IDEA: Planning at the Core of Autonomous Reactive Agents

Nicola Muscettola **Gregory A. Dorais** **Chuck Fry** **Richard Levinson** **Christian Plaunt**
NASA QSS NASA

NASA Ames Research Center
Moffett Field, California 94035
mus@email.arc.nasa.gov

Abstract

Several successful autonomous systems are separated into technologically diverse functional layers operating at different levels of abstraction. This diversity makes them difficult to implement and validate. In this paper, we present IDEA (Intelligent Distributed Execution Architecture) a unified planning and execution framework. In IDEA a layered system can be implemented as separate agents, one per layer, each representing its interactions with the world in a model. At all level, the model representation primitives and their semantics is the same. Moreover, each agent relies on a single model, plan database, plan runner and on a variety of planners, both reactive and deliberative. The framework allows the specification of agents that operate within a guaranteed reaction time and supports flexible specification of reactive vs. deliberative agent behavior. Within the IDEA framework we are working to fully duplicate the functionalities of the DS1 Remote Agent and extend it to domains of higher complexity than autonomous spacecraft control.

Introduction

Several successful autonomous systems are separated into technologically diverse functional layers operating at different levels of abstraction (Bonasso et al. 1997) (Currie and Tate 1991) (Wilkins et al. 1994). However, there are some significant drawbacks to this approach. Developing layered systems is complex. For example, it is unreasonable to expect that domain experts (e.g., system and mission engineers in a spacecraft domain) will directly encode their knowledge in a form usable by the different agent layers. Instead, this encoding becomes the responsibility of specialists familiar with each agent layer, which increases development cost and reduces the applicability of the autonomous software. Another problem is the frequent need to encode the same requirement in different forms in the different layers. The difficulty of tracking encoding discrepancies can decrease the reliability of the autonomous software. In this paper, we describe IDEA (Intelligent Distributed Execution Architecture) an approach to planning and execution that provides a unified representational and computational framework for an autonomous agent. IDEA provides a well-founded virtual machine that integrates planning as the reasoning module at the core of the execution engine. The virtual machine is composed by four main components whose interplay provides the basis for the agent's autonomous behavior: the domain model, the plan database, the plan runner, and the reactive planner. Deliberative planning is not a core requirement for the virtual machine but, through modeling and problem solving on the plan database, IDEA provides the means to program arbitrary combinations of reactive and deliberative problem solving. IDEA also defines a simple protocol for communication among several separate IDEA agents, i.e., agents implemented using the IDEA virtual machine. We believe that this representational and problem-solving approach can be applied at all levels of the architecture of a complex agent, such as Remote Agent (Bernard et al. 1998). We have recently taken a first significant step toward demonstrating this by re-implementing the high-level control layer of the Remote Agent. This includes closed-loop reactive planning after an unrecoverable hardware fault to put the spacecraft in standby while allowing the deliberative planner to regenerate the mission plan to adapt to the degraded spacecraft capabilities.

By defining a virtual machine IDEA aims at the agent's "assembly level". We believe that using IDEA is not incompatible with current high-level execution languages (Gat 1996) (Simmons and Apfelbaum 1998) since programs written in these languages could be compiled into IDEA's "assembler" and executed by an IDEA virtual machine. Moreover, IDEA aims at defining the required functionalities and interfaces of the modules constituting the virtual machine. As such, IDEA encourages the use of different technologies and implementations for the plan database and the reactive and deliberative planners (Kim, Williams and Abramson 2001).

In the rest of the paper we briefly describe the Remote Agent architecture as an example of the state of the art in multi-layered agents. We then describe how idea differs from current multi-layered architectures. We sketch the IDEA virtual machine and point out some of its implications, mainly with respect to the reactivity and interaction between reactive and deliberative decision-making.

Layered Agent Architectures: Remote Agent

The Remote Agent (RA) was developed at the NASA Ames Research Center and at the Jet Propulsion Laboratory. RA is an autonomous control system capable of closed-loop commanding of spacecraft and other complex systems. RA was demonstrated by running on-board the Deep Space 1 (DS1) spacecraft and controlling its operations for a total of two days in May 1999 (Bernard

et al. 1998) (Nayak et al. 1999). Unlike traditional spacecraft command sequencers, RA was designed to be goal-achieving and robust. While a command sequencer simply issues low-level commands at fixed times, a goal-achieving system receives a specified state to be maintained for a specified period of time and from this it determines the relevant commands and when to issue them. A command sequencer is brittle when confronted with command failures and cannot further proceed, but RA can modify pre-planned commands in order to overcome obstacles that would normally prevent the achievement of a goal. Operational constraints were explicitly encoded into RA models. RA used these models to avoid violating the constraints regardless of the commanded goals.

