enhancing the modularity of the system without significant loss of performance.

The main original contribution of this paper is to adopt a multiagent system to control a single satellite by associating the agents to its subsystems (devices). According to this perspective, we do not propose original methods for multiagent planning and scheduling but an original approach to the problem of developing control systems for autonomous satellites. Preliminary versions of some parts of this paper, mainly related to the distributed scheduler, appeared in [1,31].

This paper is organized as follows. The next section reviews the related works. Section 3 introduces our reference scenario constituted by the Palamede satellite. Section 4 overviews the multiagent architecture we have developed for Palamede. Sections 5, 6, and 7 present the distributed planner, the distributed scheduler, and the distributed executor, respectively. Section 8 experimentally validates the proposed multiagent architecture. Finally, Section 9 concludes the paper.

2. Related works

Many efforts have been devoted to the autonomy of spacecrafts. In this section, we compare our approach with the most significant agent-based software architectures for autonomous space systems. Other discussions on the literature related to some specific topics (e.g., planning and scheduling) are presented throughout the paper.

2.1. Single agent architectures

The classical single agent architecture for space applications is the so-called three-layered architecture, composed of a deliberative or reactive planning layer, a monitor layer, and an executive layer. The prominent example of this architecture is the Remote Agent experiment (RAX) conducted by NASA [28]. The idea is to separate the functionalities of the sense-decide-act cycle and to have a layer for each set of similar functionalities. The layers are designed as independent software components that exploit the functionality of adjacent lower layers. The advantages of the three-layered architecture are its simplicity and modularity. NASA's RAX has an integrated planner/scheduler that generates a set of time-based event-based activities [28]. However, RAX has shown the drawbacks of such approach. If the three layers do not use the same semantic model of the autonomous system, then subtle differences of the models can result in planning or execution failures hard to debug. This is the main motivation that has led to the development of the Intelligent Distributed Execution Architecture, IDEA [27], discussed later. The architecture proposed in this paper uses a single semantic model of the satellite.

The three-layered architecture has also been used by the Autonomous Spacecraft Experiment (ASE) onboard the Earth Orbiter 1 mission in 2003 [6,8]. The main idea of ASE is to perform onboard science analysis in order to store and transmit to the ground station only very valuable data, and possibly to autonomously change planned activities according to science analysis results. Science analysis includes, for instance, detection of thermal anomalies, clouds, changes, and gen-
eralized features, and classification of flood scenes. The main loop of an autonomous science experiment is: (i) take an image as planned, (ii) process onboard the image (i.e., to perform science analysis), (iii) re-plan onboard, (iv) re-target for new observation goals, and then looping back to (i). In order to perform this loop, several onboard techniques are necessary: continuous planning, machine learning, and pattern recognition. These techniques fit into a three-layered architecture, with onboard science algorithms interfacing with the planner. The CASPER planner [23] is used at the deliberative layer, while the Spacecraft Command Language (SCL) is used at the executive and monitoring layer. Unfortunately, CASPER and SCL reason on different semantic models, similarly to what strongly discouraged by the RAX experience. We also cite ASPEN, that is a planner/scheduler system similar to CASPER, but developed for managing ground operations [7].

A delicate issue with three-layered architectures is the interfacing between monitor and planning layers. When should the monitor invoke the planner to repair or re-plan against a plan failure? How is it possible to enforce some hard real-time constraints? How to build fault-tolerant plans? These are some of the questions addressed in [2], where the authors present the Cooperative Intelligent Real-time Control Architecture II (CIRCA-II). Their work describes an interface between a real-time resource allocator and a planner to automatically create fault-tolerant plans. Fault tolerance is guaranteed only with respect to a list of pre-determined possible faults. For each fault, a plan is constructed and fed to the real-time resource allocator (i.e., a real-time scheduler). If the plan is not schedulable, the most costly task is identified and used to dynamically backtrack the search for another valid plan. The planning algorithm is based on a state-space search, and uses a probability model to guide the exploration and to prune “highly improbable” states.

Finally, the Nexus project (by NASA’s Marshall Space Flight Center) is intended to be the planning and scheduling system for Lunar and Mars explorations. It will allow the cooperation of multiple users in order to build an integrated schedule [19]. This system, differently from our proposal, tackles the problem of managing space activities in presence of human operators. Although our agents use a three-layered architecture, the distributed multiagent architecture proposed in this paper is not directly comparable to the above systems because they follow a single agent approach.

2.2. Multiagent architectures

IDEA [27] is a multiagent architecture for a single autonomous system developed by the same team that worked on RAX. The objective of IDEA is to move from the three-layered architecture of RAX towards a distributed multiagent architecture. One of the key features of IDEA is that all the agents share the same knowledge-base, that is, they all have the same model of the underlying autonomous system. Therefore, there is the guarantee that when two agents, for instance a planner agent and an executive agent, communicate, they do speak the same language with the same semantic. A second feature of this architecture is that more agents with similar functionalities can be used. For instance, instead of having a single planner agent, the architecture could accommodate for one or more deliberative planners and for one or more reactive planners; at execution time, depending on pre-defined triggering conditions, one of the planners is invoked. Finally, uncoupling functionalities and embedding them into software agents allow reusability of specialized agents. The IDEA architecture is an example of a multiagent architecture for a single autonomous system. In this sense, it is similar to the multiagent architecture proposed in this paper for the Palamede satellite. However, by associating agents to subsystems (devices), our architecture differs from that of IDEA, which associates agents to layers or functionalities (e.g., some agents are planners and some others are executors). Moreover, while IDEA has been developed as an alternative to the three-layered architecture, we use this structure for our agents.

Multiagent architectures are usually employed for integrating different autonomous systems, that is, there is a one-to-one mapping between autonomous systems and agents. For instance, [29] presents a control architecture for behavior-based situated teams of robots called ALLIANCE. In this multiagent architecture, robots execute missions composed of sub-tasks that are accomplished to the maximum possible extent, also in case of single-point failures. The robots are heterogeneous and have specialized skills; the same task can be achieved by different robots, but a robot cannot achieve all tasks. Therefore, tasks have to be assigned to robots. The robot team is tolerant to communication loss, which is managed by two mechanisms: (i) periodic broadcast of current robot activities and (ii) maximum period of time without communicating with a teammate after which the teammate is considered lost. An interesting property of this control archi-
ture is the lack of any beforehand knowledge of the
others, which allows the addition or removal of
teammates at any time. The ALLIANCE architecture
has been implemented in real robots performing lab-

oratory experiments.

Another multiagent architecture, which focuses
on communication amongst several autonomous sys-
tems, is the distributed Multi-Agent Reasoning Sys-
tem (dMARS) [11], that is based on the Belief-
Desire-Intention paradigm. In dMARS, plans are pre-
programmed and stored into a plan library. Finally,
in [9] a multiagent system has been built around a
planning level that uses the Hierarchical Task Network
paradigm. Our approach (especially for scheduling) is
similar to the constraint-based interval approach of [9]
with the difference that, in that work, the variables rep-
resent both the starting and the ending time of the ac-
tivities, while we represent the starting time and the
duration.

