Model-based Execution through Reactive Planning for Autonomous Planetary Rovers

Alberto Finzi[**]
DIS, Università of Roma
La Sapienza
Via Salaria, 113, 00198 Roma
Italy
finzi@dis.uniroma1.it

Félix Ingrand
LAAS/CNRS
7 Avenue du Colonel Roche,
F31077 Toulouse Cedex 04
France
felix@laas.fr

Nicola Muscettola
NASA Ames
MS 269-2
Moffett Field, CA 94035
USA
mus@email.arc.nasa.gov

Abstract. This paper reports on the design and implementation of a real-time executive for a mobile rover that uses a model-based, declarative approach. The control system is based on the Intelligent Distributed Execution Architecture (IDEA), an approach to planning and execution that provides a unified representational and computational framework for an autonomous agent. The basic hypothesis of IDEA is that a large control system can be structured as a collection of interacting agents, each with the same fundamental structure. We show that planning and real-time response are compatible if the executive minimizes the size of the planning problem. We detail the implementation of this approach on an exploration rover (Gromit, an RWI ATRV Junior at NASA Ames) presenting different IDEA controllers of the same domain and comparing them with more classical approaches. We demonstrate that the approach is scalable to complex coordination of functional modules needed for autonomous navigation and exploration.

1 Introduction

As robotics research advances, planetary robotics is tackling increasingly challenging mission scenarios. Rovers have demonstrated autonomous traverse of several kilometers in Mars-analogue terrains (Wettergreen *et al.* 2002) and several field experiments are showing increasing effectiveness in autonomously placing scientific instruments on observation targets (Pedersen *et al.* 2003; Lacroix *et al.* 2003). The increased level of autonomy opens the possibility of much more productive planetary science missions than present ones (e.g., each of the Mars Exploration Rovers is expected to perform close investigation of 6 to 12 targets in 90 days). It also promises a reduction of workload and stress for the ground crew, two factors that make it impossible to use the traditional highly manual commanding process for the more complex future rovers.

The increased mission and rover complexity requires more capable on-board software. Not only individual modules must be more robust and capable, but there must be a substantial increase in the ability to coordinate these modules. This is a significant problem since complexity increases exponentially with the number of possible interaction among complex modules. In complex operational scenarios the interactions that need to be considered also increases because of the number of concurrent anomalies that must be handled robustly.

Several current approaches to autonomy tackle the coordination problem by separating the control software into multiple layers of increasing levels of abstraction and coordination complexity (Muscettola *et al.* 1998b; Bonasso *et al.* 1997). For example, in a three-layered architecture the low level constitutes a *functional layer*, including control modules such as platform mobility drivers and more complex functionalities such as obstacle avoidance and stereo-map construction. The middle layer is an *executive* that can run a library of procedures that monitor and activate lower level functional modules to achieve different types of mission goals (e.g., "go to location X" or "take a image mosaic of rock Y"). Finally, at the highest level a *planner* takes several mission goals and schedules them for execution over an extended period of time, determining which execution procedures need to be invoked to achieve the selected goals and which resources can be allocated for their achievement at what time. Several current approaches to rover autonomy essentially follow the previously described structure (Volpe *et al.* 2001; Chouinard *et al.* 2003).

The multi-layered approach has had some significant successes (e.g., the implementation of a highly autonomous spacecraft controller on DS1 (Muscettola *et al.* 1998b)) but integration and testing is difficult because of the technological diversity of the different layers. Focusing on the relation between the planner and the executive, while the planner typically uses a declarative cause-effect model of the all possible behavior of the system and of the external environment, the executive has only a compiled view of such models into its procedure library. Such procedures are optimized to achieve the few behaviors that they encode. Exceptional conditions outside the covered behaviors must be caught by more drastic fault protection measures (e.g., putting the rover in standby and waiting for external help from ground operators). The manual encoding of control knowledge in

[**]List of authors in alphabetical order.

the procedures has also the undesirable effect of making the executive's logic much more opaque than that of the planner. This makes more difficult the testing, verification and validation with formal methods such as model checking. Building autonomy software that is easier to validate is essential for its adoption as the on-board controller of a planetary mission.

This paper describes the design and implementation of a real-time executive for Gromit, a mobile robot with capabilities equivalent to state-of-the-art field exploration rovers. The executive is significantly different than traditional procedural executives since it uses a temporal reactive planning as its only run-time reasoning engine. The executive has the same capabilities of a procedural executive but uses a fully declarative domain representation rather than a procedural one. Our executive conforms to the Intelligent Distributed Execution Architecture (IDEA) for the development of multi-agent systems (Muscettola *et al.* 2002). An IDEA control agent has a model based on temporal planning operators that describes its internal functioning and all of its communications with other agents or with the controlled plant. The model is interpreted at run time by a planner and the next planned task is then executed. Our model of plan execution is an extension of the one used in Remote Agent to execute high-level plans (Muscettola *et al.* 1998a). Reliance on a planner for on-line decision making has been traditionally excluded from consideration due to the apparent incompatibility of real-time responses with possibly exponential computation. We show that temporal planning and real-time response are not incompatible if the executive minimizes the size of the planning problem solved at each execution cycle. Previous work (Lemai *et al.* 2003) demonstrated the feasibility of the approach for a simple rover. In this paper we demonstrates that the approach is scalable to the more complex coordination of functional modules necessary for the control of state-of-the-art rovers.

The paper is organized as follow: we first present the rover experimental platform. We then present a traditional procedural controller that has been used to control the rover. Then we introduce the IDEA control architecture (Muscettola *et al.* 2002; Lemai *et al.* 2003) that provides the controller template on which we implemented the planner-based controller. Subsequent sections present the planner-based executive, focusing on the planning model and on issues related to the minimization of the size of the planning problem. We conclude with a comparison of our approach and the classical procedural executive approach, and a more general discussion on the state of the art on planning and execution control.

