# A Unified Approach to Model-Based Planning and Execution

Nicola Muscettola*, Gregory A. Dorais+, Chuck Fry+, Richard Levinson#, Christian Plaunt+

## I. INTRODUCTION

Writing autonomous software is complex, requiring the coordination of functionally and technologically diverse software modules [Bonasso et al. 97] [Currie & Tate 91] [Firby 89] [Georgeff & Lanskey 89] [McDermott 92] [Musliner et al. 93] [Simmons 92]. System and mission engineers must rely on specialists familiar with the different software modules to translate requirements into application software. Also, each module often encodes the same requirement in different forms. The results are high costs and reduced reliability due to the difficulty of tracking discrepancies in these encodings. In this paper we describe a unified approach to planning and execution that we believe provides a unified representational and computational framework for an autonomous agent. We identify the four main components whose interplay provides the basis for the agent's autonomous behavior: the domain model, the plan database, the plan running module, and the planner modules. This representational and problem solving approach can be applied at all levels of the architecture of a complex agent, such as Remote Agent. In the rest of the paper we briefly describe the Remote Agent architecture. The new agent architecture proposed here aims at achieving the full Remote Agent functionality. We then give the fundamental ideas behind the new agent architecture and point out some implication of the structure of the architecture, mainly in the area of reactivity and interaction between reactive and deliberative decision making. We conclude with related work and current status.

## II. LAYERED AGENT ARCHITECTURES:REMOTE AGENT

The Remote Agent (RA) was developed at the NASA Ames Research Center and at the Jet Propulsion Laboratory as an autonomous control system capable of long-term, high-level, 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 98]. Unlike traditional spacecraft command sequencers, RA was designed to be *goal-achieving* and *robust*. We define goal-achieving as the ability to transition the system to a specified state for a specified period of time. We define robust as the ability to overcome obstacles encountered due to uncertainty in the environment or system that would normally prevent the achievement of a goal. Finally, RA was designed so that operational constraints could be explicitly represented in models. RA would use these models to make sure that these constraints are not violated regardless of the commanded goals.

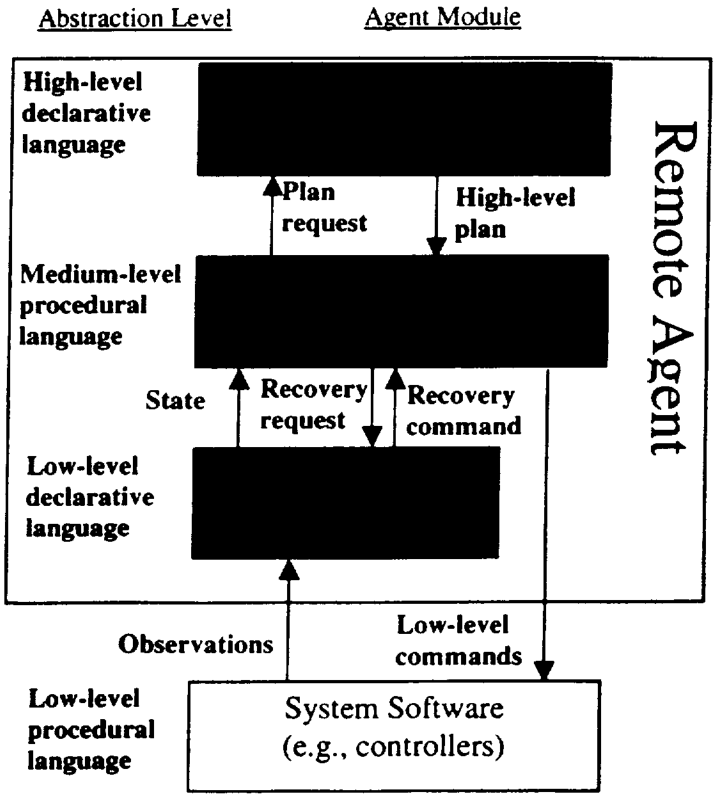
Abstraction Level
Agent Module
High-level declarative language
Plan request
High-level plan
Medium-level procedural language
State
Recovery request
Recovery command
Low-level declarative language
Remote Agent
Observations
Low-level commands
Low-level procedural language
System Software (e.g., controllers)

**Figure 1: Remote Agent layered architecture**

The RA architecture achieved these specifications by integrating a constraint-based planner/scheduler (PS) [Jonsson et al. 00], a model-based truth maintenance system with diagnosis and recovery capabilities called MIR [Williams & Nayak 96], and a reactive executive called EXEC layer [Pell et al. 99], into a layered architecture shown in figure 1. Each layer uses a different modeling

* NASA Ames Research Center, Moffett Field, CA
+ Caelum Research Corp.
# Recom Technologies Inc.

language and a different way to specify problem-solving control.

