Interactive Autonomy for Space Applications

Amedeo Cesta
IP-CNR, Roma
cesta@ip.rm.cnr.it

Enrico Giunchiglia
DIST - Univ. di Genova
enrico@dist.unige.it

Paolo Traverso
ITC-IRST, Trento
leaf@irst.itc.it

Abstract

In this paper we describe JERRY, a system which supports the interactive design, planning, control and supervision of the operations of autonomous systems in a space environment. The requirement of Interactive-Autonomy in JERRY is achieved by a set of tightly integrated specialized sub-systems, which have been designed to perform effectively and efficiently their specific tasks, and, at the same time, to be open to the interaction among each other. This results in a system with a potential high level degree of autonomy, but which can still be controlled and guided through user interaction.

A System Supporting Interactive Autonomy

The increasing complexity of the services requested to robotic devices in space applications results in a need for more and more sophisticated and autonomous systems. A compelling requirement for space *autonomy* has led to the development of systems that perform automatically complex, time consuming and critical tasks without the need of human intervention (Muscettola *et al.* 1998).

The recent development of space autonomy for critical tasks results however in a set of novel problems, including the possibility of humans to control autonomous robotic devices (see for example (Dorais *et al.* 1998; Jerico 1998)) still guaranteeing a high level of safety of the operations that are perfomed interactively. This is due to quite a number of factors, such as the possibility of completely unexpected (and possibly dangerous) events that may require humans intervention. We call *Interactive-Autonomy* the ability of a system to provide a high level of autonomy still retaining the possibility for the user to monitor, control and override potentially autonomous operations in a safe way.

In this paper we describe JERRY, a system which supports the interactive design, planning, control and supervision of the operations of autonomous systems in a space environment. The requirement of Interactive-Autonomy in JERRY is achieved by a set of tightly integrated specialized sub-systems, which have been designed to perform safely, effectively and efficiently their specific tasks, and, at the same time, to be open to the interaction among each other. The user can directly operate each module step-by-step, and verify (at different levels of detail) the results of critical steps against safety requirements.

JERRY has been developed as part of an ongoing and more ambitious project funded by ASI, the Italian Space Agency. In this application, JERRY provides its functionality to different kinds of users which have to design, control and monitor a SPIDER Robot Arm performing quite complex tasks, e.g., the set up of several kinds of experiments in a space workcell. Even though the project is still running, a first prototype is already working and available for experimentation. In this scenario, e.g., the SPIDER arm is supposed to extract a tray from a shelf, fix it to one out of two tables and then automatically perform experiments moving objects contained in the tray.

In this paper, we first provide a global overview of JERRY by describing its high level architecture. We then describe the main features of each subsystem: the user interaction module, the planning module, and the execution module. Some conclusions end the paper.

JERRY's Architecture

The design of JERRY is based on three main components:

- **User-System Interaction Module.** It provides the user with the ability to inspect and direct every step of a system operation, via user requests of different services to specialized subsystems designed as "open systems".
- **Planning Module.** It provides a set of different planning services, including the generation of different kinds of plans of actions to achieve different kinds of high-level specifications of tasks to be performed (called goals), the validation of plans against requirements, their

step-by-step simulation.

- **Execution Module.** It provides a set of different execution services. It can compile a high level plan (provided by the planning module) into a program that is directly executable by a robotic device, execute it according to different modalities (e.g., either interactive or automatic), and monitor the execution. Further services include the step-by-step generation of actions, the verification that the executable program satisfies certain requirements and its step-by-step simulation.

The architecture of JERRY has been designed according to a main design choice: the User-System Interaction Module is central and can request different services from the other modules, in a "client-server style". The user can control the flow of JERRY 's operations and choose both the degree of automation and the level of interaction.