2 The Gromit Domain

We shall illustrate our presentation with a simplified version of a real world experiment, running on Gromit, an RWI ATRV Jr at NASA Ames. The mission of the robot is to visit a number of waypoints, into an initially unknown rough environment, while monitoring interesting targets on its path. The robot uses stereo vision to continuously build a model of the environment. Thus, considering the current position, the targeted waypoints and the environment model, a 3D motion planner continuously produces an arc trajectory which avoids obstacles, maximizes stability and tries to reach each waypoint in turn. At the same time, a monitoring task senses the surrounding environment and upon detecting an interesting target, stops the robot and takes a picture of it, tagged with its position for possible future study if the target is considered worth reexamining by scientists.

The functional layer of Gromit has been implemented using the functional modules of GenoM that is part of the LAAS Architecture (Alami *et al.* 1998). Modules are programs providing services to a client upon request and producing data to fulfill the service. Each data production is called a poster. For each module used in Gromit (Figure 1) we briefly describe the functional capabilities of the module, the request and the poster. For more information on the implementation and algorithms used by each module see (Lacroix *et al.* 2003).

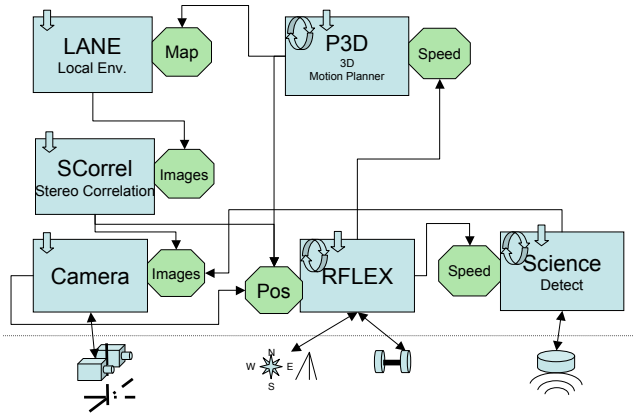

Figure 1: The functional modules of the example (arrows represent the "use" of the pointed poster by the pointing module).

- **RFLEX** is interfaced with the low-level speed controller of the wheels, to which it passes the speed available from a speed reference found in a poster (in our example, the poster is produced by either P3D or Science) and provides speed control ($rflex\_speed\_track\ speed\_ref\_poster$) on the wheels of the robot on a speed reference found in a poster (in our example produced by P3D or Science). It also produces a poster containing the position of the robot based on its odometry $robot\_pos$. Both posters are produced/used at 25 Hz. To stop the robot one can set up a poster with a null speed and instruct RFLEX to use it.
- **Camera** takes a pair of stereo calibrated images upon the $camera\_shot$ request and save them in the $camera\_images$ poster. This takes between 1 and 7 tenth

of a second. These images are tagged with the current position of the robot available in the $robot\_pos$ poster.

- **SCorrel** takes the stereo pair in the $camera\_images$ poster, produces a stereo correlated image and stores it in $scorrel\_image$ upon receiving the $scorrel\_scorrel$ request. $scorrel\_image$ is tagged with the $robot\_pos$ poster value. SCorrel takes a few seconds (2-3) to complete.

- **Lane** builds a model of the environment by aggregating subsequent cloud of 3D points produced by SCorrel. It can service two requests $lane\_read$ to read the $scorrel\_image$ in an internal buffer and $lane\_fuse$ to fuse the read $scorrel\_image$ in its map which is available in the poster $lane\_map$. It takes a second or so to complete.

- **P3D** is a rover navigation software using a method very close to the GESTALT rover navigation software operating on the Mars Exploration Rovers (Goldberg *et al.* 2002). It produces an arc trajectory in the $P3D\_speed$ poster, to try and reach a waypoint, still avoiding "obstacles" by making a stability analysis in the environment available in the poster $lane\_map$. As long as it has not reached a particular waypoint, this module runs continuously and periodically (0.5 Hz) reevaluates the position and the environment to produce a new arc, which is translated in a speed reference poster: $p3d\_speed$. This is the speed reference used by RFLEX to get the robot moving.

- **Science** This last module monitors a particular condition of interest to scientist (such as a detecting rocks with a particular features) using a particular instrument. In our case, when such condition arises while the robot is moving toward a waypoint, it stops (by instructing RFLEX to use a $Science\_speed$ which is null) and takes a picture of the rock.

In order for all of these module to correctly operate as an integrated system, we need to specify how to coordinate their concurrent execution. In particular we need to specify which sequences of poster productions/consumptions performed by which modules yield a correct overall rover behavior. The high-level description of rover operations is the following. The robot continuously takes pictures of the terrain in front of it, performs a stereo correlation to extract cloud of 3D points, merges these points in its model of environment and starts this process again. In parallel, it continuously considers its current position, the next waypoint to visit, the obstacles in the model of the environment built and produces a piece of trajectory, which result in a speed reference. These two interdependent cyclic processes need to be synchronized. Last, a third process interrupts regular way point visiting whenever an interesting rock has been detected.