3. The Palamede satellite

The Palamede project [3], started in 1997 at the De-
partment of Aerospace Engineering of the Politecnico
di Milano, aims at giving to the students the possibility
to participate in the design, development, and imple-
mentation of the micro-satellite Palamede. Palamede is
designed as a cube-shaped three-axis stabilized satel-
lite with sides of 400 mm. The satellite is supposed
to orbit around the Earth at a height of approximately
650 km, so one orbit lasts 97 minutes of which 30 min-
utes are the eclipse period.

The payload of Palamede is basically the EarthSur-
face Imagining System (ESIS), which is composed of
a CCD camera, that will take pictures of the Earth sur-
face, and of a Global Positioning System (GPS) re-
ceiver, that will determine the position of the satellite
when a picture is taken. The onboard electrical energy
is provided by five body-mounted solar arrays and a
Li-Ion battery assembly. By applying a semi-regulated
bus system topology, the nominal mode power is pro-
vided by the solar arrays; whenever extra power is
available, it is used for recharging batteries, nominally
discharged during eclipses [32]. The main objectives
of Palamede are the collection of telemetry data about
the behavior of the Top Solar Array and the transmis-
sion down to Earth of the images taken by the cam-
era, with a dedicated frame grabber, and stored into the
onboard computer. The pictures will not be immedi-
ately downloaded to the ground station, but they will
be preliminarily analyzed in order to roughly evaluate

their quality and decide whether they are worth to be
downloaded [5].

The telecommunication subsystem is based on a ra-
dio modem, linked with the ORBCOMM constella-
tion, which provides a constant contact with the ground
station (with a delay of about 20 minutes) in order to
send housekeeping telemetry and scientific data (pic-
tures) and to receive telecommands. This choice allows
to use, as ground station, a general-purpose computer
connected with the Internet instead of a dedicated sys-
tem, because all communications will be performed by
email [12].

Finally, the ADCS (Attitude Determination and
Control Subsystem) provides both the determination
of the satellite orientation, by exploiting six sun sensors
and a magnetometer, and the control of the satellite, by
using three magnetic torquers [4].

4. Overview of the proposed multiagent
architecture

The most challenging issue in our work has been
studying the possibility of developing a multiagent sys-

tem for the management of a single completely au-

nomous spacecraft. The two main features of our
multiagent architecture are:

- Each agent is associated to a subsystem (de-
vice) of Palamede. Our multiagent system in-
cludes the ESIS agent, which manages the ac-

tivities of taking pictures using the Earth Sur-
face Imaging System, the ADCS agent, which
controls the Attitude Determination and Control
Subsystem of the satellite, and the ORBCOMM
agent, which manages the communication flows
between Palamede and the ground station via the
ORBCOMM transponder. To take into account
the electrical power consumption in managing the
activities of the satellite, a manager agent is as-

sociated to this depletable resource: the Battery
agent.

- All the agents share the same three-layered ar-
chitecture, as often encountered in space appli-
cations [28] and, more generally, in agent applica-
tions [17]. Each agent is organized in three layers,
or modules: planner, scheduler, and executor. In
general, a module interacts only with the adjacent
modules in the vertical layered structure. Ideally,
the corresponding (i.e., at the same layer) mod-
ules of different agents create a distributed plan-
In general, the activities of the agents onboard of Palamede are subject to temporal and resource constraints. These constraints can be global or local. Global constraints involve activities performed by different agents, while local constraints involve activities performed by a single agent. Temporal constraints determine the temporal ordering in which the activities should be performed. Temporal constraints can be local or global. The temporal constraints we considered in modeling the relations between activities of Palamede are reported in Fig. 2. In general, a temporal constraint involves two activities \( a \) and \( b \). The overlap constraint means that activity \( a \) must overlap activity \( b \) (i.e., \( a \) must start after \( b \) has started and must end before \( b \) ends). For example, in order to take a picture, the camera should be on during the grabbing of a picture, so \( \text{Grab overlap Camera On} \). Conversely, not overlap means that activity \( a \) must end before activity \( b \) starts or must start after \( b \) has ended. \( SAE \) means that activity \( a \) must Start After the End of activity \( b \). \( SBS \) means that activity \( a \) must Start Before the Start of activity \( b \). Note that, from Fig. 2, many temporal constraints are local and only some are global (for exam-
Resource constraints determine when it is possible to perform the activities, according to the available on-board resources. The only resource we consider in this paper is the electrical energy provided by batteries and solar arrays. The resource constraints trivially impose that the power demand should not exceed the power availability. Resource constraints are only global, because electrical power is needed by the activities of all the agents.

Some further comments are worth to illustrate the role of the Battery agent in connection to resource constraints. We decided to introduce the Battery agent to handle the aggregate demand of electrical power and to check if it satisfies the resource constraints. Our design choice to use a manager agent (i.e., the Battery agent) for the electrical energy resource has several advantages. A manager agent is the only entity that manages the physical resource and can be viewed as a wrapper that allows a better management of the information flows from and to the resource, hiding its real status and its intrinsic complexity to the other agents. Moreover, a manager agent could perform other specific tasks, such as monitoring, management, forecasting, and trend analysis. Note that, thanks to the modularity of our multiagent approach, other manager agents can be used for other resources of the satellite (including the much less critical onboard memory and processing power), like the Battery agent is used for the electrical energy resource.

The three activities of the Battery agent, namely Charge, Eclipse Discharge, and Discharge (see Table 1), require some comment. We decided to introduce the charge and discharge activities to model the electrical power subsystem in the distributed planner and scheduler. These activities do not correspond to any “physical” operation to be executed, but their introduction makes it easier to plan and schedule consistently with the electrical resource constraints. The executor module of the Battery agent (see Fig. 1) is implemented as an electronic device that automatically charges and discharges the onboard battery according to the current amount of power provided by the solar arrays and to current amount of power requested by the satellite equipment (see Section 3). In this sense, the executor module of the Battery agent is independent from the corresponding planner and scheduler mod-
ules. These modules plan and schedule the “ghost” charge and discharge activities to ensure that the requested power is less than the available power (that produced by the solar arrays plus that stored in the battery), namely that the electrical resource constraints are satisfied. For instance, the planner and scheduler modules of the Battery agent ensure that when an activity requesting more power than that provided by the solar arrays needs to be performed (e.g., a Transmit message), a corresponding Discharge activity is performed at the same time.

The main advantages of our approach are: robustness, reactivity, parallelism, and easy management of shared resources. Robustness and reactivity are guaranteed both by the layered architecture for agents and by the presence of different agents associated to different devices. This allows to manage faults and unexpected situations in a modular way, by dealing with them in the appropriate place, without affecting the working of the rest of the system. This issue will be discussed in Section 8. Parallelism is provided by the possibility to run agents on different processors. In this sense, our approach fits well with the possibility of having a processor in each device of the satellite, a situation that is envisaged by the technological evolution.