- **Degree of automation.** The user can decide to run the system within a wide range of options with different degrees of automation, from fully automatic to step-by-step interactive modes. The possibility is given to the user to decide to run the system in a fully automatic way: a goal is provided to the planning module that generates a plan to be executed by the execution module. failure). On the other hand, the user can decide to control interaction modality, the user may ask the Planning Module for a plan; the plan is inspected, validated and/or simulated; the first planning step is extracted from the plan and passed to the Execution Module that compiles it into an executable program; the program is inspected, possibly verified and simulated, and, finally, it is executed and monitored.
- **Level of interaction.** The user can access data and control the behavior of highly automatic systems by providing either high level specifications of what has to be achieved or detailed constraints on how the task should be performed. For instance, the user can request the planning module to generate automatically a high-level plan which achieves a high-level specified goal, or can direct the planner by imposing constraints on how to generate the plan. Analogously, the user can request the execution module to generate automatically the low level program corresponding to a plan, or can direct the execution module by imposing constraints on how the low-level robotic plan has to be generated.

The structure of the system is represented in Figure 1. In this figure, the planning and execution modules are visible in the top part, while the interaction module (with the sub-modules acting as interfaces with one of the other modules) is the

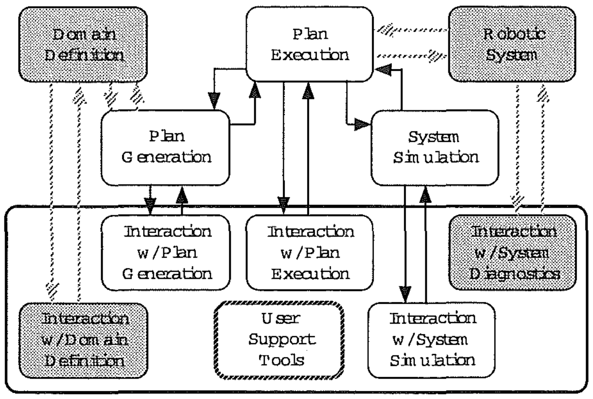
Domain Definition
Plan Execution
Robotic System
Plan Generation
System Simulation
Interaction w/Plan Generation
Interaction w/Plan Execution
Interaction w/System Diagnostics
Interaction w/Domain Definition
User Support Tools
Interaction w/System Simulation

Figure 1: Structure of the System

"big box" at the bottom. The "Domain Definition" box represents a module that allows the user to specify the domain considered, and is currently part of the simulator. The "Robotic System" box represents the real robotic device. The solid arrows represent a flow of information, while the dotted arrows represent a still missing connection. For example, the dashed arrow between "Domain Definition" box and the interface, means that currently the user can specify a domain not through the interface, but only interacting directly with this module.

JERRY can work at two levels of interactions that are targeted to two typical users of space robotic devices: the "*programmer-level*" contains functionalities offered to the robotic system operator; the "*user-level*" deals with activities performed by on-ground scientists or payload operators. At the programmer-level, the user can program the behavior of the device using its typically low-level interface language, e.g. the language (called PDL2) currently used to control the SPIDER arm. A typical PDL2 instruction is "MOVE LINEAR TO `point-in-space`", where `point-in-space` is a triple of real values. This level of interaction is adequate for an experienced user. Nevertheless, programming complex tasks at this level may be very difficult for a user which has no experience with the programming language, e.g. PDL2. Moreover, low-level programs can be hard to maintain and re-use. For this reason, interaction at the user-level provides also non experts (e.g. scientists) with the ability to specify robotic tasks. Such users do not need any knowledge of the underlying physical structure of the robotic device (e.g. of the degrees of freedom of the arm) or of the physical scenario (e.g. of the exact position in space of the objects). A typical high-level instruction is "GET OBJECT `object-name`".

Operationally, the two interaction levels reflect two working modalities:

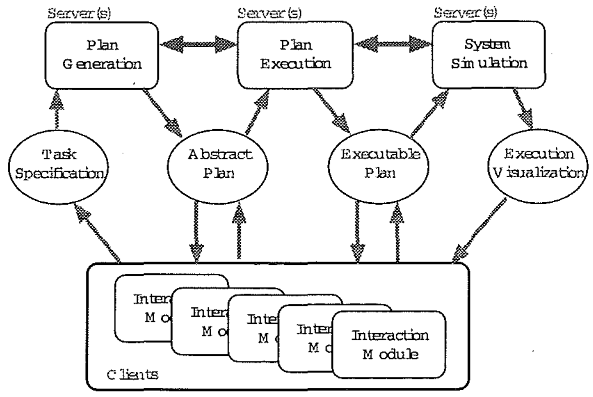

Figure 2: JERRY's Current Architecture

user-drives-system-supervises: in this modality an expert, knowledgeable of the underlying robotic device and mission interacts with the system by describing the mission in the robotic device interface language. The mission is encoded as a low level plan which is directly executable by the execution module.