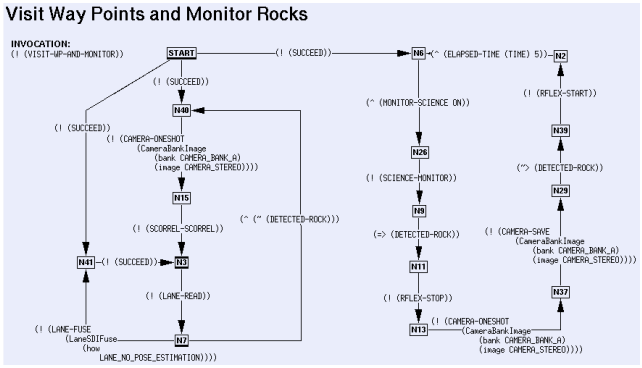

Figure 2: Example of a PRS procedure.

3 A Procedural Controller

The aforementioned scenario was originally implemented using a procedural executive (PRS : OpenPRS (Ingrand *et al.* 1996)). Figure 2 shows the top level procedure used then to properly sequence the calls to the functional modules to perform this autonomous navigation.

PRS is composed of a set of tools and methods to represent and execute plans and procedures. Procedural reasoning differs from other commonly used knowledge representations as it preserves the control information (*i.e.* the sequence of actions and tests) embedded in procedures or plans, while keeping some declarative aspects. OpenPRS is composed of:

- **A database** which contains facts representing the system view of the world and which is constantly and automatically updated as new events appear. In our robot architecture, the database contains symbolic but also numerical information such as the position of the robot, the status of the various requests, *etc*.
- **A library of procedures/plans,** each describing a particular sequence of actions and tests that may be performed to achieve given goals or to react to certain situations. The content of this procedure library is application dependent and it may include predefined plans to perform robot tasks.
- **A task graph** which is a dynamic set of tasks currently executing. Tasks are dynamic structures which execute the "intended plans", they keep track of the state of execution of the intended procedure and of the state of their posted subgoals.

In PRS, each plan/procedure is self-contained: it describes in which conditions it is applicable and the goals it achieves. This is particularly well adapted to context based task refinement and to incremental robot tasks. In our example, see Figure 2, the top level procedure describes the proper sequencing to perform the navigation and the science experiment. One can see on the left part the two synchronized loops (camera/scorrel and lane read/fuse), while on

the right part, we have the loop which monitor "rocks" and take picture of them.

Overall, the procedural approach does identify the control cycles but fails to tie them to the actual physics of the controlled devices and thus to their states. This has negative effects on the ability to handle failures and analyze the validity of the control loops under all possible execution circumstances.

4 IDEA Architecture

IDEA (Muscettola *et al.* 2002) is a model-based autonomy architecture that supports the development of large, multi-agent control systems. Unlike three-layered architectures, each IDEA agent strictly adheres to a single formal virtual machine and uses a model-based reactive planner as its core engine for reasoning.

The IDEA Virtual Machine Figure 3 gives an overview of the components of an IDEA agent. The agent communicates with other agents (either controlling or controlled by the agent) using an *Agent Relay*. The function of the relay is to maintain the context of execution in term of procedures and their return values. The procedures are considered as executed on separate parallel threads. Each procedure invocation is handled by the relay analogously to CORBA's Asynchronous Method Invocation (Schmidt and Vinoski 1999) with the possibility of each procedure to be aborted by the invoking agent. The agent relay maintains the execution context by sending or receiving message invocations (respectively corresponding to goals sent to controlled agents or received from controlling agents) and receiving or sending method return values (corresponding to the achievement of a goal at its status respectively by the controlled agent or by the agent itself on behalf of a controlling agent). It is assumed that each method is executed by a different thread.

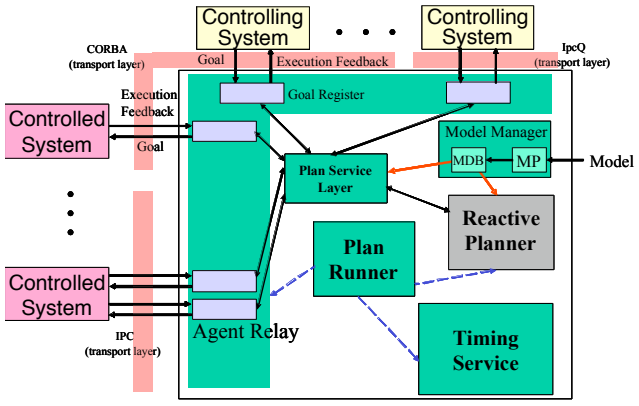

Figure 3: Structure of an IDEA Agent.

At any point in time the execution context of the Agent Relay is synchronized with the internal state of a *Reactive Planner*. The Reactive Planner has access to a domain model, maintained by the *Model Manager* that specifies the state variables (i.e., threads) that are describe the input/output state of the agent and also its internal control state. For example, a rover model will contain a thread describing the camera and another describing the state of the camera's pan/tilt unit. The model also specifies the procedures that can be executed on each different thread (e.g., a pan/tilt unit can either move to a pointing or maintain a pointing), the sequence in which they can appear on a thread (e.g., after the pan/tilt unit executes a procedure moving to a destination camera pointing, the next procedure is one that maintain the destination pointing) and the necessary synchronization between procedures occurring on parallel threads (e.g., while the camera is executing a procedure acquiring an image with a certain pointing, the pan/tilt unit must be executing a procedure maintaining that pointing). The Reactive Planner is responsible for generating new procedure invocations to be maintained in the Agent Relay (and therefore causing communication with external agents) that are consistent with the model maintained by the Model Manager.

The *Plan Service Layer* is the module that ensures complete consistent synchronization of the execution context between the agent relay and the internal state of the Reactive Planner. One of the services provided by the Plan Service Layer is keeping track of which procedure parameters or return values have been communicated with other agents and which have been only inferred by the internal reasoning of the Reactive Planner. The parameters communicated externally, either incoming or outgoing with respect to the agent, are assumed as fixed and cannot be modified by the reactive planner for any of its current or future invocations.