The easy management of shared resources is exemplified by the management of the electrical energy discussed above. The main drawbacks of our approach lie in the complexity of the interactions between the different modules and the different agents.

In the following sections we discuss in detail the distributed planner, scheduler, and executor, respectively.

5. The distributed planner

This section presents the planner layer of the architecture in Fig. 1. Although in principle our architecture can accommodate for fully distributed planners, given the limited complexity of the planning domain of Palamede, we have implemented a planner that exploits distributed knowledge to centrally build plans that are executed in a distributed fashion, as it is sometimes used in multiagent systems [37, Section 3.5].

The agent that is in charge of executing the centralized planning algorithm is chosen with a leader election algorithm. Note that, in order to better understand our contribution, a clear distinction should be made between distributed architecture and distributed algorithms. The distributed architecture we propose is connected to the “locality” of the activities. For example, the details about the activities of the ESIS agent (e.g., about their number, names, execution, ...) are known only to that agent and not to the other agents. On the other hand, algorithms run on a distributed architecture could be centralized or distributed. We adopted the first option for planning and, as shown in the following sections, the second option for scheduling and execution (in this last case, the distribution is pushed to its limit, with the executor modules running in parallel, without communicating with each other).

The planner module of an agent can fulfill requests for new plans coming from its own scheduler, from a planner module of another agent, or from a user. The request for a new plan triggers the execution of a leader election algorithm, which elects the agent in charge of managing the planning. The planner module elected as leader firstly collects the knowledge from other modules, secondly runs the planning algorithm, and finally broadcasts the new plan to other agents (and to the scheduler module of its agent). The planner modules of the other agents are in charge to fulfill the requests of the leader, to receive the new plan, and to forward it to their own schedulers.

The next sections present the leader election and the planning algorithms.

5.1. The leader election algorithm

The election problem, also called leader finding, is to start from a configuration where all modules have the same state, and arrive to a configuration where exactly one module is the leader and all others are not. In our application, the agent that is the leader is in charge of performing planning.

We use an extinction mechanism applied to the echo (wave) algorithm [35]. Informally, a wave algorithm exchanges a finite number of messages and elects a leader (in general, makes a decision) that depends causally on some event in each agent. Let us suppose that agents have unique identifiers, totally ordered. As in all distributed algorithms, there is a distinction between initiators and non-initiators, also called followers. An agent is an initiator if it starts the execution of its local election algorithm triggered by some internal condition. The initiator starts a new wave of messages. An agent is a follower if it triggers its local election algorithm after receiving a message from the initiator. When a follower receives a wave message for the first time, it broadcasts the message to all of its neighbors and stores the sender as leader. When a follower receives another wave message, it decides whether keeping on echoing wave messages by com-
The core idea of temporal networks is to represent temporal relations using a graph, where nodes are time events and arcs between pairs of nodes denote the minimal and maximum delay which can occur between events [10]. Therefore, temporal networks contain both relative (i.e., precedence) and metric (i.e., duration) information. A temporal network can be associated to a temporal constraint satisfaction problem, where each variable represents a time point, and unary and binary constraints represent metric and relative information amongst time variables. A special kind of temporal networks, called Simple Temporal Networks (STNs), are those where each constraint has only one possible time interval (no disjunctions are allowed). The consistency of a STN can be checked in polynomial time using any all-pairs-shortest-paths algorithm from classical graph theory, or using a path consistency algorithm from the constraint satisfaction theory. The main limitation of STNs is the lack of expressivity to model constraints such as a before b or a after b, that is, disjunctions on temporal intervals. Checking consistency for general temporal networks with more than one interval associated to constraints (i.e., with disjunctions on possible intervals) is NP-hard. Another limitation of STNs is the lack of support to temporal constraints involving resources. The motivation for embedding STNs into our planner is to enable the generation of temporally flexible plans and was inspired by the Kirk planner [22]. The temporal flexibility of the plans is exploited by the underlying distributed scheduler described in the next section. This is different from [22], which exploits the temporal flexibility directly into the executor.

Basically, Palamede can achieve three tasks. Two of these tasks are primitives: transmit housekeeping (that is performed by an operator that exploits the ORBCOMM On and Transmit HK activities) and transmit photo (that is performed by an operator that exploits the ORBCOMM On and Transmit Photo activities). The third task is non-primitive: take-and-transmit photo to the ground station. In our algorithm, a task is primitive if and only if it can be executed by a single agent. Hence, the last task, namely having a picture at the ground station, involves the ESIS, ADCS, and ORBCOMM agents and therefore is not primitive; instead, the housekeeping transmission is primitive since it can be achieved by only the ORBCOMM agent. These tasks are mapped into the STNs used as basic entities by our planning algorithm. Note that this is the knowledge collected by the planner module elected as leader before starting the plan-

5.2. The planning algorithm

The actions of the planning domain of Palamede are given in Table 1. Since this domain is extremely limited, we have devised a hand-tailored planner. Hand-tailored planners are between domain-independent and domain-dependent planners: they have a set of properties and algorithms independent of the planning domain, but they leave room to heuristics derived for a specific domain. Hand-tailored planners are largely used in practice, since they offer a good trade-off between efficiency and generality. Our planning algorithm combines ideas from HTN planners and STNs that are briefly described in the following.

The Hierarchical Task Network, or HTN, is an approach to automated planning in which the dependency among actions is given in form of networks [30, Chapter 12]. The objective of an HTN planner is to produce a sequence of actions that perform some task. The description of a planning domain includes a set of operators and a set of methods, each of which is a prescription for how to decompose a task into sub-tasks. Given a planning domain, the description of a planning problem contains an initial state and a partially ordered set of tasks to accomplish. Planning proceeds by using the methods to decompose tasks recursively into smaller and smaller sub-tasks, until the planner reaches primitive tasks that can be performed directly using the planning operators. For each non-primitive task, the planner chooses an applicable method and instantiates it to decompose the task into sub-tasks. If the plan later on turns out to be infeasible, the planning system backtracks and tries other methods. HTN planners are hand-tailorable: their planning engines are domain-independent, but the HTN methods and operators may be domain specific.

The actions of the planning domain of Palamede are given in Table 1. Since this domain is extremely limited, we have devised a hand-tailored planner. Hand-tailored planners are between domain-independent and domain-dependent planners: they have a set of properties and algorithms independent of the planning domain, but they leave room to heuristics derived for a specific domain. Hand-tailored planners are largely used in practice, since they offer a good trade-off between efficiency and generality. Our planning algorithm combines ideas from HTN planners and STNs that are briefly described in the following.