system-drives-user-supervises: in this modality the user (even a non expert, e.g. a scientist) fixes the goal in a high level specification language. The high level specification cannot be executed directly. The system can generate automatically executable low level programs. This is achieved in two steps. First, the planning module generates a set of high level actions which have to be executed in different situations and which are guaranteed to achieve the goal. Then the execution module, for each high level action, generates a corresponding sequence of low-level actions in the robotic device interface language (e.g. PDL2). Independently from how the low-level plan is generated, the execution module is responsible for its execution, and for the monitoring of the behavior of the robotic system. At each step of the execution process, the user can be prompted for validating the high-level action to be executed, or, if required, the current low-level program.

The resulting architecture is highly modular and configurable: the system can be configured to work at different levels of automation (e.g. depending on the activity performed by the planning module) and the user has the possibility to flexibly access data manipulated at different levels of detail (e.g. data at the execution or at the planning level). The interface can be set to be used by users with different experience (programmers or scientists) and can also be adapted to different input devices (e.g., driven entirely from mouse or touchpad, entirely from keyboard, or, possibly, from custom input devices).

A first version of the demonstrator has been fully implemented, is available for inspection, and is currently under development to improve its general performance and to enrich the services offered to the user. This demonstrator (whose architecture is represented in Figure 2) is based on a client/server architecture in which a client interface service is able to continuously interact with the planning, execution and simulator modules. This has involved the development of specialized protocols that allow each interaction module to safely exchange data with the three servers through point-to-point communication. Current protocols are deliberatively designed to be very simple to minimize the overhead of communication between modules and to quickly arrive to a first integration.

Interaction Module

The role of software systems like JERRY is to allow different users to employ complex robotic devices while preserving the levels of responsibility that users have in their working contexts. Both the user-level and the programmer-level preserve the usual working activity, but offer a number of additional functionalities that allow the users to focus on strategic and decisional tasks and to delegate repetitive or very difficult tasks to the interactive planning software.

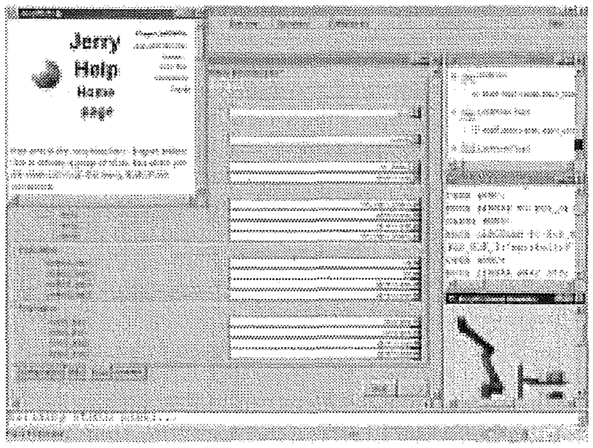

Figure 3: JERRY Interactive Module

The JERRY Interaction Module consists of a Graphical User Interface endowed with the following functionalities:

- Task oriented help.
- Problem specification targeted to the planner domain representation language.
- Inspection of high-level plans: a rather simple representation of the plan returned by the planner is shown and the possibly of inspecting the representation of single plan states is given.

- Inspection of plan compilation: the low-level code produced by the plan compilation and execution module is shown to the user.
- Robotic device simulator visualization.

The current look of the Interactive Module is shown in Figure 3. In the Figure we can see (*i*) the Help window (top-left) that is designed as a separate entity; (*ii*) the planning problem specification window (main window below the Help window); (*iii*) the plan current in execution (top-right); (*iv*) the PDL2 code corresponding to the action being executed (middle-right); and (*v*) the execution of the plan coming from the simulator (bottom-right). The size of the 4 windows corresponding to point from (*ii*) to (*v*) are interconnected and vary according to the user current focus of attention that is always contained in the main window.