Finally, the *Plan Runner* executes a simple, finite state machine that implements the sense/plan/act cycle of the IDEA agent.

IDEA Execution Cycle The Plan Runner operates as follows:

1. The Plan Runner wakes up according to an *agent clock* at the first time after a message has been received from another agent or a wakeup timer maintained by *Timing Services* has gone off;
2. The state of the Agent Relay is updated with respect to the information resulting from the wakeup event (e.g., a procedure has received a return value);
3. The Reactive Planner is invoked and the planner synchronizes its internal state with the agent relay through the Plan Service layer compatibly with the planning method used by the reactive planner;
4. When the Reactive Planner terminates the Agent Relay loads the new context of execution and sends appropriate

messages to the external agents. For example, if a procedure has been terminated by the Reactive Planner, the corresponding return value (determined by the Reactive Planner) is sent to the controlling agent;

5. The Reactive Planner is invoked to determine what is the next time at which execution is expected to occur (barred any external communication). Such time for example could be minimum of the earliest times for the start or end of any current or future procedures in the plan. The time is set in the Timing Services module as the next wakeup time for the agent;
6. The Plan Runner goes to sleep and waits for an external message or the expiration of a wakeup timer.

Attributes, Tokens and Compatibilities Although IDEA does not prescribe the format of the internal organization of the Reactive Planner, its model assumes a semantic that is equivalent to that of the EUROPA planning technology. (Jeremy Frank 2003). We will assume the existence of a centralized Plan Database where the input/output and internal state of an agent is represented as set of *attributes* (also referred to as *state variables*) whose value (representing the constraint on a procedure invocation) changes over time. Values of attributes corresponds to actions and state literals in the sense of classical AI planning. A value extended over a period of time is also called a *token*. The history of states for a state variable over a period of time is called a *timeline*. For example, given a rover domain, $position$ is a possible attribute; $going(a, b)$ from time 1 to 3 and $at(b)$ from time 3 to 5 are intervals representing, respectively, an activity and a state. The IDEA modeling language is also compatible to the EUROPA planning technology and identical to the modeling language of the Remote Agent planner (Jonsson *et al.* 2000). The interval constraints among all possible values that must occur among tokens for a plan to be legal are organized in a set *compatibilities* that absolve the same function than temporally scoped operators in temporal planning. A compatibility is a conjunction of relations each defined by: i. equality constraints between parameter variables of different tokens; ii. simple temporal constraints on the start and end variables. The latter are specified in terms of metric version of temporal relations a la Allen (Allen 1981): $meets$, met_by, $contained\_by$, $contains$, $before[d, D]$, $after[d, D]$, $starts$, $ends$, etc. For instance, $going(x, y)$ $meets$ $at(y)$, and $going(x, y)$ met_by $at(x)$ specifies that each $going$ interval is followed and preceded by a state at.

Reactive and Deliberative Planning In IDEA reactive planning determines the next action on the basis of sensory input and time lapse wakeups. Reactive planning may use a standard planning engine but it is restricted to operate within a maximum time limit, the agent's *latency* (i.e. the time interval between two ticks of the agent clock). More complex problem solving (e.g., long-term task planning) typically requires more time than the latency allows. IDEA provides a rich environment for integrating any number of deliberative planners within the core execution cycle (Figure 4). Different specialized planners can cooperate in building a single plan coherently with the agent's model. Also in IDEA the activation for a deliberative planner is programmed in the model. This can be obtained by modeling the planner like any other subsystem, i.e., by specifying a timeline that can take tokens whose execution explicitly invokes the planner. This makes it possible to appropriately plan the time at which deliberate planning can occur compatibly with the internal and external state modeled by the agent. For example, it is possible to determine that deliberative planning should happen while a rover is either not moving or moving in a predictable way that allows to accurately predict the initial conditions assumed by the deliberative planner. We will discuss issues related to interaction between deliberative and reactive planning for execution further in this paper.

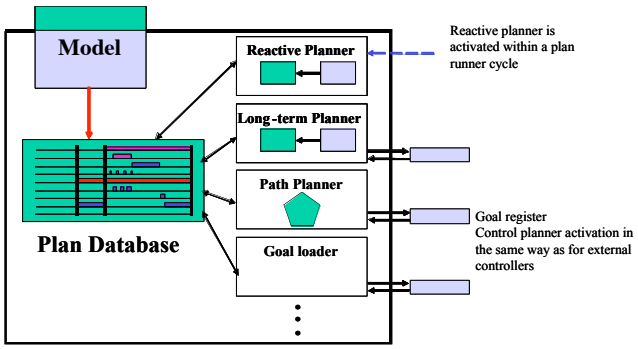

Figure 4: Reactive-Deliberative interaction.

5 A Model-based Controller for Gromit

The IDEA Gromit executive is a single IDEA agent that operates as controlling agent for the functional modules of Gromit. For each of them we shall consider the "visible" state variables of interest and the associated compatibilities (See Figure 5).

5.1 Attributes and Tokens

- RFLEX has a state variable for the position of the robot $position\_sv$, with each token representing a specific robot position, and another one for the speed control of the robot $speed\_sv$, with each token representing the reference speed passed to the wheels controller.
- Camera has one state variable ($camera\_sv$) representing the camera status (taking a picture, or idle).
- SCorrel has one state variable ($scorrel\_sv$) representing the SCorrel process (performing the stereo correlation, or idle)

- Lane has one state variable ($lane\_sv$) representing the model building process (modeling or idle)
- P3D has one state variable ($p3d\_sv$) for its state (idle or computing the speed of the robot) and one for the way_points to visit (wp_sv).
- Science has one state variable ($science\_sv$) for its status (monitoring interesting rocks or idle).