The RA architecture integrates three layers of functionality: a constraint-based planner/scheduler (PS) (Jonsson et al. 2000) a reactive executive (EXEC) (Pell et al. 1999), and a Model Identification and Recovery system (MIR) consisting of a model-based truth maintenance system with diagnosis and recovery module (Williams and Nayak 1996) (Figure 1). Each layer uses a different modeling language and a different way to specify problem-solving control.

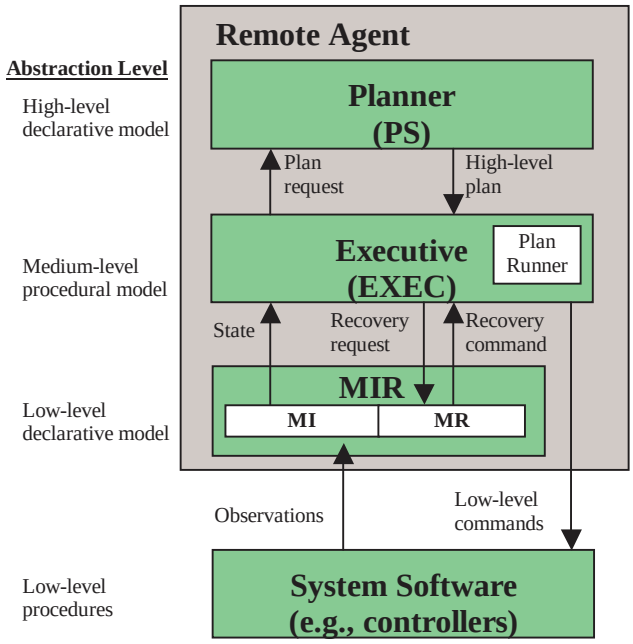

Figure 1: Remote Agent's layered architecture

At the highest level is PS, which uses a high-level declarative modeling language (HSTS DDL) to define the state machines and the temporal constraints needed to create valid plans. PS uses depth-first, backtrack search as the basic problem-solving engine. In order to produce plans in reasonable times, developers can use a simple language to specify choice selection heuristics. For the DS1 RA, we were able to write heuristics that drastically reduced backtracking, limiting it to shallow trees. This allowed PS' response time to stay between ½ hour and 4 hours depending on the size of the planning problem. This was achieved on a 20 MHz CPU for ½ of the available CPU time and within 32 MB of available memory.

The Executive (EXEC) occupies the second layer. EXEC's function is to translate high-level actions in the plan into a stream of timed, low-level commands to System Software. EXEC does so with two separate mechanisms. First it interprets the plan one step at a time with a specialized module called the plan runner. For each action currently in execution, the plan runner checks whether all logical and temporal termination conditions for the action are satisfied. If so, it terminates the action, it propagates the action termination time to the rest of the plan, and it starts the next action in the plan. When executing an action, EXEC runs a procedure associated with it in EXEC's model. Each procedure is written in ESL, an extension of LISP (Gat 1996). It specifies how to achieve the success states associated with the action using low-level commands to System Software. An ESL procedure operates at a level of abstraction higher than that of low-level commands in order to enhance reactivity. On DS1, an EXEC procedure needed to respond to any handled event within a worst-case 4-second bound.

EXEC relies on MIR to support low-level sensor interpretation and commanding. MIR provides two main functions. MI (Mode Identification) estimates state and notifies EXEC when a state changes. MI uses a detailed model of the system components (e.g., switches). Typically MI needs to consider interactions between several subsystems (e.g., sensors) in order to determine the state of some device (e.g., whether a thruster is ON or OFF). MR (Mode Recovery) uses the same model of MI and determines the least costly path from the MI estimated (faulty) system state and the one EXEC requires in order to satisfy the plan. MR also guarantees that the recovery actions do not pass through invalid states communicated by EXEC. For DS1, the maximum response time for MR was 5 seconds while MI could generally generate a diagnosis within a few hundred milliseconds

Using different problem solving modules with different representation languages and different reasoning engines had a direct advantage. In large part the modules constituting RA were based on technology already available. For DS1, it was therefore possible to concentrate on the still very hard problem of weaving these modules into a single, coherent agent. Also, one may argue that the representation and problem solving capability of each module could be tuned to maximize performance at that level. However, this heterogeneous approach made it difficult to validate all the models and procedures and to insure that they did not conflict.