The Hierarchical Task Network, or HTN, is an approach to automated planning in which the dependency among actions is given in form of networks [30, Chapter 12]. The objective of an HTN planner is to produce a sequence of actions that perform some task. The description of a planning domain includes a set of operators and a set of methods, each of which is a prescription for how to decompose a task into sub-tasks. Given a planning domain, the description of a planning problem contains an initial state and a partially ordered set of tasks to accomplish. Planning proceeds by using the methods to decompose tasks recursively into smaller and smaller sub-tasks, until the planner reaches primitive tasks that can be performed directly using the planning operators. For each non-primitive task, the planner chooses an applicable method and instantiates it to decompose the task into sub-tasks. If the plan later on turns out to be infeasible, the planning system backtracks and tries other methods. HTN planners are hand-tailorable: their planning engines are domain-independent, but the HTN methods and operators may be domain specific.
Fig. 3. The take-and-transmit photo task: graph of temporal relations among composing activities; basically this is a reformulation of part of Fig. 2; in this and in the next figure, red arcs represent global temporal constraints while black arcs represent local temporal constraints.

Fig. 4. The take-and-transmit photo task: mapping the graph of Fig. 3 into a STN.

Algorithm 1 gives the pseudo-code of our temporal planning algorithm: it takes as input a set of goals (tasks) \( G_S \) and a time horizon \( T \) over which the tasks must be achieved, and it gives as output a course of actions \( P \) with temporal domains on their start and end time events. The goals might have static priorities, that give the order of goal extraction (step 4), namely \( G_S \) is a priority queue. If a primitive task is selected, the corresponding STN is created (steps 5–6). Since the
Algorithm 1 Build a temporal plan

Input: a set of goals (tasks) \( gs \), a time horizon \( T \)
Output: a plan \( p \)

1: \( p \leftarrow \) empty
2: \( G \leftarrow \) create the STN over \( T \) mapping the plan \( p \)
3: while \( gs \) is not empty do
4: \( g \leftarrow \) extract goal (task) from \( gs \)
5: if \( g \) is primitive then
6: \( stn \leftarrow \) create a STN achieving goal (task) \( g \)
7: \( \text{if not insert}(stn, G, p) \) then
8: discard \( g \)
9: end if
10: else
11: \( \text{newgoals} \leftarrow \) decompose \( g \) in sub-goals (sub-tasks)
12: \( gs \leftarrow gs \cup \{\text{newgoals}\} \)
13: end if
14: end while
15: return \( p \) (represented by \( G \))

Palamede domain is simple, there is a one-to-one mapping between tasks and primitive STNs, that is, there is a single operator achieving the given primitive task. Once the STN is created, the procedure insert is called (step 7). This procedure (Algorithm 2) computes all the possible positions where the new \( stn \) could be inserted in the current plan \( G \) and loops until a valid position for \( stn \) in \( G \) is found or all positions have been considered. A position is valid if and only if it maintains the consistency of the STN \( G \) representing the whole plan.

If the task selected at step 4 is not primitive, it is decomposed in other sub-goals (sub-tasks) that are added to the set of goals. During the decomposition, constraints among the sub-tasks are created and associated to the tasks. In this manner, when a decomposed task is considered for insertion (by procedure insert) the planner is able of adding also the intra-task (intra-agent) constraints.

The planner as described here is not complete, hence it is not guaranteed to find a plan when it exists. However, given the limited domain of Palamede, a plan has been always found in our experimental activity, starting from realistic sets of goal tasks. For example, given the following realistic goal tasks: 1 transmit housekeeping after and before every eclipse and at most 3 take-and-transmit photos to ground station, our planner is able to plan all the housekeeping transmissions and two take-and-transmit photos. This is a remarkable result, since the manual planning of activities on Palamede resulted in two housekeeping transmissions and only a single take-and-transmit photo per orbit. Other experimental results are reported in Section 8.1.

6. The distributed scheduler

The input to the distributed scheduler is the output of the distributed planner and consists of a set of activities that should be scheduled and, for each activity, in a domain set that contains the possible starting times of the activity that do not violate any temporal constraint. Differently from the planner, which first collects the distributed knowledge and then runs a centralized planning algorithm, the scheduler runs a completely distributed algorithm. The motivation for having a distributed scheduling algorithm is to allow every agent to locally schedule its activities and manage contingencies to a certain degree of autonomy. While the planner considers only some temporal constraints, the scheduler considers also resource constraints. More precisely, the planner considers constraints that can be defined at design time, whereas the scheduler considers constraints that are determined by the contingent situation of the agents.

We formulated the scheduling problem as a Constraint Satisfaction Problem (CSP) [24], following a well-established practice [34]. Actually, the problem is a Distributed CSP (DCSP) [38], because both the activities and their constraints are distributed over a set of agents. More precisely:

- there are \( n \) agents, in our case \( n = 4 \): ESIS agent, ADCS agent, ORBCOMM agent, and Battery agent;
– there are $n$ sets of activities to schedule $A_1$, $A_2$, …, $A_j$; each $A_i$ contains the activities of agent $i$ (Table 1);
– each activity $j$ is associated to a variable of the DCSP representing its starting time, whose time values are taken from a finite domain set $H_j$ (the scheduling horizon); an activity is characterized by its duration and its power consumption (Table 1);
– an agent knows all the temporal constraints (both local and global) relevant to its variables; the (global) resource constraints are known only by the Battery agent.

In our case, the scheduling horizon lasts two orbits (194 minutes), divided in time quanta of 30 seconds.

Generally speaking, this means that the domain set $H_j$ of a variable (from which the starting time of the corresponding activity is selected) is the set of integers from 0 to 387. In practice, a domain set $H_j$ contains less elements because the planner leaves in $H_j$ only the starting times of activity $j$ that are consistent with the possible starting times of other activities. A solution for the DCSP is an assignment of values to all the variables (i.e., starting times of activities) such that all the temporal and resource constraints are satisfied.

In our distributed scheduler, the scheduling of the activities is performed at two levels: at the local level, in which each agent (actually, its scheduler module) determines the sequence of its local activities satisfying its known constraints, and at the global level, in which the local schedules are harmonized to ensure the consistent consumption of shared resources. In this way, the scheduling problem is a DCSP that involves $n$ CSPs local to the agents. The choice of using local CSPs allows agents to handle multiple local variables in an efficient way, when playing on the DCSP stage. Basically, we transform a problem of exponential complexity into a set of still exponential, but smaller, problems, thus improving worst-case performances. In the following sections, we discuss our local and global scheduling algorithms, respectively.

6.1. The local dynamic backtracking algorithm

At the local level, each agent uses a CSP algorithm called dynamic backtracking [16]. Using this algorithm, an agent finds value assignments to its local variables that satisfy its local constraints; these assignments form a local schedule. More precisely, a local schedule of agent $i$ is a vector of values assigned to the variables of $i$, representing the starting times of the activities in $A_i$. For example, a local schedule for ADCS agent can be $<$ Read Attitude $= 12$, Control Attitude $= 13$ $>$, with the meaning that the activities Read Attitude and Control Attitude start at 12th time quantum (360th second) and at 13th time quantum (390th second), respectively. Note that a local schedule satisfies the local constraints of an agent; in this case Control Attitude starts after the end of Read Attitude (Fig. 2).