According to the subdivision made between the "programmer-level" and the "user-level", the tasks allowed to each level have been defined. In the current implementation of the "user-level interaction" the users can: (*i*) get acquainted with an operating environment; (*ii*) define specific parameters of the scenario (e.g., decide the number of trays in an experiment); (*iii*) specify the goal he want to achieve and the constraints to satisfy in achieving it; (*iv*) ask the planning module to determine the set of actions (the plan) that achieves the goal; (*v*) display and comment the resulting plan; (*vi*) activate plan execution. Special attention has been dedicated to automatically checking the consistency of commands selected by the user and in offering explanation facilities for non-expert users. The "programmer-level interaction" offers: (*i*) the possibility of creating robot programs directly using the robot language, (*ii*) the choice of having the planning and execution mechanisms that work as background help of the programmer; (*iii*) the possibility of experimenting different operational situations offering a choice among alternative input modalities. The possibility of customizing the interaction modality is relevant for experimenting on-flight use of the programming ability. In is worth observing that being the Interaction Module configured as a client it is possible to serve multiple users at the same time each of them interacting with personalized functionalities.

This module is implemented in Java (compatible with JDK 1.2) and represent right now a quite effective platform for studying multiple interactions styles that refer to different ways of sharing task responsibility among users and the system.

Planning Module

The Planning Module provides the user with the ability of requesting services by means of high level specifications entered through the Interaction Module. It has been designed to be highly independent of the programming language that is directly executable by robotic devices (e.g. PDL2). The Planning Module works on data structures that encode a high level description of the possible situations in a given domain (called states), of the operations that a robotic device can execute (called actions), and of the requirements for tasks to be performed (called goals).

A state of the domain is represented symbolically, e.g. by expressions like `Y is on experiment tray Z`. Operations of the robot are represented as high-level actions, e.g. `Get object Y, put Y on experiment tray Z`. Plans are possibly conditional and iterative programs that specify the actions to be executed. An example of plan is the following.

```
Plan experiment-1 is:
Get object Y;
if this action succeeds,
then put Y on experiment tray Z,
otherwise get object Y1;
...
```

At this level of abstraction, the Planning Module provides three main functions: Validation, Simulation and Plan Generation.

The *validation function* allows the user to check properties of the scenario or of putative plans. For instance, the user can ask whether `Y is on experiment tray Z` in the current scenario, or whether `Y is on experiment tray Z` after executing plan `experiment-1`. This functions provides the user with a higher confidence that a task can be performed correctly in a given way. It can also be used to inspect and debug possible plans.

The *simulation function* allows the user to simulate the execution of a given plan. It shows the evolution of the states of the domain. For instance, the user can request a simulation of the plan `experiment-1`. The simulation is performed at the level of abstraction of the Planning Module, i.e., by showing high level actions and how they affect the state of the domain. Again this functionality can be used to gain a better understanding of how a plan can perform a given task and to debug plans.

The *plan generation function* is the core of the Planning Module. The user, through the Interaction Module, provides a goal to be achieved, i.e., a high level specification of the task to be performed. The goal is a high level description of what has to be achieved. It does not detail how the task should be performed. For instance, a simple goal can be `one of Y,Y1 on experiment tray Z`. The Planning Module generates automatically plans of actions (e.g. the plan `experiment-1`) whose aim is to achieve the task specified by the goal. The plan of actions is the output which can be passed, through the Interaction Module and possibly under control of

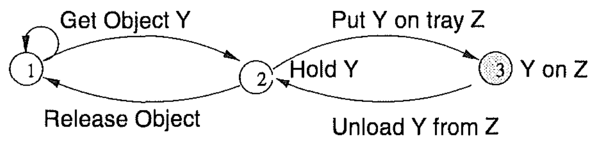

Figure 4: An example of FSM

the user, to the Execution Module.

This set of services has been designed to meet the requirement of *Interactive-Autonomy*. Indeed, a main characteristic of the planning module is that it is an open system, i.e. each of its operations (e.g. plan search) can be inspected, controlled and guided by the user. This fact opens up the possibility to provide a planning functionality which supports a "user-centered operation mode" for JERRY, in which the planner interacts flexibly with the user interface module. The user, beyond asking for a goal to be satisfied, can ask the planner for different services, e.g. show all the plans which satisfy a goal, select one of them, query the planner about the possible effects of the execution of plans, re-use existing plans, ask the planner to validate a user defined plan, inhibit some plans, query the planner about the current state of the execution in terms of high-level actions. This "user-centered" modality requires a design of the planning module which is different in philosophy wrt current state of the art planners. The planner is no longer the automatic generator of solutions, it becomes a system which exploits its automatic generation capabilities to support the user to find the right solution and is flexible enough to adjust its plan generation activity to different user requirements.