The structure of IDEA

After an in depth analysis of RA's functionality, we believe that it is possible to duplicate it within a new, unified agent framework, where all layers have the same structure. In this section we give an outline of the main components in IDEA.

Tokens and Procedures

In IDEA, the fundamental unit of execution is a *token*, a time interval during which the agent executes a *procedure*. A procedure has the following general form:

$$P(i_1, \ldots, i_n \rightarrow m_{1, \ldots,} m_k \rightarrow o_1, \ldots, o_m;\ s)$$

Each i_i, m_i and o_i represents respectively an *input, mode* and *output* argument. It is possible for any or all of *n, k* and *m* to be zero. For example, if *n=0*, the procedure has no input arguments. A procedure has also a status value *s*. Normally, at any time during its execution, a procedure returns a value for each o_j. There are no constraints either on the order or on the exact time at which output values are returned. When the procedure returns a value for the status *s*, however, the token is terminated and one or more tokens may be started. To execute a procedure the value of all input arguments i_i must be known. If so, *P* can be called and the time of invocation of *P* is the token start time. The procedure continues execution until one of two things happen: 1) a status value is returned; or 2) the agent decides to interrupt the token's execution (e.g., because the token has timed-out, i.e., the current time is equal to the latest end time of the token). The time at which this happens is the token end time. While inputs, outputs and status play an active role in the execution of a token, the mode arguments play only an indirect role. Their value is not monitored at execution but can be arbitrarily modified by a planning activity at any time during the agent's problem solving.

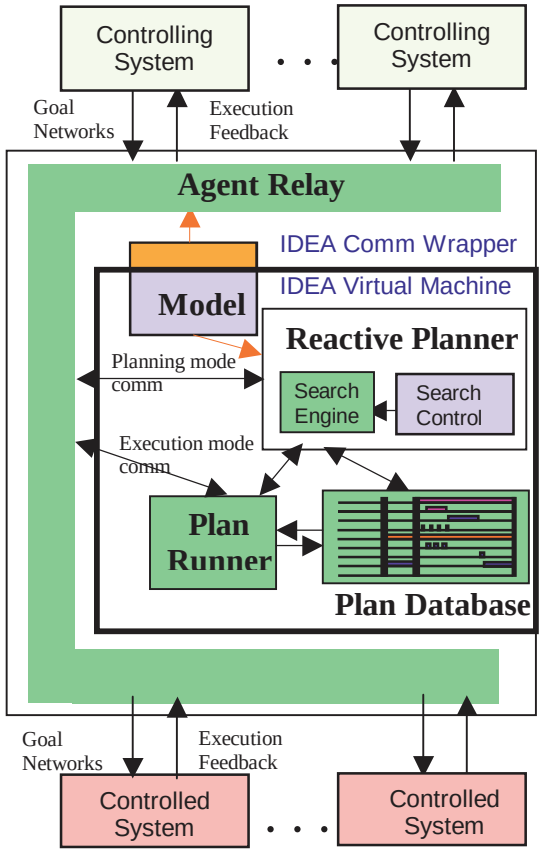

Figure 2: Structure of an IDEA agent

Communication Wrapper and Virtual Machine

Figure 2 gives an overview of the basic components of an IDEA agent. The agent communicates with other agents (either controlling or controlled by the agent) using a communication wrapper. The function of this wrapper is to send messages that initiate the execution of procedures by other agents or to receive goals that are treated by the agent as tokens. The arguments and the start and end time of each received token are treated as parameters used by the internal problem solving of the IDEA agent to decide what to do next. An IDEA agent can communicate with multiple agents both controlling and controlled. Moreover two agents could mutually control each other. Therefore, there is no restriction on the communication topology of a multi-agent system implemented with IDEA agents.