We decided to use the dynamic backtracking algorithm because it has been proved to be very efficient, thanks to the use of the Elimination Explanation (EE) matrix [16,36]. For every activity of an agent, the EE matrix registers the time positions that are not allowed for the activity because constrained by other activities, with an indication of these constraining activities in order to backjump to them and change their assignments, if necessary. The local CSP algorithm iteratively picks up at random an activity, creates its EE matrix, and checks if allowable positions are present: if they are, the time position with the maximum level of available resource is chosen (see Section 6.2 below). This heuristic is called maximum availability, but other heuristics could be used as well, such as the least-constraining one. If a consistent time position cannot be found, the CSP algorithm backjumps to the scheduling of the activity responsible of the flaw by analyzing the EE matrix. Ties are randomly broken.

6.2. The global asynchronous weak commitment algorithm

The local schedules found at the local level are harmonized at the global level using a variation of a DCSP algorithm, called Asynchronous Weak Commitment (AWC) [38]. In the original algorithm, each agent chooses a local schedule (i.e., a “legal” assignment of values to its variables or, equivalently, an assignment of starting times to its activities that satisfies local constraints) and exchanges this information with the other agents, sending asynchronous $ok?(s)$ messages, where $s$ is a local schedule. Each agent stores the information about local schedules of other agents, received via $ok?$ messages, into a local data structure called agent_view that, due to message delays, may contain obsolete information. Each agent has a priority value. When an agent cannot find any local schedule that is consistent with the local schedules of higher priority agents stored in the agent_view, the agent generates a new global constraint, called nogood. After generating a
nogood, the agent increases its priority value and sends
its local schedule to the other agents. In the original
formulation of the algorithm, a heuristic called min-
conflict is proposed for choosing a new local schedule
for an agent. Under some assumptions about the reliabil-
ity of communication, the AWC algorithm is com-
plete: it always terminates, either by finding a solution
(if it exists) or by stating that a solution does not exist.

The main issues of our version of the AWC algo-

rithm are illustrated in the following. With a global
constraining entity, like the electrical energy resource,
a manager agent checks the satisfaction of the resource
constraints. This manager, the Battery agent, receives
all the local schedules and checks if they are consist-
ent with the resource global constraints. Whenever
all the local schedules are consistent with these con-
straints, the Battery agent broadcasts a message to let
the agents stop their computations. Note that the Bat-
tery agent does produce its local schedule (involving
the Charge and Discharge activities, as discussed
in Section 4). The role of the Battery agent as resource
manager is a specialization of the original AWC algo-
rithm. An innovation we introduced is that, initially,
the Battery agent estimates the resource availability
over a (future) two-orbit time window. More precisely,
it determines the amount of electrical power that is es-
timated to be available at every time quantum in this
window. Then, it transmits this information to the other
agents before they start local scheduling. In this way,
we facilitate finding a solution because an agent knows
the (expected) availability of resources and can build
its local schedule accordingly. However, the agent does
not know exactly how the resources are going to be
used by the other agents (recall that information in
agent_view may be obsolete). As said, the global consis-
tency on the resource usage is enforced by the Bat-
tery agent. We can say that the agents know relaxed
resource constraints to drive their local schedules to a so-
lution. Another innovation we introduced with respect
to the original AWC algorithm is that agents use local
CSP algorithms (Section 6.1) instead of min-conflict
heuristic to find their local schedules.

We now illustrate in detail how our modified AWC
algorithm works. The instantiation order of agents cre-
ates the initial priority values required by the AWC al-
gorithm. The basic procedures executed by a generic
agent in our version of the AWC algorithm are de-
scribed in the following.

Initialization. Each agent receives the relaxed re-
source constraints from the Battery agent and gener-
ates a local schedule, that is sent to lower priority
agents and to the Battery agent via ok? messages.

Receiving ok? Upon receiving an ok?(s) mes-
sage from another agent, an agent inserts s in its
agent_view. This interaction protocol guarantees
that the asynchronous modifications of an agent’s
agent_view can take place only upon the arrival (via
ok? messages) of local schedules of other agents. After
updating its agent_view, an agent checks for its con-
sistency.

Checking agent_view. Each agent evaluates the
consistency of its local schedule with respect to its
agent_view and to the constraints it knows. At this
point, the agent can face two situations:

- The current local schedule and the local sched-
ules in the agent_view are consistent with all the
global temporal constraints known by the agent,
so the agent does not need to modify its current
local schedule.

- The current local schedule and the local schedules
in the agent_view are not consistent, namely they
violate some of the global temporal constraints
known by the agent. In this case, the agent needs
to modify its current local schedule by invoking
again its local CSP algorithm. If the new local
schedule is consistent with the agent_view, it is
sent to the agents with lower priority and to the
Battery agent. When a consistent local schedule
cannot be found (nogood situation), the agent re-
laxes the constraints in two ways and generates
a new local schedule. The first relaxation has the
effect that, when the agent checks the consistency
of the schedule, it tolerates small extra resource
consumption beyond the limit it knows (the re-
laxed resource constraint). With the second relax-
ation, the agent does not take into account global
temporal constraints with other agents’ activities.

The new local schedule may contain some flaws; how-
ever, this relaxation allows the agent to find
a local schedule to which other agents can easily
adapt. The agent increases its priority and sends
the new local schedule and a new priority value to
all the other agents, including the Battery agent.
The new priority value is calculated in order to be
larger than that of the highest priority agent with
conflicting local schedules. In this way, the other
agents are “forced” to adapt to the local schedule
of the agent.

At the end of the execution of the above scheduling
algorithm, the local schedules of the agents are har-
monized, namely they satisfy all the temporal and re-
source constraints. Note that, in the original AWC al-
algorithm, the nogood messages are recorded to guar-
antee the completeness of the algorithm by preventing
loops. In our implementation of the AWC algorithm,
we do not store the nogood messages. Actually, the
nogood situations are even not communicated to the
other agents; in this way, we can save computational
resources (that are scarce onboard the satellite). Not
communicating the nogood situations, the global com-
pleteness of our version of the AWC algorithm can-
not be guaranteed, because the algorithm can enter a
loop and can repeatedly generate the same local sched-
ules. In practice, in our application, the number of pos-
sible local value assignments (local schedules) is so
large that the probability of entering a loop is very
small. In our implementation, we introduced a timeout
as a heuristic that allows avoiding useless computation
when a solution does not exist. The use of a timeout is
thus justified by the fact that our version of the AWC
algorithm is not complete, since we do not communi-
cate nogood messages.