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. In order to reduce the search time to create a valid plan, developers limit the possible actions PS considers to the highest level possible without sacrificing the needed expressiveness. For the DS1 RA, which shared a 20 Mhz CPU with the System Software, the response time between a plan request and the plan from PS was between 1-4 hours.

The Executive (EXEC) comprises the second. It uses a reactive, executive language ESL, an extension of LISP[Gat 96]. ESL defines how to execute the actions used by PS. This requires specification of how to achieve the success states associated with each action using low-level commands to System Software. The system is procedurally described at level of abstraction higher than that of low-level commands in order to enhance reactivity. On DS1, EXEC was required to respond to any event it handles within 4 seconds in the worst case.

EXEC relies on MI (Mode Identification) to do state estimation and to notify EXEC when a state it is watching changes. In order to do this, MI must model the system at a lower level than EXEC. For example, MI may need to model interactions between several sensors in order to determine whether the state of a thruster is ON or OFF. Because MI is in the control loop between EXEC and the System Software, it is important that MI be responsive. It achieves this be limiting its low-level modeling to a qualitative, declarative language. MR (Mode Recovery) is used to determine the least costly path from the state MI estimates the system is in to the one EXEC specifies it should be in according to the plan without passing through states EXEC marked as invalid. For DS1, the maximum response time for MR was 5 seconds.

Using different problem solving modules with different representation languages at each level 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 at each level and to insure that they did not conflict.

## III. A UNIFIED ARCHITECTURE

Analysis of RA's functionality has shown that it should be possible to provide a single, unified framework that can then be used to provide functionality at each of RA's levels of abstraction.

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

$$P(i_1, i_2, \ldots, i_n \rightarrow o_1, o_2, \ldots, o_m; \ s)$$

Each $i_i$ represents one of the *n* input arguments ($n >= 0$), while $o_j$ is one of the *m* return values ($m >= 0$). A procedure always returns a status value *s* whose main function is to signal an exception due to anomalous execution conditions of *P*. Normally, a procedure returns a value for all of its $o_j$ return values and the value *nominal* for *s* at the same time of the last returned $o_j$. 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 it returns, either in a nominal or exceptional state. This time corresponds to the token end time.

Figure 2 gives an overview of the basic components of the new architecture. The agent executes tokens only after they have appeared in a *plan* maintained in a central constraint database [Jonsson & Frank 99]. The database is partitioned in 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 token on other timelines.

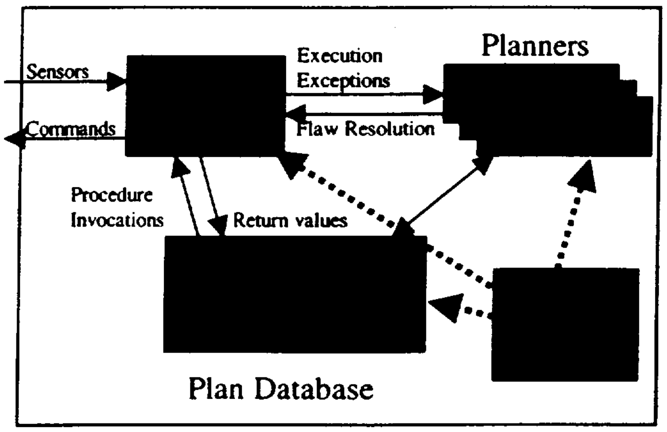

**Figure 2: Unified Agent Architecture**

Therefore, each timeline can be seen as a thread of execution that calls procedures according to the tokens in the plan.

At any point in time, the constraints in the plan describe the possible ways in which future tokens can be executed. Each token parameter (input argument, return value, start and end times) has an associated variable. For example, there is a variable for a token start and one for its end, and a variable for each input and output parameters. All these variables can be connected by explicit constraints. For example, the start and end time variables of each token are related by an explicit duration constraint. The constraint network implicitly restricts the possible value of each argument. Associated with the constraint database are constraint propagation services that, by imposing appropriate levels of consistency (e.g., arc consistency or path consistency), 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 and is expected to return in a *nominal* state. The fact that the input argument has an associated *[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 in the plan. The exact value of the input parameter could be obtained by executing a token *read_state_of_charge( → soc)* 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*.