The format of the allowed communications is governed by the central *Model* that describes which procedures can be exchanged with which external agents. It also specifies which procedure arguments are expected to be determined before a goal is sent to another agent (input arguments) and on which arguments the agent is expecting execution feedback from some other agent executing the token (output and status arguments). As we will see this model is also central to the functioning of the virtual machine. To communicate with other agents the relay relies on an underlying inter-process communication mechanism that is not part of IDEA proper. Our current implementation relies on the IPC package from CMU (Simmons and James 2001) and we are also exploring the use of real-time CORBA (Real-Time CORBA 2002).

Plan Database and Model

The IDEA agent executes tokens only after they have appeared in a plan maintained in a central database. This can happen either because a controlling agent has communicated new goals or because some internal planning (reactive or deliberative) has generated appropriate subgoals. Although our reference implementation is based on the constraint-based EUROPA planning technology (Jonsson 2000), the use of different planning technologies is possible as long as they satisfy IDEA's requirement. In particular, the database must be partitioned into a series of parallel timelines, each representing the evolution over time of a dynamic property of a subsystem. To be considered for execution, a token must lay on an appropriate timeline. Sequences of tokens on a timeline will be executed sequentially and in parallel with tokens on other timelines. From now on we will continue discussing IDEA assuming the existence of a sophisticated constraint representation and propagation in the database, although this is not a strong requirement of IDEA.

At any point in time, the *Plan Database* describes the portion of the past that is remembered, the tokens currently in execution, and the currently known future tokens, including all the possible ways in which they can execute. Each token parameter (input, mode, output, status, and start and end time) has an associated variable. All these variables are connected by explicit constraints into a single constraint network. For example, the start and end time variables of each token are always related by an explicit duration constraint. The network implicitly restricts the possible value of each argument. The constraint database provides constraint propagation services that impose appropriate levels of consistency (e.g., arc consistency or path consistency) and can restrict

the range of variables to appropriate sets of values (possibly a singleton). For example, consider a simple case with two timelines, one representing the actions of a robot and the other representing the state of the robot's on-board battery. The plan may contain a robot action:

recharge ([10, 20]→ →; nominal)

This takes as input the level of charge of the battery, has no mode and output arguments, and is expected to return in a nominal state. The [10, 20] range means that the actual value of the input battery state of charge must be between 10 and 20 units for the procedure to be legally executed. The exact input value could be obtained by executing a token *read_state_of_charge (→ → soc; s*) on the battery state of charge timeline. Such a token could be present in the plan and constrained to execute before the recharge token. The communication of the state of charge between the two procedures can be obtained by a co-designation constraint between the output of *read_state_of_charge* and the input of *recharge*.

Tokens and constraints between them must respect the requirements of the agent's central domain Model. For example, the domain model could contain the constraint that before recharging the battery, it is necessary to *read_state_of_charge* from the battery. If this is the case, then *recharge* will not be executable unless such model constraint is satisfied in the plan at the time of execution.

Procedures can be executed only if the value of their parameters is consistent with the plan database constraints. The framework does not require that all database constraints be fully consistent at all times. It is possible to allow model constraints to be unsatisfied or for constraints to be inconsistent. The only consistency requirement is local and pertains to all the tokens that are currently being executed, about to be executed or that have already completed execution. This situation is similar to classic repair-based scheduling methods, where the scheduler can relax some constraints in the plan and attempt to satisfy them later. Since inconsistencies can only involve future tokens, the agent should have a reasonable belief that there will be a way to fix the inconsistency before the future tokens involved are executed. However the latter is not a strong requirement in this framework since usually it is possible to degrade performance by rejecting lower priority goals.

Generating and Running Plans

The core execution component of the agent is the *Plan Runner*, an extension and generalization of the RA plan runner. The plan runner is very simple so that it can be extremely efficient and easy to validate.

The Plan Runner is activated asynchronously when either a message has been received from another agent (e.g., a new goal is being communicated or the value of an output parameter becomes available for execution feedback) or an internal timer has gone off (e.g., the maximum allowable duration of a token has been achieved). When the Plan Runner wakes up, it makes the messages available for inclusion in the Plan Database and then immediately calls a *Reactive Planner*. The Reactive Planner has the responsibility to return with a plan that is locally executable. The planner is essentially in charge of guaranteeing two conditions: (1) consistency of token parameters with the plan constraints; and (2) support for the token according to the domain model.

Checking plan constraints is obtained as part of the constraint propagation within the Plan Database. This automatically communicates the effect of a received output or status value to the unexecuted part of the plan. Similarly, the actual end time for a token is propagated to the rest of the plan.