We decided to utilize the AWC algorithm instead of
other possibilities, such as Asynchronous Backtrack-
ing (AB) or Distributed Breakout (DB), because it is
one of the most efficient algorithms for solving DC-
SPs. As an example, AWC outperforms AB because
it can revise a bad decision without exhaustive search,
but by changing the priority order of agents dynami-
cally [24]. On the other side, DB is more efficient than
the AWC search when problem instances are critically
difficult, because DB does an intensive analysis of all
the possibilities of each agent before changing its val-
ues [38]. The system proposed in this paper automati-
cally does this local work by using the local CSP algo-
rum, so DB is unnecessary. AWC could be partially
centralized by introducing an agent that receives
and evaluate the nogood messages. Although this so-
lution fits well with the proposed architecture, we have
not implemented it.

7. The distributed executor

The main task of an executor module is to actually
execute actions by sending commands to the underly-
ing hardware. In our system, each executor module –
recall that there is an executor module for each agent –
is structured as shown in Fig. 5.

The figure represents the interaction between the
scheduler and the executor modules of an agent and
the monitor embedded in the executor. The execu-
tor receives from the scheduler the local schedule to
be executed. This local schedule is processed by a
component, the Low Level Scheduler, with the pur-
pose to generate, for every activity in the schedule,
an item to be inserted into the Task Agenda (in a
way similar to RAPs [15]). The Task Agenda is a
data structure containing data on activities to be per-
formed. These data are completed by those stored in
the Task Properties database, which contains all the
information about activities, such as their preconditions,
their effects and, for activities not directly executable
by the hardware, the different ways in which they
can be decomposed in more specific sub-activities.
For example, a Control Attitude activity has
no preconditions (it can be always executed) and
has the effect of getting the angular velocity around
the three axes of the satellite smaller than $2^\circ/s$.
A Control Attitude activity cannot be directly
executed by the hardware and must be split in
three sub-activities that are executed in sequence:
Magnetotorques On, Magnetotorques
Control, and Magnetotorques Off. (Control
Attitude can be decomposed only in this way; in
general, multiple decompositions can be possible for
an activity.) All these data are summarized in the fol-
lowing entry for Control Attitude in the Task
Property database:

```xml
<CONTROL_ATTITUDE_ACTIVITY>
  <task-effects>
    <effect>OMEGA_X <; 2</effect>
    <effect>OMEGA_Z <; 2</effect>
  </task-effects>
  <task-decomposition>
    <min-time>15</min-time>
    <max-time>45</max-time>
  </task-effects>
  <task-decomposition>
    <condition>
      after end MAGNETOTORQUES_ON
    </condition>
    <name>MAGNETOTORQUERS_OFF</name>
  </task>
  <name>MAGNETOTORQUERS_CONTROL</name>
  <condition>
    after end MAGNETOTORQUERS_ON
  </condition>
</task>
</CONTROL_ATTITUDE_ACTIVITY>
```

Note that the above entry contains also some informa-
tion about the expected execution time of the decom-
position (discussed later). Items in the Task Agenda
are sorted according to the time the activities they re-
fer to should be executed. In such a way it is trivial to
search the Task Agenda for the next activity to execute.
However, it is not possible to associate to all activities a precise (absolute) starting time, since there are some activities that, to be performed, must wait the end of some other activities (for example, when two activities have a Start After End constraint, see Section 4). In this case, the former activities are placed in the Task Agenda after the latter ones.

The Agenda Monitor has the purpose to monitor the content of the Task Agenda and to look for the next activity to execute, namely for the first activity according to the sorting in the Task Agenda. If this activity has a precise (absolute) starting time, the evaluation of whether its execution must start is straightforward for the Agenda Monitor: it is sufficient to compare the starting time with the current time. Otherwise, if the execution of the first activity in the Task Agenda depends from a number of conditions that must be satisfied, it is necessary to evaluate them. For example, a Grab requires that a Camera On is started and not yet ended. Checking whether these conditions hold is up to the Agenda Monitor. If at least one condition is not verified, a failure is raised and the situation is notified to the Recovery component to be handled appropriately. We will come back to recovery later.

Verifying preconditions and effects for a given activity requires the executor module to know the state of the subsystem under its control. For this purpose, the executor interacts with the monitor, which queries sensors to acquire low level information (such as voltage or current values) and translates it in state variables values (such as CAMERA_STATUS=ON), that are used by all the components of the executor. So, our executor is completely procedural and does not perform any reasoning about the state of the system.

This is one of the main characteristics that make our executor different from many other execution archi-
The Freezer Handler is the manager of the Freezer, that is a database storing a forest, where every tree is associated to an activity of the local schedule under execution. The name Freezer refers to the fact it “freezes” the state of the system providing an overview of the current execution situation, allowing to know which activities have been completed, which ones are under execution, and which ones are waiting to be performed. The Freezer, for each activity non directly executable by the hardware, tracks the way it has been decomposed in sub-activities, so it is always possible to estimate the time still needed to accomplish a portion of the local schedule. This time depends, in a recursive manner, on the execution time of all sub-activities not yet concluded.

A problem the executor must face is monitoring whether activities can be completed within the temporal constraints imposed by the scheduler. This is the job of a component called Activities End Time Checker, that receives the time by which each activity should be concluded from the Low Level Scheduler. The Activities End Time Checker checks the items stored in the Task Agenda and in the Freezer. If it finds any item concerning an activity not yet accomplished, but that should have been already finished, a failure called end-time-bound failure is raised. It is up to the Recovery component to handle it.

Our executor can deal with many failures, including:

- Unsatisfied preconditions: if an activity has to be executed, but its preconditions do not hold (e.g., due to hardware failures), then the executor suspends that activity and all other activities whose execution depends on it. The activities are then reactivated as soon as the failure is no more diagnosed. When the unsatisfied preconditions correspond to the effects (for instance, the executor tries to turn on the camera, but it is already on), the activity is not performed, because the system is already in the desired state.

- Unsatisfied effects for an executed activity (for example it has been commanded to the camera to turn on, but this did not happen): in this case the executor re-executes the activity. That is a simple and widely used recovery method in autonomous systems.

- End-time-bound failure: this kind of failure is raised by the Activities End Time Checker to notify the Recovery module that the temporal window the scheduler allocated for a given activity is expired.