For now, one will assume that the data themselves (e.g., pictures, stereo correlated images, map) are available as a result of the associated tokens on each state variable (e.g., the position value is available on the considered token on $robot\_position\_sv$)

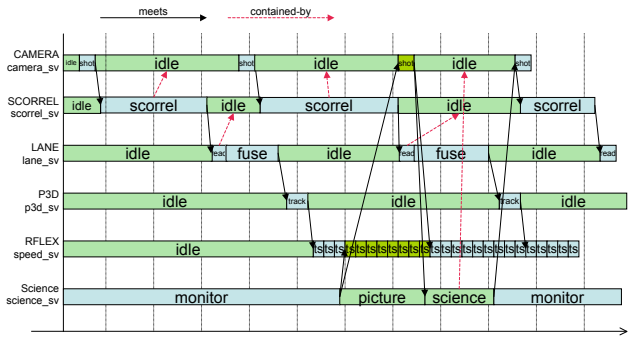

Figure 5: Partial Gromit model (timelines and compatibilities).

5.2 First Principles in Gromit

Now that we have the state variables of interest, we can consider the compatibilities which link them. If we express the aforementioned problem using a first principle approach, we will write constraints specifying things such as:

- The p3d token to reach a waypoints (on the wp_sv) has to end with a successful plan token on $p3d\_sv$. This plan token needs the waypoint token (on wp_sv), the current position on $position\_sv$ and the model of the environment on $lane\_sv$.
- $lane\_sv\_fuse$ requires a $read$ which requires a stereo correlation on $scorrel\_sv$,
- $scorrel\_sv$ requires a pair of picture taken by the camera with a $shot$ on the $camera\_sv$ timeline and requires that no other pictures should be taken until it is done with the $scorrel$.
- the $science\_sv$ is started with a $monitor$ token which will trigger whenever an interesting rock is seen. This will require stopping the robot (thus setting the $speed\_ref$ to zero) and using the $camera\_sv$ to $shot$ a picture (thus interrupting the whole navigation/mapping process). Following this $shot$, a $science$ token is performed (while the camera is $idle$).

One can see that by expressing causal and temporal relationships between these tokens/timelines we describe how the overall experiment may run. It is still up to the reactive planner to produce on each timeline the proper flexible sequence of tokens resulting in internal calls in the modules. One of the interesting part in this problem is the handling of the science activities which tightly interact with the navigation activity. Such interaction, when dealt with the classical procedural approach, is the potential source of many problems and pitfalls (deadlocks, corrupted data, etc).

6 Modeling with a Planning Horizon

The core of and IDEA control agent is the reactive planner. All IDEA agents implemented so far in this and other applications (e.g., for the Personal Satellite Assistant at NASA Ames) use the EUROPA planning technology as its base, with a simple heuristic-guided chronological backtracking engine as the search mechanism used by the planner. The key control parameter on the speed of the reactive planner is the length of the horizon over which the reactive planner is requested to build a consistent plan. As the horizon becomes shorter, the size of the reactive planning problem becomes smaller and, consequently, the size of the planning search space and the maximum latency also become smaller. In other words, the smaller the planning horizon is, the more reactive the IDEA control agent becomes. However, the horizon reduction has a complementary effect on the complexity of the domain model required to achieve correct reactive execution. During execution the agent could be required to achieve a goal (e.g., a standby state) that can only be achieved over multiple reactive planning horizons. Since the reactive planner has only visibility on subgoals and tokens that occur during one planning horizon, the planning model will have to incorporate enough contextual information to "look ahead" to decisions that may be crucial to build a correct plan in future horizons and to achieve the future goal. Therefore, the shorter the planning horizon, the more contextual information each subgoal must contain on future goals and, ultimately, the more complex the declarative model becomes. This increase in complexity makes the modeling task more difficult and reduces the effectiveness of the model-based approach in capturing the structure of the domain when compared to encoding a number of prescripted control procedures.

In this section we present three different IDEA controllers for Gromit that illustrate the tradeoff between horizon duration, planner performance and model complexity.

6.1 Minimal horizon model

To minimize the size of the reactive planner's search space, it is necessary to expand as little as possible of the plan in one execution cycle. One way to obtain this is to reduce the planning horizon to its minimum possible length, i.e., the granularity of the agent clock or one execution latency. This

granularity guarantees that during reactive planning at most one token will be expanded after a token that must be terminated during an execution cycle. This is the minimal amount of information to maintain an adequate execution context in the Agent Relay. We now describe a Gromit model for a reactive planner operating over a one-latency planning horizon that fully duplicates the PRS controller in Figure 2. Figure 5 shows the timelines, tokens and some temporal relations representing a simplified version of this model.

Given the one-latency horizon, the planner can only expand tokens to cover the next execution cycle by exploiting the model to select tokens and to decide about their consistency. Since no backward search can be employed, subgoals can be interpreted as commands and the temporal model must provide a complete description of the control information: for each execution context the set of available commands must be explicitly specified. It is easy to see that the one-latency representation can become very complex. For instances, Figure 6 depicts a possible execution context in the Gromit model: $scorrel$ ends while the first $lane\_fuse$ is still processing. In the same figure we can find another context: $scorrel$ ended during $lane\_idle$. Each of these contexts must be associated with a suitable control rule, e.g., in the previous case we have $scorrel(s)\ meets\ scorrel\_wait$ and $scorrel(s)\ meets\ lane\_read(s')$. Such conditional constraints ramification is a typical phenomenon in the one-latency model: all the possible choices a long horizon planner could explore, need to be folded back into the next agent clock tick.