It is worth remarking that the functionalities of JERRY allow to implement an interesting set of mixed-initiative problem solving strategies that are somehow disjunct from other approaches devoted to inserting the user in the loop (Smith & Lassila 1994; Tate 1997). In our current framework the user is left free of choosing to either solve the problem himself or request the intervention of specific JERRY functionalities.

The Planning Module is developed on top of the MBP system (Model Based Planner) (Cimatti *et al.* 1997; Cimatti, Roveri, & Traverso 1998). MBP is an extension of the NuSMV Model Checker (Cimatti *et al.* 1999). It implements the "Planning as Model Checking" paradigm. The implementation relies heavily on existing work in the context of finite-state program verification and in particular on the work described in (Clarke, Grumberg, & Long 1994; Burch *et al.* 1992; McMillan 1993). The underlying idea is that domains are modeled as Finite State Machines (FMSs) and plans are generated by searching the states of FSMs. For instance, a FSM for a simple domain is depicted in Figure 4. We have three possible states (labeled as 1,2,3). Each state is labeled with the facts that hold in that state, e.g. `Hold Y` states that in state 2 the robot has at hand the object `Y`, `Y on Z` states that in state 3 the object `Y` is on the experiment tray `Z`. The actions `Get Object Y`, `Put Y on tray Z`, `Release Object`, `Unload Y from Z` are represented as transitions (arcs) between states.

Plan Compilation/Execution

The Plan Compilation/Execution module is responsible for transforming a high-level, user-oriented abstract plan into a sequence of low-level, machine-oriented execution plan. In more detail, the Plan Compilation/Execution module receives in input from the interface an arbitrarily long sequence of actions to be performed, and generates a sequence of actions (a "program") that the robot can directly execute. For example, in the case of a robotic arm, the program corresponding to a $move(o, l)$ ("move object o to location l") looks like the following sequence of instructions

```
move_near <pos_o> by <distance>;
open_hand;
move_linear <pos_o>;
close_hand;
move_near <pos_l> by <distance>;
move_linear <pos_l>;
open_hand;
```

where `<pos_o>` and `<pos_l>` are six tuples of real numbers specifying the positions of the object and of the location respectively, while `<distance>` is a real number specifying how far (along the vector specifying how to approach the position) the arm should be from the final position. Notice, the fundamental distinction between a "user-level" and a "programmer-level" plan: while the former is a sequence of symbolic actions, the latter necessarily involves some reasoning about the geometry of the scenario. As for the planning module, the execution module provides the possibility to *validate*, *simulate* and/or *generate* a PDL2 program.

Given that the positioning module has to deal with variables with infinite domains (typically intervals over the reals), the set of possible low-level plans cannot be directly encoded into FSMs. Because of this, we adopted a different mechanism in which the "validation function" rely on a user-defined LALR(1) grammar (see (Aho, Sethi, & Ullman 1986)) defining the whole set of *admissible* PDL2 plans. This grammar, though dependent on the particular scenario under consideration, can be inspected and/or modified by the user before each session. The grammar is then directly compiled into an executable code corresponding to a program accepting a plan only if it is admissible.

A "generation function" allows to associate to any high-level sequence of actions a corresponding sequence of low-level sequence of PDL2 instructions. In any case, the sequence of actions given to the execution module does not need to correspond to a complete plan. Instead, the user can (*i*) break a plan as given by the planning module into blocks of planning actions, (*ii*) require the compilation of all or some of the blocks, or (*iii*) ask for an execution program differing from the proposed one.

Finally the "simulation function" allows for the validation of either the generated or user-provided PDL2 program by a suitable call to the simulator. As obvious result, the movement of the SPIDER is displayed on a screen for validation by the user.