For handling an end-time-bound failure, the Recovery component asks the scheduler for some more time to accomplish the activity that generated the failure. Estimating the time still necessary to conclude the activity is performed by the Remaining Time Checker. This component returns two values, $m$ and $M$, representing the time still necessary to carry out the activity in the best and in the worst case, respectively. To estimate the two values, the Remaining Time Checker interacts with the Freezer Handler to get a reference to the Freezer tree concerning the activity it is interested in, so it is possible to know which sub-activities have been accomplished and which have yet to be executed. For each activity directly executable by the hardware, the minimum $m$ and maximum $M$ time necessary for its execution are stored into the item of the Task Property database associated to it. For other activities, $m$ is obtained by summing up the minimum times requested by their sub-activities; similarly $M$ is given by the sum of all the maximum times requested by these sub-activities. Once the Remaining Time Checker returns to the Recovery component the two values $m$ and $M$, these values and the activity name are sent to the scheduler module (described in Section 6), which executes a consistency test on the local schedule by simply enlarging the time window of the given activity to $M$. If the test is not successful, the scheduler tries to allocate to the activity a shorter and shorter time window until it reaches $m$. If a consistent schedule is found before reaching $m$, the new local schedule is sent to the Low Level Scheduler and all the sub-activities of the failed activity can continue their execution as before. Otherwise, if it is not possible to extend the time window for the failed activity, the execution of the entire schedule must be aborted and both the Task Agenda and the Freezer cleared. At this point, a recovery strategy can work as follows. The scheduler starts a new scheduling with the purpose to allocate on the timeline all the activities not yet accomplished, including that just failed. In computing a new schedule the scheduler tries first to solve the problem locally and, if it is not possible, it globally involves all other scheduler modules. In general, whenever a faulty
event in a subsystem makes a local schedule inconsistent, consequences upon the schedules of other agents are possible. The multiagent architecture we adopted addresses this problem exploiting decentralization: the agent managing the faulty subsystem could adjust its local schedule without requiring the other agents to change their assignments. Only when the situation cannot be recovered locally, a new global scheduling is performed. If the global scheduling process does not find any solution, a new planning process starts. This modular way of dealing with fault recovery shows the potential of our multiagent architecture.

8. Experimental evaluation

The multiagent system described in the previous sections has been implemented using the JAVA programming language along with the JADE framework [20]. The planner, scheduler, and executor modules of the agents have been implemented as JADE agents. All messages are exchanged according to the FIPA ACL standards. Plans and schedules are sent as JAVA objects in the content fields of the messages. Experiments have been performed with a computer equipped with a 500 MHz processor and 128 MB RAM. We have run all the agents on a single processor. An interesting future possibility is to run the agents in parallel on different processors (ideally, one for each device), in order to fully exploit the potential of our architecture. In the following, we present the results obtained experimenting with the distributed planner, with the distributed scheduler, and with the distributed executor and the whole system. Note that we have been more interested in testing the correctness of the system rather than in optimizing its performance.

8.1. Experimenting with the distributed planner

This experimental activity aimed at evaluating the planning algorithm with respect to the Palamede planning domain. For this evaluation, we have considered planning problems with different number of tasks on several time horizons.

Table 2 shows the results of an experimental evaluation of our planner with respect to the required computational time. The first column shows the numbers of desiderata tasks, in this example the number of take-and-transmit photos, the other columns report the number of actually planned tasks and the computational time (in seconds) over a time horizon of respectively one, two, and four orbits. All the problems have been solved within few seconds. The maximum number of tasks that ideally can be performed in a single orbit is 6, so the planner is not able to plan all the 10 tasks in the first column. We remark how the computational time seems to scale well with the number of orbits. Note that the planning of the same number of tasks takes longer when performed for four orbits than when performed for two orbits (or for one orbit), because the possible positions of the activities are less constrained in the former case, leaving the planner more possibilities to explore.

<table>
<thead>
<tr>
<th>Number of</th>
<th>1 orbit</th>
<th>2 orbits</th>
<th>4 orbits</th>
</tr>
</thead>
<tbody>
<tr>
<td>tasks</td>
<td>Tasks</td>
<td>Time [s]</td>
<td>Tasks</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0.6</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>1.6</td>
<td>10</td>
</tr>
</tbody>
</table>

8.2. Experimenting with the distributed scheduler

In experimentally evaluating our distributed scheduler, the first question we answer is if using distributed scheduling on a single spacecraft leads to dramatic decrease of solution quality and increase of computational time. The experiments show that this is not the case. We initially considered a single agent to schedule \( h \) dummy activities with a schedule horizon of two orbits. All the dummy activities do not require any energy to be performed and have the same duration of \( T = 5 \) time quanta (recall that a time quantum is 30 seconds). The dummy activities are required not to overlap each other by means of the overlap constraint. Of course, given \( T \) and the schedule horizon, the larger \( h \), the more difficult the scheduling problem, and the more difficult to find a consistent local schedule within the timeout. Figure 6 shows the performance of a centralized scheduler facing the problem of allocating an increasing number of not overlapping dummy activities. Each point is the average over 100 trials. Variations between different trials are due to random choices in selecting the variable to instantiate, the value for a variable, and the activity to backjump to in the local CSP algorithm. A timeout of 3 minutes has been given to the system for solving each trial: whenever this time limit was reached the trial was stopped and declared unsuccessful. Note that 3 minutes of scheduling is a very short time when compared with the 194 minutes of the scheduling horizon. Until
We now present some realistic experiments with the distributed scheduler. We input to our multiagent scheduler a number of activities that are expected to be performed by the satellite in a temporal horizon of two orbits. In order to systematically evaluate the performances of the proposed multiagent scheduler, 192 realistic scenarios have been created by varying the number of activities to be performed: the number of periods of Camera On could be 2 or 3, the number of photos to be taken (Grab) s could be 3, 5, 7 or 9, the number of Transmit HKs could be 2 or 4, the number of Transmit Photos ranges between 2 and 5, and the number of Charges could be 4, 6 or 8. The distributed scheduler has run 10 trials for each scenario. A timeout of 3 minutes has been given to the system for solving each trial. About the 60% of the trials (1180 out of 1920) resulted in consistent sets of local schedules, namely in consistent global schedules, before the timeout. 76 scenarios out of 192 have been always solved (solutions have been found for all trials), while in 26 scenarios out of 192 a solution was never found for any trial. Note that 60% is the average success rate over all the scenarios we have considered, including, for example, that with 3 Camera Ons, 9 Grabs, 4 Transmit HKs, 5 Transmit Photos, and 8 Charges. This scenario is very complex and would probably never be experienced on the real spacecraft. Expected typical scenarios are much simpler and show higher success rates.

The analysis of the sensitivity of the system to the number of activities to be scheduled is reported in Ta-

Table 3

<table>
<thead>
<tr>
<th>Agents’ activities</th>
<th>Successes</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 [%] s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 15 15 15 98 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 15 15 10 96 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30 10 10 10 94 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35 10 10 5 89 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40 10 5 5 87 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45 5 5 5 86 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the number of activities is smaller than 60, the agent instantaneously finds a solution in all the 100 trials. From 60 to 65 activities the agent finds a solution, but with a longer average solving time. The computational limit for our local CSP algorithm is reached for \( h = 66 \), where the solution is found only in the 12% of the trials, within 90 seconds on average. From these results, 60 appears to be the maximum number of activities that a centralized scheduler can easily manage. Hence, we tried different distributions of 60 dummy activities on four agents. Table 3 shows the performances of the distributed scheduler. In fair situations, with activities evenly assigned to agents, the performances are almost the same of the single scheduling agent case. In situations with activities asymmetrically assigned to agents, the loss of successful schedules is slightly more than 10%. We can conclude that the distribution of the scheduling process does not have any dramatic impact on the performance of the system.