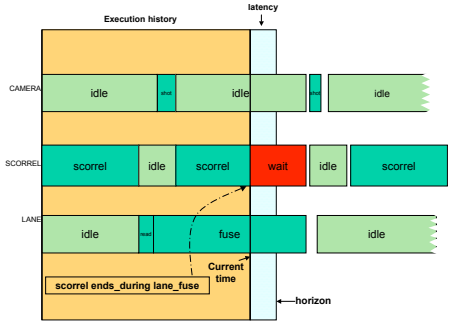

Figure 6: Scorrel-fuse interaction.

6.2 Reactive Long Horizon

The increase of model complexity due to one-latency "myopia" can be mitigated by having the reactive planner operate over a longer horizon. If the horizon is long enough a simpler model of the domain can be coded. In this context, the control information is much simpler since the context needed to achieve long term goals can be reconstructed on the fly during the planning search. This allows the model to be devoid of practically all search control information and to adhere more thoroughly to the principles of model-based declarative design.

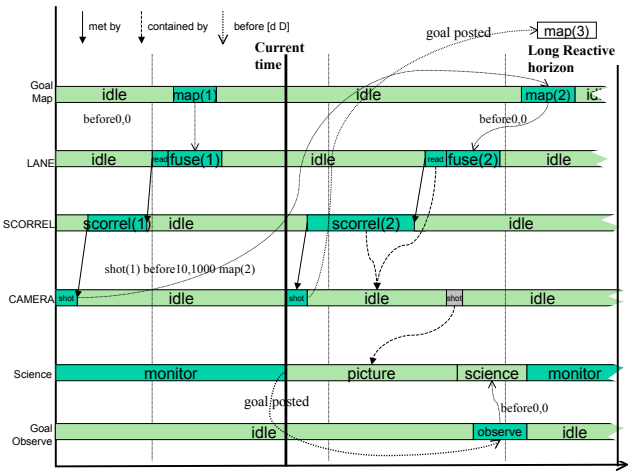

Figure 7: Gromit model: long horizon.

Figure 7 depicts the timelines involved in the mapping and observing processes for a long horizon model for Gromit. Each mapping process is started once a goal, e.g. $map(2)$ (here the activities are indexed by the cycle), is posted on the $goal\_map$ timeline and the reactive planner has to provide a plan for it within a latency. At the end of the latency, if the planner is successful, a control sequence for a mapping cycle is generated and the reactive planner can play in the role of an execution monitor checking for the consistency of the plan database. While the mapping activities are running, the next mapping cycle can be generated (e.g. Figure 7 shows $map(3)$ posted after $camera(2)$). Note here that the long horizon ensures a reactive goal-driven behavior, and, since the plan database stores a temporally flexible plan, also event reactivity is ensured. Figure 7 illustrates also the monitor token (see $science$ timeline) triggering and posting the goal $observe$ on the $goal\_observe$ timeline. Once the goal is posted, the reactive planner has to find a consistent solution where the activities involved in both mapping and science are coordinated. Note that the observing and the mapping processes can be easily integrated in the long horizon model since their coordination is managed by the reactive planner, instead, in the one-latency model, the same integration determines a multiplicative effects on the number of execution contexts.

The reactive long horizon controller is based on a simple and natural domain representation and allows for a smooth and robust behavior. The main drawback is performance because plan generation is more complex and the latency, driven by the worst case cost of plan generation, is higher.

6.3 Deliberative and Reactive Interaction

One way to address the dualism between performance and ease of representation is to exploit "dead time" to look ahead while retaining the capability to react immediately if an event signals an exceptional situation. This approach was

adopted by the Remote Agent Experiment (Muscettola *et al.* 1998b). In that case the deliberative planning horizon covered an entire day since the goal was to trade off between the achievement of several conflicting goals over a long period of time. A very similar approach can be used for short term execution, appropriately augmenting the long-horizon model described above with the information needed to trigger and execute long-horizon planning in a deliberative manner.

Besides the reactive planner, several other processes can manipulate the *plan database*. Some of these, called *deliberative planners*, are allowed to build long term plans over extended periods of time (see Figure 8). As we have previously illustrated, reactive and deliberative planners are fully integrated as they share the same domain representation and operate on the same data structures. Such uniformity allows for a tight interaction.

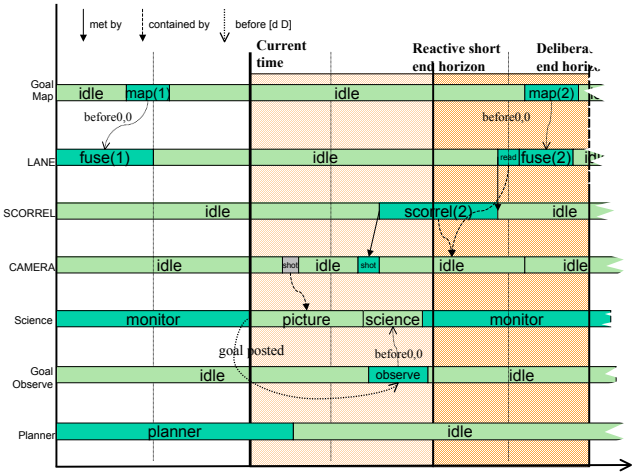

Figure 8: Long Horizon Deliberative Planning.

We experimented with a rover model exploiting the cooperation between the declarative and the reactive planner, and we tested it in simulation. Our goal was to exploit the deliberative planner to pipeline the reactive activities and thus reduce the long horizon latency. This new model is obtained as a very simple extension of the long-horizon one. The deliberative planner is associated with a long horizon while the reactive planner works with a one-latency horizon so that some control rules are visible to the deliberative planner, but not to the reactive. In this setting the two planners can concurrently work at different tasks. Figure 8 depicts again the observe-mapping cycle in the new setting. In this case mapping $map(2)$ is treated as a long term goal, while $observe$ is a short term one. In this scenario, the reactive planner can provide a final plan for $observe$ while the deliberative planner is still working at $map(2)$, and the experiment can be executed without waiting for the camera.