There are two possible sources of plan database constraints. The first source is an external user who communicates goals to the agents as (partial) networks of tokens. The second is the agent's *model* of the domain. 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 *recharging* 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. This does not necessarily mean that constraints coming from all tokens in the plan must be completely consistent. 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 the token that is currently being executed or about to be executed. Therefore, the agent can tolerate plan database inconsistencies as long as their effect does not impact the procedures currently in execution. This could be obtained similar to classic repair-based scheduling methods, by relaxing some constraint in the plan and attempt to satisfy it later. The agent should also have a reasonable belief that there will be a way to fix the inconsistency before the tokens involved are considered for execution. However the latter is not a strong requirement of the agent architecture since usually it is possible to degrade performance by rejecting lower priority goals.

The core execution component of the agent is the *plan runner*. The plan runner is very simple so that it can be extremely efficient. Its only responsibility is to terminate the execution of a token and start the execution of the next on the timeline. This process requires checking 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 done locally when the executing token terminates. Checking is local and translates into verifying that the actual procedure return values and the token end time fall within the set of possible values identified by the plan. If so, then the domain of the variable associated with the return value is restricted to the actual return value and constraint propagation in the plan database automatically communicates the effect of this value to the 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 is done before starting its execution. A new token must start when the token that immediately precedes it on the timeline ends. Before invoking the token procedure, the plan runner checks whether the procedure is indeed supported by the model in the plan. This means that there must be a set of constraints in the plan that correspond to a set of requirements necessary for the token execution according to the domain model. If this is the case, the plan runner checks whether the procedure can be called. This requires all of its input variables to be bound to a single value. If so, the plan runner calls the token procedure with the input variable found in the plan.

It can be that one of the two conditions above may not be satisfied. This can happen if the output returned by a procedure does not match the set of possible return values anticipated in the plan, or if some constraints required by the model are missing in the plan. For example, the plan runner may be on the verge of executing a *recharge* 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 plan runner interrupts execution and signals an appropriate exception. In the *recharge* token case, the exception will signal that there is a "model flaw" in the plan database that needs to be fixed. Within a fixed amount of time, the *execution latency* [Muscettola 98], the plan runner is expected to receive a notification that the flaw has been fixed and complete the initiation of the new procedure. If this does not happen, then the agent will have irrecoverably failed and some low-level fault protection behavior will take over control.

Resolution of the execution time exceptions is the responsibility of the third component the architecture, the *planner*. Actually, the architecture allows for the use of a set of planning modules, potentially using different internal logic to work with different scopes. 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 00]. 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 by rescheduling tokens, 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 after an execution time exception, the second when the agent anticipates potential problems in the future and asks the planner to intervene. 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 the plan flaw that caused the invocation of the planner in the first place. 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 propagate a unique value for the input parameter of *recharge*. The plan quality criterion may now allow the planner to stop and signal the resolution of the flaw. The plan runner can now resume execution and call *recharge*.

## IV. IMPLICATIONS OF THE NEW ARCHITECTURE

### A. *Centrality of the model*

The proposed architecture is strongly based on a single, core domain model. The plan database always checks the consistency with the domain model. For example, the only procedure invocations that a planner can lay on a timeline are those whose type has been associated with the timeline in the model. Also, the plan runner refuses to execute a token inconsistent with the plan or not fully supporting domain model requirements. Reliance on an explicit model provides a strong basis for the formal validation of the overall agent behavior in a specific domain. The model can be acquired incrementally, one requirement at a time during system design and engineering. The model is the place where constraints and desired behaviors for fault protection can be gathered. Using the model as a single locus of this information and using a modeling approach that makes this information directly usable by automatic reasoning systems (e.g., the planners) has significant advantages with respect to the current practices to complex system design and implementation (e.g., spacecraft flight software). In the traditional approaches there is always a significant gap between specifications (in natural language or other semi-formal format) and implementations (a language in some low-level implementation language such as C).

### B. *Reactivity*

One important aspect of practical agent architectures is the amount of time needed by the agent to decide what to do next in a way consistent with its predictions of how the world will evolve and with its goal. This architecture allows tuning of this time by providing support for reactive decisions at two different levels. At the lowest level, reactivity is obtained by propagating procedure return values in the plan constraint database as soon as they become available. Short response times here depend on the characteristics of the constraint propagation algorithms and on the size of the plan. For certain classes of constraints, plan constraint networks can be compiled to allow extremely fast propagation.[Tsamardinos et al. 98].

At a higher level, reactivity depends on a planner responding to an execution time exception. Here short response times depend on limiting the scope of the planning problem. Under certain circumstances selecting the next action may require significant effort. This is what is typically addressed by "generative planning", where there is a significant gap between the current state and the goals. However, more generally the amount of effort spent in planning depends on the overall level of quality (e.g., make sure that your next action will guarantee achievement of all future goals with minimal resource usage), on how much information is available before planning, and on the uncertainty on the return value of procedures during execution. In several cases the model may force the choice of the next action (e.g., turn on the heater if the room 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 room temperature in range, it does not know how future temperature will change and, therefore, whether it will next have to turn on the heater or the cooler). In this case it may be sufficient to just determine the next token on a timeline and make sure that it satisfies all the model requirements. This planning problem can be solved in a very short period of time. Later we will discuss how more expensive, generative planning can be integrated in the agent's behavior.