Checking model support for a token requires guaranteeing that a new token must start when the token immediately preceding on the timeline ends. Before starting a new token and invoking the new procedure, the Reactive Planner checks whether the token is indeed supported by the model in the plan. This means that there must be a set of constraints in the plan that corresponds to a set of requirements necessary for the token execution according to the model. If this is the case, the Reactive Planner may further constrain the procedure's arguments so that it can be called during the current execution cycle. This may require constraining the input variables so that all of the procedure's input arguments are bound to a single value. If so, the Plan Runner starts execution of the token procedure with the input variable found in the plan.

It may be that one of the two conditions above is not satisfied. This can happen, for example, if the output returned by a procedure does not match the set of possible return values in the plan, or if some model constraints are missing in the plan. For example, the plan runner may be on the verge of executing a *recharg*e token but the plan may not have an explicit constraint connecting recharge with a specific past *read_state_of_charge* token. In this case the Reactive Planner has the responsibility to fix the plan so that execution can continue. This may involve resolving execution exceptions (such as the missing constraint between *recharge* and *read_state_of_charge* described before) or refining future token parameters on the basis of the information received during the execution of current tokens (e.g., decide to execute a token as early as possible because of the value of some received output argument).

The total cycle time of the Plan Runner and Reactive Planner is bound by a fixed amount of time, the execution latency (Muscettola et al. 1998). The Plan Runner is expected to wake up, process all received messages, call the Reactive Planner, receive termination notification from the Reactive Planner, send appropriate messages to external agents and suspend itself within the execution latency. If this does not happen, then the agent will have irrecoverably failed and some low-level fault protection behavior will have to take over control. This hard requirement ensures that the agent will operate within a well-defined real time guarantee, a condition that is usually overlooked in intelligent agents research but is crucial to the design and implementation of a viable embedded control system.

Reactive and Deliberative Planning

IDEA allows the use of several planning modules in the same agent, each potentially using a different internal logic and working with a different scope. All of these modules satisfy the same input/output behavior: given an initial plan database, a planner generates a new plan

database that satisfies some given plan quality criterion (Jonsson et al. 2000). For example, the plan quality criterion may require that all tokens present in the initial plan database be present in the final plan and be fully causally supported. This may require removing inconsistencies present in the initial state, and generating new tokens and constraints according to the requirements of the domain model. A planner can be invoked in a reactive or proactive fashion. The first case occurs within the execution cycle of the Plan Runner, the second when the agent anticipates potential problems in the future and asks the planner to intervene. Deliberative planning can also be invoked to produce a high quality plan for a future horizon (e.g., an optimized observation plan for the next day), an activity that cannot be adequately carried out within the reactive execution latency. We will discuss later how this can be accomplished. Here we want to point out that there is no limitation on how small a planning problem could be, provided that the generated plan resolves any local plan flaw that was present in the plan before the invocation of the planner. For example, consider our example of an unsupported recharge token. The plan database may contain a previously executed token that invoked *read_state_of_charge*. On the basis of the domain model it may be determined that the result of that procedure invocation is still viable as an input to *recharge*. Therefore, the planner may simply create the temporal constraint and the parameter co-designation constraint from *read_state_of_charge* to *recharge*. Subsequent constraint propagation will assign a unique value for the input parameter of *recharge*. The plan quality criterion may allow the planner to stop and signal the resolution of the flaw. The Plan Runner can now resume execution and start execution of *recharge*.

Implications of the New Framework

Centrality of the model

The proposed framework strongly relies on a single, core domain model semantic. Unlike RA where models were internal to each layer and could have very different semantics, the common IDEA "modeling assembler" forces all agents to share the same semantics. At present, the modeling language used is the DDL language used in the PS model of the DS1 Remote Agent (Jonsson et al. 2000). Layering of the agent's functionality depends on partitioning the overall model into groups of timelines of different abstraction levels, each being the responsibility of a separate IDEA agent. For example, RA EXEC's action decomposition procedures are implemented by simply specifying an appropriate set of timelines and constraints in the model and by relying on fast, reactive planning for next action selection (see below). Partitioning a model among several agents is important to appropriately balance the responsiveness of each control agent with its ability of taking into account more complex constraints and longer horizons when deciding the next step. For example, a decision on what scientific observation to execute next at the highest level of abstraction may require looking ahead several steps in the current plan. This means that the reactive behavior at the higher level may require a relatively large execution latency (e.g., 10 seconds). At the lowest level, however, devices may have to be controlled with a much shorter latency (e.g., responding to a fault within tens of milliseconds). This may limit the amount of interactions and look-ahead that an agent will be allowed to take into account, trading off responsiveness for myopia. The coordination between different agents at different levels of abstraction allows us in principle to achieve the best compromise and design of the overall control system. Defining a robust methodology of the design of such a multi-agent, multi-latency control system is a current area of research.