Reactive execution with deliberative pipelining can speed-up execution in nominal conditions, when the predictions of the deliberative planner are not invalidated by exceptional execution events. If a fault occurs, then it has to be guaranteed that the system can remain in the off-nominal state during deliberative planning without endangering the safety of the rover. This requires the identification of *immediate standby states* that can be reached in at most one step and can persist for at least the duration of a deliberative planner. This is similar to what was done in the Remote Agent Experiment where the response to a fault required the execution of a standby script and the planner was activated only while the spacecraft was in standby. In the context of reactive, real-time execution using on-line planning as its sole reasoning method, the time to come up with the standby script is reduced to a single latency horizon. To keep modeling as simple as in the long horizon case therefore we have two possible ways. The first is to identify immediate standby states and model the requirement that the deliberative planner be invoked only if the standby state has been achieved. The second, in case that an immediate standby cannot be achieved for some fault conditions, is to resort to cached standby plans. The reactive planner would then load the cached plan and start immediately executing its first step after a failure. Clearly, cached plans are analogous to procedural scripts but, in this approach, they will be only need to achieve standby. Therefore, it is expected that their number will be limited and their encoding will not require caching a large number of conditional information in the plan. The explosion of conditional information is a major limitation in achieving fully robust reactive scripts.

Our work so far has identified a framework on which to conduct quantitative studies of the tradeoff between ease of programming, encoding of search control and caching of limited standby plans. A full examination of these issues will be conducted in future work.

7 Results

The previously presented example has been implemented using a procedural executive, as well as the various IDEA models (one-latency and long horizon), and deployed on Gromit. Gromit has a dual Pentium III 1.3 Ghz CPU running Linux. As stated before, the real experiment was more complex than the one presented here and involved three others modules we did not describe in this paper ($STEO$, for stereo odometry, POM for position management to combine the classical and the stereo odometry and $Platine$ to handle the pan and tilt unit.). All the programs (i.e. functional modules, OpenPRS[1] or IDEA) are running onboard.

We started the experiment using procedural reasoning, and by using all available computational power, we were able to run the robot at roughly 10cm/s (the two most computationally intensive processing are the stereo correlation, and the stereo odometry) and to perform a “complete” cycle

[1] OpenPRS is an open source version of PRS.

(i.e. the time between two subsequent camera shot) in about one second. The monitor activity of the $science$ module would trigger the stop in a tenth of a second.

The same experiment was then programmed in IDEA using a one-latency reactive model and, despite the difficulty of correctly implementing the model, we were able to run with a latency of 0.3 seconds. The measurable reactivity of the system to new events (such as a $science\_monitor$ triggering) was on the order of one second. Using a long horizon model and a reactive planner, able to produce a plan for the next cycle and change it when $science\_monitor$ requires it, we ran at latency 1.5 seconds. The reactivity was then in the order of 3 seconds. Note that the plan produced by the IDEA model presented a better flexibility than the one produced by PRS (in particular, in the complete example, we were getting a better sequence leading to a camera_shot while lane_read was still executing). Moreover, although a subtle race condition in the PRS procedure could lead to a situation where one could take a science picture while the navigation stereo correlation is still running, the IDEA based experiment did not have such problem.

8 Discussion: Model-based vs. Procedural Executive

We have demonstrated that flexible temporal planning can be deployed at the executive level using a temporal model of some fairly low level rover control primitives. Still, beyond pure numerical results, we need to analyze how our approach compares and scales with classical approaches, in particular procedural executives.

Let consider a number of properties and see how these two approaches compare:

- **Validation and Verification** The IDEA approach has a clear advantage on this issue. Indeed, using formal models to generate plans at run time is a guarantee that the model will always be satisfied. Compared to procedural executives which only provide a threaded control structure, this is clearly an advantage.
- **Performance** Doing a temporal model consistency checking has a high tag price, compared to a simple next step execution in a procedure. Still, performing such checking on a limited horizon provides some acceptable performance. The problem is how to guarantee that such checking can be done in a time frame compatible with the given horizon.
- **Flexibility** On this aspect, a temporal model-based approach should behave better than a procedural one. Indeed, procedure are usually hardcoding the execution paths, while the control sequence generated by the IDEA reactive (long horizon) planner is context dependent and temporally flexible, hence we have a robust behavior associated with high parallelism.
- **Error Detection/Recovery** In IDEA the declarative model implicitly defines both nominal and non-nominal scenarios, thus is more robust than a procedural representation centered on a nominal scenario. In the procedural controller, each exceptional situations must be explicitly captured in some particular decision points during the course of execution. For example, the PRS controller depicted in Figure 2 is not robust: a camera shot belonging to the science cycle could be allowed during the stereo correlation. In the IDEA context, instead, this behavior violates some explicit constraints for the nominal execution, hence the planner has to react to keep the plan database consistent. The planner activity enables for smooth recoveries reducing the need for entering a standby state.
- **Ease of programming** The relative success of procedural executives comes in part from the ease they offer at the first level to express procedures, plans, scripts. This is a very natural way to encode a process which is usually thought in the same way by engineers and programmers. Meanwhile expressing planning and execution control logic using a temporal model can be difficult, in particular because of the limited horizon effects discussed in this paper.
- **Expandability/Composability** The experimental development of robust rover execution scenarios often requires adding new capabilities or processing on an existing system. Thus the need to complete or compose a new functionality in an existing execution control system. On this aspect, the procedural executive performs poorly, as one needs to reassess the consequences of the possible interactions between the new functionality and the preexisting system. IDEA in such situation can focus on the state variables and the related compatibilities which interact between the new added functionality and the system.