As for the planning module, the execution module is an open system in which the parameters affecting its behavior (e.g. the availability of a given low-level action) can be inspected, controlled and eventually modified by the user. For example, the user can inhibit the execution module from using a certain low-level action because it involves some dangerous or unavailable move for some joint. This will affect both the validation and the generation routines: no plan with the undesired instruction will be accepted and generated. As above, this fact opens up the possibility to provide a "user-centered operation mode" for JERRY, in which the execution module interacts flexibly with the user interface module.

A Java (compatible with JDK 1.2) implementation of the execution module has been realized, and is currently tested for improvements. The compilation from the grammar to the executable code has been realized via CUP (see (Hudson *et al.* 1998)).

Conclusions

This paper describes JERRY, a system for the automatic generation and execution of plans for robotic devices, and briefly reports about the case study of the SPIDER arm. The main feature of the system is the high-level of interaction that the user can decide to have with the system. This level of interaction is critical in the context of spatial missions, where (*i*) unforeseen emergencies can happen, and (*ii*) still the mission has to proceed, possibly under the humans' supervision.

JERRY has been designed to be a flexible, open architecture. Care has been taken in order to distinguish the domain-dependent from the domain-independent tasks in order to minimize the customization efforts. JERRY's architecture and underlying ideas have been tested and made operational for monitoring and controlling a SPIDER robotic arm operating in an indoor environment very close to the payload tutor experiment described in (Di Pippo *et al.* 1998).

Acknowledgments

We would like to thank Marco Daniele, Maria Grazia Maccione and Paola Riccucci for their work in JERRY. Thanks to Marco Schaerf for making the robot simulator available. This research is supported by ASI (Italian Space Agency) as part of the project "Un Sistema Intelligente per la Supervisione di Robot Autonomi nello Spazio". Special thanks to the other participants in the project for the many fruitful discussions and the technical help.

References

Aho, A. V.; Sethi, R.; and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Reading, MA, USA: Addison-Wesley.

Burch, J. R.; Clarke, E. M.; McMillan, K. L.; Dill, D. L.; and Hwang, L. J. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation* 98(2):142–170.

Cimatti, A.; Giunchiglia, E.; Giunchiglia, F.; and Traverso, P. 1997. Planning via Model Checking: A Decision Procedure for AR. In *Lecture Notes in Computer Science*, volume 1348.

Cimatti, A.; Clarke, E.; Giunchiglia, F.; and Roveri, M. 1999. NuSMV: A new symbolic model verifier. *Lecture Notes in Computer Science* 1633.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proceeding of the Fifteenth National Conference on Artificial Intelligence (AAAI-98).* Madison, Wisconsin: AAAI-Press. Also IRST-Technical Report 9801-10, Trento, Italy.

Clarke, E.; Grumberg, O.; and Long, D. 1994. Model Checking. In *Proceedings of the International Summer School on Deductive Program Design.*

Di Pippo, S.; Colombina, G.; Boumans, R.; and Putz, P. 1998. Future Potential Applications of Robotics for the International Space Station, Robotics and Autonomous Systems. *Robotics and Autonomous Systems* 23:37–43.

Dorais, G.; Bonasso, P.; Kortenkamp, D.; Pell, B.; and Schreckenghost, D. 1998. Adjustable Autonomy for Human-Centered Autonomous Systems on Mars. In *Proc. Mars Society Conference.*

Hudson, S. E.; Flannery, F.; Ananian, C. S.; Wang, D.; and Appel, A. W. 1998. CUP parser generator for Java.

1998. JERICO Project Description. ASI (Italian Space Agency) Internal Documentation.

McMillan, K. 1993. *Symbolic Model Checking.* Kluwer Academic Publ.

Muscettola, N.; Nayak, P.; Pell, B.; and Williams, B. 1998. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence* 103:5–48.

Smith, S. F., and Lassila, O. 1994. Toward the Development of Mixed-Initiative Scheduling Systems. In Burstein, M. H., ed., *Proceedings ARPA-Rome Laboratory Planning Initiative Workshop, Tucson (AZ)*, 145–154. Morgan Kaufmann.

Tate, A. 1997. Mixed-Initiative Interaction in O-Plan. In *Papers from 1997 AAAI Spring Symposium on Computational Models of Mixed-Initiative Interaction, Stanford (CA)*, 163–168.