### C. *Time-bounded response*

One of the critical parameters in this agent framework 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. A closer look, 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 practically equivalent to establishing a minimum sampling rate in a traditional control system. Reacting intelligently may be obtained by providing the planner with 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 complete download of further tokens in the script.

In some situations this scripting approach may not be fast enough to respond within the latency. In this case we will need to rely on appropriate system design to provide a "standby state", i.e., a safe state that can be maintained for a long enough period of time for the planner to address the original plan flaw. Once the planner solves the problem, the system can exit the standby state and continue nominal execution. Note that both the standby state, the planner behavior and the procedure to exit from standby will need to be loaded into the plan database. In other words, standby is a concept that is modeled at the same level as other requirements in the domain model. It is the decision to go into standby that the planner will take within the latency, with the intent of gaining enough time be more deliberate in taking the next steps.

### D. *Modeling the control system*

In both previous sections we have mentioned the possibility of a planner taking longer than the latency to modify the plan database. However no special architectural support is given for this deliberative activity. This is because we follow the approach of explicitly including in the domain model the requirements on the behavior of the controller itself, besides that of the controlled system. In order for the agent to call the planner ahead of the time in which the plan flaw will appear in execution, the domain model must include a model of the planner itself, i.e., a timeline that could accommodate for token whose execution explicit invokes the planner. The model may then include the constraint requirements under which "planned" planner invocation can be achieved [Pell 97]. For example the model could provide an evaluation of the time needed by the planner to produce a solution, and a requirement that planning not be done while other CPU intensive activities are scheduled. The planner can then be called proactively provided that the previous invocation of the planner has already left in the plan the token that will call the planner again in the future. In summary, rather than making a hard-wired, architectural assumption on the relation between reactive and deliberative agent behaviors, our architecture allows to explicitly "program" the interaction policy in the model, allowing for a much wider and adjustable range of possibilities.

## V. REFERENCES

[Bonasso et al. 97] R. P. Bonasso, R. J. Firby, E. Gat, David Kortenkamp, D. Miller, and M. Slack, Experiences with an Architecture for Intelligent, Reactive Agents, in Journal of Experimental and Theoretical Artificial Intelligence , January, 1997.

[Bernard et al. 97] 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. Willams, "Design of the remote agent experiment for spacecraft autonomy." In Proceedings of the IEEE Aerospace Conference, March 1998.

[Currie & Tate 91] K. W. Currie and A. Tate. "O-Plan: the Open Planning Architecture," Artificial Intelligence, 52(1), pp. 49-86, 1991.

[Firby 89] R. James Firby. Adaptive Execution in Complex Dynamic Worlds. Ph.D. dissertation. Yale University, Department of Computer Science, #672, 1989.

[Gat 96] 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.

[Georgeff & Lanskey 89] Michael Georgeff and Amy Lanskey, "Reactive Reasoning and Planning," Proceedings of AAAI, 1987.

[Jonsson & Frank 99] 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.

[Jonsson et al. 00] 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.

[McDermott 92] Drew McDermott. Transformational Planning of Reactive Behavior. Technical Report YALEU/DCS/RR-941, Dept. of Computer Science, Yale University, 1992.

[Musliner et al. 93] David J. Musliner, Edmund H. Durfee, and Kang G. Shin, "CIRCA: A Cooperative Intelligent Real-Time Control Architecture," IEEE Transactions on Systems, Man, and Cybernetics, Nov/Dec 1993.

[Simmons 92] Reid G. Simmons, "Concurrent planning and execution for autonomous robots." IEEE Control Systems, 12(1):46-50, February 1992.

[Muscettola 98] N. Muscettola, P. Morris, B. Pell, B. Smith, "Issues in temporal reasoning for autonomous control systems." In Proceedings of the Second International Conference on Autonomous Agents, ACM Press, Minneapolis, MN, 1998.

[Pell et al. 97] Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith. "Robust Periodic Planning and Execution for Autonomous Spacecraft." In Proceedings of IJCAI, 1997.

[Pell et al. 99] 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.

[Tsamardinos 98] I. Tsamardinos, N. Muscettola, P. Morris "Fast Transformation of Temporal Plans for Efficient Execution" in Proc. of the Fifteenth Nat. Conf. on Art. Int. (AAAI '98), Madison, Wisconsin, 1998.

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