Fig. 6. Fraction of successes and average solving time vs. number of activities.
Table 4
Experimental results in realistic scenarios

<table>
<thead>
<tr>
<th>Activity</th>
<th>#</th>
<th>Successes [%]</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera</td>
<td>2</td>
<td>59</td>
<td>41</td>
</tr>
<tr>
<td>On</td>
<td>3</td>
<td>64</td>
<td>45</td>
</tr>
<tr>
<td>Charge</td>
<td>6</td>
<td>64</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>53</td>
<td>48</td>
</tr>
<tr>
<td>Transmit</td>
<td>2</td>
<td>73</td>
<td>40</td>
</tr>
<tr>
<td>Photo</td>
<td>3</td>
<td>68</td>
<td>42</td>
</tr>
<tr>
<td>HK</td>
<td>4</td>
<td>58</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>47</td>
<td>51</td>
</tr>
<tr>
<td>Transmit</td>
<td>2</td>
<td>70</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>50</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>98</td>
<td>22</td>
</tr>
<tr>
<td>Grab</td>
<td>5</td>
<td>78</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>57</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>12</td>
<td>92</td>
</tr>
</tbody>
</table>

Table 4. Changing the number of periods during which the camera is turned on and the number of battery charges does not change significantly neither the percentage of successful cases nor the average solving time. Similarly, the performances are not worsening dramatically by increasing the number of transmissions. This is a very important result because the activities involving transmissions are the most power consuming, thus requiring the intensive use of the satellite’s battery and imposing restrictions to other activities. This positive result demonstrates the efficiency of the local CSP algorithms and of our approach for the management of the resource that is based on a resource manager (the Battery agent). The analysis of the performances related to the number of Grab activities shows that both the fraction of failures and the average solving time increase with the number of photos required. Grab activities are characterized by the shortest duration and very low power consumption, but also by constraints with activities performed by other agents (Fig. 2). An attitude control action has to be done before grabbing and photos cannot be taken during eclipse. Grab is thus the most constrained activity, representing a very critical node in the network of relationships among activities.

Finally, in order to evaluate the impact of the value of the timeout on the system, the 26 failed realistic scenarios have been considered again with a timeout of 5 minutes (instead of 3). With 7 trials for each scenario, the scheduler with extended timeout has solved 15 out of 26 scenarios at least for one trial, proving the high sensitivity of the system to this parameter and the difficulty to set a general limit between hard and impossible scenarios (recall the discussion of Section 6.2).

8.3. Experimenting with the distributed executor and the whole multiagent system

To evaluate the global multiagent architecture proposed in this paper, we used a scenario-based approach. In particular, we considered the following scenario: the satellite is turned on by a remote station, which requires as goal to plan as much take-and-transmit photos as possible over a time horizon of 80 time quanta with an eclipse starting after 30 and lasting 10 time quanta. Once Palamede has planned and scheduled the tasks and the activities, we inject an end-time-bound failure. Running this scenario, the following messages are exchanged:

- when the satellite is turned on: it has no task to execute, and therefore every executor module sends a request to the corresponding scheduler;
- every scheduler module receives the request, but it has not any local schedule; therefore, it informs the other scheduler modules and forwards the request to its planner;
- every planner receives the request, broadcasts the request to the other planners, and initiates the election algorithm;
- the elected planner module collects the knowledge, runs the centralized planning algorithm, and broadcasts the new plan (if it finds one) to the other planners;
- the schedulers receive the new plan, run the distributed scheduling algorithm, and once the plan is scheduled they forward the local schedules to their executors.

Note that if, differently from the case presented above, the distributed scheduler cannot find a schedule consistent with both temporal and resource constraints, the planner is invoked to produce an alternative plan. After successful scheduling, while every agent is executing and monitoring the activities of its local schedule, we inject an end-time-bound failure over the frame grabber (managed by the ESIS agent). Figure 7 shows the messages exchanged between the three modules of the ESIS agent, and the planner modules of the ADCS and ORBCOMM agents. The messages are:

- the executor of the ESIS agent proposes to the corresponding scheduler to shift the end-time of
Fig. 7. Messages following an end-time-bound failure and recovery (the screenshot of the JADE Sniffer shows on the left the ID of messages; Camera agent is the ESIS agent).

Note that it is possible to define an alternative recovery procedure that, after finding that a local re-schedule does not solve the problem, tries a global re-schedule before attempting a new planning process (as it happens in the described procedure). Which procedure is better can be related to diagnosis of the fault performed by a diagnosis component that could be added to our architecture (this is an interesting possible future work).

In this scenario, the injected failure is an end-time-bound failure, but similar behaviors have been observed to occur with other types of failures, such as hardware failures (simulated by killing the agents associated to the damaged subsystems). We remark that the time required for planning and scheduling in this scenario is of few seconds. These experiments show the robustness and the modularity of the proposed multiagent architecture, that is able to address failures first locally and, when this attempt is unsuccessful, globally, involving the same or higher layers (see Fig. 8).

9. Conclusions

In this paper, we have presented a multiagent architecture for autonomously managing activities onboard a satellite. Each agent is associated to a subsystem of the Palamede satellite and manages its activities. The experimental validation of the multiagent system has been satisfactory, for example demonstrating that the performances of the distributed scheduler are compa-
rable with those of a centralized scheduler, but with the advantage of the modularity provided by the multiagent architecture.

The relevance of the proposed multiagent system in the context of the Palamede project is in the support it can provide to the engineers in planning and scheduling activities efficiently and autonomously. This is an improvement with respect to manual and ground-based planning and scheduling. For example, our planner has been able to plan two take-and-transmit-photos per orbit, one more than what has been obtained by manual planning (see Section 5.2). More generally, the work presented in this paper demonstrates in the realistic test case of the Palamede satellite that a multiagent system could be employed for controlling the onboard activities of a single spacecraft. The main advantage of the proposed approach is its modularity, namely the fact that each subsystem is managed by an agent. This allows each agent to work locally, by managing the activities of the associated subsystem, and globally, by interacting with other agents and influencing their operations. This modularity results in a more robust management of the onboard activities, with the possibility to address faults both at the local and at the global level. This can be exploited on highly autonomous satellites and on spacecrafts with limited contacts with the ground (e.g., interplanetary probes).

As about future works, our multiagent architecture could be completed by considering other resources, like onboard memory and processing power. Also fault recovery could be improved, starting from the primitive procedures we implemented. Moreover, the potential advantages of the proposed multiagent architecture could be made more concrete by distributing agents over more processors. In general, our future research will address the use of agents for developing autonomous space systems.

Acknowledgements

The authors would like to thank Michèle Lavagna and Simone Farè for their contributions in the development and implementation of the multiagent system presented here.

References