In each agent, the plan database always checks consistency with the domain model. For example, a planner can lay a procedure invocation on a timeline only if the procedure type is associated with the timeline in the model. Also, the plan runner refuses to execute a token that is locally inconsistent or with partially supported model requirements.

The model can be acquired incrementally (i.e., one requirement at a time) during system design and engineering and at any time it contains all of the known constraints and desired behaviors in nominal and fault-protection conditions. Having the model as a single locus for this information and making the model directly usable by automatic reasoning systems (e.g., the planners) makes this knowledge directly usable at execution. This is in contrast to traditional software practices for complex systems (e.g., spacecraft flight software), where there is always a significant gap between specifications (in natural language or other semi-formal format) and implementations (a low-level language such as C or C++).

Reactivity

Even within a single IDEA agent, one important aspect is its *reactivity*, i.e., the time needed by the agent to decide what to do next in a way consistent with its predictions and with its goal. As we mentioned before, short response times depend on limiting the scope of the planning problem. Selecting the next action may require significant effort, requiring the intervention of a "deliberative planner" to bridge the gap between the current state and the goals. However, in general the amount of planning effort depends on the required level of plan quality (e.g., your next action must guarantee achievement of all future goals with minimal resource usage), on how much information is available before planning, and on the uncertainty on the values returned by procedure executions. In several cases the model may force the choice of the next action (e.g., turn on the heater if the temperature is too low) but the information needed to make the decision may not be available ahead of time (e.g., while the agent is keeping the temperature in range, it does not know future temperature changes and, therefore, whether it will need to turn on the heater or the cooler next). In this case planning may just need to determine the next token and, therefore, may need very little time. Later we will discuss how more expensive planning is integrated in the agent's behavior.

Time-bounded response

One of the critical parameters in this agent framework is the *execution latency*, i.e., the time needed by the plan runner to terminate execution of a token and start execution of the next on a timeline. At first this would appear to severely restrict the amount of intelligence that an agent can bring to bear when reacting to faults. If we look closer, however, this requirement simply states that a subsystem (timeline) can remain without commanding for a maximum amount of time equal to the latency. This requirement is equivalent to establishing a minimum sampling rate in a traditional control system. The agent can react intelligently by relying on a number of pre-compiled alternative solutions (scripts). When invoked, the planner could quickly select a script by matching its plan database with the script applicability conditions. Then, the planner could download the first token in the script and immediately signal the plan flaw resolution so that the plan runner can resume. Subsequently, the planner can download the remaining tokens in the script. This script interpretation (together with local replanning to react to new sensor data) essentially describes the functionality of the action execution capabilities of the RA EXEC module.

In some situations there may not be a planner (scripted or not) that can respond within the latency. In this case the system will need to provide a "standby procedure", i.e., a procedure or combination of procedures that maintains a safe state while the planner addresses the original plan flaw. Once the planner solves the problem, the system can exit the standby state and continue nominal execution. Note that the standby procedure, the planner behavior and the "standby exit" procedure are all described in the domain model and must be loaded into the plan database like any other procedure. In other words, standby is a concept that is explicitly modeled like any other system requirement. The planner will decide to go into standby within the latency time. This will gain enough time to take the next steps in a deliberate way.

Modeling the control system

Although a planner may need more time than the latency to modify the plan database, no special architectural support is given for this deliberative activity. For example, the agent may need to call the planner before a predicted plan flaw will actually appear in execution. 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. The model may also include constraint requirements for "planned" planner invocations (Pell et al. 1997). For example the model may say how to evaluate the time needed by the planner to produce a solution, and it may require that planning does not occur in parallel with other CPU intensive activities. Proactive planner invocations will therefore appear in a plan. In summary, our framework does not "hard-wire" the relation between reactivity and deliberation but allows explicit programming of the interaction policy with a much wider and adjustable range of possibilities.