9 Conclusion

We have presented a rover executive that uses IDEA, a novel architecture paradigm which proposes a model based multi-agent organization to deploy embedded autonomous systems such as mobile robots. Such approach is quite different from the execution layer of traditional three-layer architectures (Gat 1997) and even goes beyond recent architecture such as CLARAty (Volpe *et al.* 2001) which aim at bridging the gap between the traditional decisional and functional layers. Still in CLARAty, one can end up with different components for planning (CASPER) and execution control (TDL), while in IDEA, the use of the same modeling framework provides a seamless transition from planning to execution control. The RMPL (Reactive Model Based Programming) approach by (Williams *et al.* 2003), is another interesting framework suitable for Reactive Model-based control. RMPL is similar to reactive embedded languages such

as Esterel, with the added ability of directly interacting with the plant state by reading and writing hidden state variables. Here it is the responsibility of the language execution kernel to map between hidden states and the plan variables. In IDEA, instead, the model is directly integrated with the functional level (drivers and sensors). Moreover, RMPL relies on HMM for mode estimation and uses abstract scripts which are to be instantiated by a model-based executive engine (Titan). In this way the control system design is simplified, but it is not clear how the cost of diagnosis/planning underneath can be controlled by the script.

A model-based approach, such as IDEA, presents a number of advantages with respect to the ambitious goal of designing an architecture and systems supporting the deployment of autonomous systems. Compared to procedural executive, it offers a more flexible execution path and has a more robust behavior for non nominal situations. Validation and verification capabilities of such approach are superior to those intrinsic in procedural execution. Integration with a high-level temporal planner is also eased by the common modeling language.

References

[Alami *et al.* 1998] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research, Special Issue on Integrated Architectures for Robot Control and Programming*, 17(4):315–337, April 1998.

[Allen 1981] J.F. Allen. An interval-based representation of temporal knowledge. In *IJCAI*, 1981.

[Bonasso *et al.* 1997] R. Peter Bonasso, James Firby, Erann Gat, David Kortenkamp, David P. Miller, and Marc G. Slack. Experiences with an architecture for intelligent, reactive agents. *Constraints*, 9(2/3):237–256, April 1997.

[Chouinard *et al.* 2003] C. Chouinard, F. Fisher, D. Gaines, T. Estlin, and S. Schaffer. An approach to Autonomous Operations for Remote Mobile Robotic Exploration. In *IEEE 2003 Aerospace Conference*, March 2003.

[Gat 1997] E. Gat. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*. MIT/AAAI Press, 1997.

[Goldberg *et al.* 2002] S.B. Goldberg, Mark W. Maimone, and Larry Matthies. Stereo vision and rover navigation software for planetary exploration. In *Proceedings of the 2002 IEEE Aerospace Conference*, March 2002.

[Ingrand *et al.* 1996] F.F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *IEEE International Conference on Robotics and Automation*, Mineapolis, USA, 1996.

[Jeremy Frank 2003] Ari Jonsson Jeremy Frank. Constraint-based attribute and interval planning. *Constraints*, 8(4):339–364, October 2003.

[Jonsson *et al.* 2000] Ari K. Jonsson, Paul H. Morris, Nicola Muscettola, Kanna Rajan, and Benjamin D. Smith. Planning in interplanetary space: Theory and practice. In *Artificial Intelligence Planning Systems*, pages 177–186, 2000.

[Lacroix *et al.* 2003] S. Lacroix, A. Mallet, D. Bonnafous, G. Bauzil, S. Fleury, M. Herrb, and R. Chatila. Autonomous rover navigation on unknown terrains, functions and integration. *International Journal of Robotics Research*, 2003.

[Lemai *et al.* 2003] S. Lemai, B. Dias, and N. Muscettola. A real-time rover executive based on model-based reactive planning. In *Proceedings of International Conference on Advanced Robotics*, June 2003.

[Muscettola *et al.* 1998a] Nicola Muscettola, Paul Morris, Barney Pell, and Ben Smith. Issues in temporal reasoning for autonomous control systems. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pages 362–368, New York, 9–13, 1998. ACM Press.

[Muscettola *et al.* 1998b] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[Muscettola *et al.* 2002] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt. Idea: Planning at the core of autonomous reactive agents. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, October 2002.

[Pedersen *et al.* 2003] L. Pedersen, R. Sargent, M. Bualat, C. Kunz, S. Lee, and A. Wright. Single-cycle instrument deployment for mars rovers. In *Proceedings of i-SAIRAS*, May 2003.

[Schmidt and Vinoski 1999] D. C. Schmidt and S. Vinoski. Programming asynchronous method invocations with corba messaging. *C++ Report*, 11, February 1999.

[Volpe *et al.* 2001] R. Volpe, I.A.D. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The claraty architecture for robotic autonomy. In *Proceedings of the 2001 IEEE Aerospace Conference*, March 2001.

[Wettergreen *et al.* 2002] David Wettergreen, M Bernardine Dias, Benjamin Shamah, James Teza, Paul Tompkins, Christopher Urmson, Michael D Wagner, and William Red L. Whittaker. First experiment in sun-synchronous exploration. In *International Conference on Robotics and Automation*, pages 3501–3507, May 2002.

[Williams *et al.* 2003] B. Williams, M. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. Sullivan. Model-based programming of fault-aware systems. *AI Magazine*, Winter 2003.