Final Remarks

It is commonly accepted that reactive and deliberative behaviors in an agent require very different representations and inference mechanisms. The framework discussed in this paper aims at providing both capabilities within a single, simple representational, planning and execution framework. This unification is based on the observation that "planning" can be arbitrarily simple for an appropriate definition of a planning problem. This can include the selection of the next action to execute from a script, a typical operation performed by procedural executives. IDEA aims at supporting all functionalities of the Remote Agent architecture. We have generated an implementation of IDEA using the EUROPA planning technology. We have re-implemented the high-level control layer of the Remote Agent and are currently applying IDEA to other applications such as the control of an interferometry testbed at the Jet Propulsion Laboratory and an analysis of the low-level fault protection system for the Deep Space 1 and Deep Impact spacecrafts from JPL.

References

D. E. Bernard, G. A. Dorais, C. Fry, E. B. Gamble Jr. B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, P. P. Nayak, B. Pell, K. Rajan, N. Rouquette, B. Smith, B. C. Williams, "Design of the remote agent experiment for spacecraft autonomy." In *Proc. of the IEEE Aerospace Conference*, March 1998.

R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack, "Experiences with an Architecture for Intelligent, Reactive Agents", *J. of Experimental and Theoretical AI*, January, 1997.

K. W. Currie, A. Tate. "O-Plan: the Open Planning Architecture," *AI Journal*, 52(1), pp. 49-86, 1991.

Erann Gat, "ESL: A language for supporting robust plan execution in embedded autonomous agents," Proceedings of the AAAI Fall Symposium on Plan Execution, AAAI Press, 1996.

A. Jonsson, and J. Frank, "A Framework for Dynamic Constraint Reasoning using Procedural Constraints, in Workshop on Constraints in Control, part of the 5th International Conference on Principles and Practices of Constraint Programming, (CP99), 1999.

A.K. Jonsson, P. Morris, N. Muscettola, K. Rajan, B. Smith "Planning in interplanetary space: theory and practice", in *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems* (AIPS '00), Breckenridge, Colorado, 2000.

Phil Kim, Brian Williams, Mark Abramson. Executing Reactive Model-based Programs Through Graph-based Temporal Planning, *Proc. of IJCAI 2001*, Seattle, WA, 2001.

N. Muscettola, P. Morris, B. Pell, B. Smith, "Issues in temporal reasoning for autonomous control systems." In Proc. of the Second Intl. Conf. on Auton. Agents (AGENTS'98), Minneapolis, MN, 1998.

P. P. Nayak, D. E. Bernard, G. Dorais, E. B. Gamble Jr., B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, K.

Rajan, N. Rouquette, B. D. Smith, W. Taylor, Y. W. Tung, "Validating the DS1 Remote Agent Experiment", in *Proc. of the Fifth Intl. Symposium on Artificial Intelligence, Robotics and Automation n Space* (iSAIRAS'99), pp. 349, 356, Nordwijk, The Netherlands, June 1999.

Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith. "Robust Periodic Planning and Execution for Autonomous Spacecraft." In Proceedings of IJCAI, 1997.

Barney Pell, Ed Gamble, Erann Gat, Ron Keesing, Jim Kurien, Bill Millar, P. Pandurang Nayak, Christian Plaunt, and Brian Williams. "A Hybrid Procedural/Deductive Executive For Autonomous Spacecraft." In Autonomous Agents and Multi-Agent Systems, 2:1:7-22 1999.

Real-time CORBA with TAO (The ACE ORB), http://www.cs.wustl.edu/~schmidt/TAO-intro.html

Reid Simmons, David Apfelbaum. "A Task Descriprion Language for Robot Control", in *Proc. Conference on Intelligent Robotics and Systems, 1998.*

Reid Simmons, Dale James, *Inter-Process Communication v3.4*, Carnegie Mellon University, February 2001, available at http://www-2.cs.cmu.edu/afs/cs/project/TCA/ftp/icp.ps.gz

D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley, "Planning and Reacting in Uncertain and Dynamic Environments", in *Journal of Experimental and Theoretical Artificial Intelligence*, 6:197-227, 1994. See also: http://www.ai.sri.com/people/wilkins/papers.html

B. C. Williams, P. P. Nayak. "A Model-Based Approach to Reactive Self-Configuring Systems", in *Proc. of the Thirteen Nat. Conf. on Artificial Intelligence* (AAAI '96), Portland, Oregon, 1996.