V. Verma, A. Jónsson, R. Simmons, T. Estlin, R. Levinson, "*Survey of Command Execution Systems for NASA Robots and Spacecraft*", "Plan Execution: A Reality Check" Workshop at The International Conference on Automated Planning & Scheduling (ICAPS), 2005

Survey of Command Execution Systems for NASA Spacecraft and Robots

Vandi Verma, Ari Jónsson, Reid Simmons, Tara Estlin, Rich Levinson

QSS / NASA Ames Research Center
USRA-RIACS / NASA Ames Research Center
Carnegie Mellon University
Jet Propulsion Lab
Attention Control Systems Inc.

MS 269-1 NASA Ames, Moffett Field CA 94035
MS 269-2 NASA Ames, Moffett Field CA 94035
3205 Newell-Simon Hall, Carnegie Mellon University, Pittsburgh, PA 15213
M/S 126-347, JPL, Pasadena CA 91109-8099
650 Castro St., Ste 120 PMB 197, Mountain View CA 94041

vandi@email.arc.nasa.gov, ajonsson@arc.nasa.gov, reids@cs.cmu.edu, tara.estlin@jpl.nasa.gov, rich@brainaid.com

Abstract

NASA spacecraft and robots operate at long distances from Earth. Command sequences generated manually, or by automated planners on Earth, must eventually be executed autonomously on-board the spacecraft or robot. Software systems that execute commands on-board are known variously as *execution systems*, *virtual machines*, or *sequence engines*. Every robotic system requires some sort of execution system, but the level of autonomy and type of control they are designed for varies greatly. This paper presents a survey of execution systems with a focus on systems relevant to NASA missions.

Introduction

As NASA's missions become more complex, autonomy becomes increasingly important to the success of those missions. At the heart of such autonomous systems are sub-systems, known variously as *execution systems*, *virtual machines*, or *sequence engines*, that execute commands and monitor the environment. Such execution systems vary in sophistication, from those that execute linear sequences of commands at fixed times, to those that can plan and schedule in reaction to unexpected changes in the environment.

For NASA missions, sophisticated execution systems are highly desirable for several reasons. One is that NASA spacecraft and robots typically operate far from Earth, and so must take on significant responsibility for their own health and safety. Second, models of robot sensors and effectors are often uncertain and the environment is generally only partially known and may even be dynamic. To account for uncertainty even a simple deterministic sequence of commands needs to use worst-case estimates of action duration and resource use. In some cases even this suboptimal sequence may not be robust to the uncertainty.

This paper presents a survey of execution systems that have been developed for various applications. We focus on those systems that are relevant to NASA-type applications. Before presenting the individual systems, we define some terms that are critical to understanding execution systems.

An *executive* is a software component that realizes pre-planned actions. Executives are particularly useful in the presence of uncertainty. Classical executive functions include selecting an action from a set of possibilities based on the current state of the robot and environment and outcome of previous actions, hierarchical task decomposition, coordinating simultaneous actions, resource management, monitoring of states, resources command status, and fault diagnosis and recovery. One way to view an executive is as an onboard system that takes a plan that assumes a certain level of certainty and expected outcomes and executes it in an unknown and possibly dynamic environment.

A *plan* is a series of actions designed to accomplish a set of goals but not violate any resource limitations, temporal or state constraints, or other spacecraft or rover operation rules. Desirable characteristics of a plan are that it be valid, complete and optimal (or of high quality). Algorithms that can reason about achieving goals over a future time period and in the face of various constraints are called *planners*. However, a plan, as generated by most any current planner,

still requires the help of an execution system to be useful for real-world execution. Making these plans executable may not involve complex AI algorithms, but is essential for achieving the plan. In order to perform plan execution, control structures such as conditional statements that catch violated assumptions, looping constructs that can retry an action until it succeeds, and more detailed descriptions of preconditions that must be checked before an action is executed must be added. Execution languages provide constructs to represent essential plan execution information in addition to the plan.

An *execution language* is a representation of actions and plans that takes into account the state of robot and environment at the time the action is executed, and the interdependence between actions, in terms of temporal, precedence, or other constraints.

Model-based systems are represented by a knowledge-base (model) of its structure and behavior and are typically specified using a declarative representation. In other words, these models do not specify the sequence of actions required to fulfill specific high-level goals of the system, but instead they specify the expected effect each action or external event may have on the modeled state. Models are often specified in a modular manner, where only the local effect of an event is described. Planners may use these models to find sequences of actions directed toward the goal or a fault diagnosis system may use them to detect and identify faults.

Some execution systems use no automated planners; we call these *execution-only* systems. Other execution systems have explicit interfaces to planners, (through an execution language or a standard format like XML), we call these *execution systems coupled with an external planner*. Yet another class of execution systems integrate planning and execution more tightly by using a planner internally within the execution system to select control actions. We call these *execution systems with internal planner*. Note this is not a mutually exclusive classification. Some execution system may be used as an execution only system with manually encoded plan execution, but may also have well defined interfaces to one or more automated external planners that may be used in other applications. An execution system may also provide an external interface to a planner in addition to having an internal planner. Finally, note that the coupling of execution systems with external planners can differ in tightness, ranging from infrequent requests for assistance to continuous information sharing. In cases where the coupling is tight, the combined functionality is similar to integrated execution-planning systems.

Traditional Command Execution

At this time, most spacecraft and rovers are operated via sequences of commands. The command sequences are fairly simple in structure and the interpretation on board the spacecraft is straightforward. Dynamic outcomes and environmental uncertainties are handled partially by making sequences conformant to possible outcomes, and partially by relying on on-board fault detection and system health software.

In this context of traditional spacecraft operations, the executive is the flight software system on board the spacecraft; more specifically, the sequence execution system and the health monitoring and fault detection system. The execution language is simple; an execution plan is a fairly small set of branching command sequences and sub-sequences, where each command is either executed at a specific time, or immediately following the completion of another command. Typically, there are no conditionals, no loops, no constraints, etc.

The most notable properties of this approach are:

- Plans become inherently conservative, so as to be conformant to expected outcomes. For example, activities are assumed to take the longest they can possibly take.
- The on-board health monitoring system is limited to general responses to failures, which often leads to unnecessary execution aborts and spacecraft operations halts. For example, a certain failure might lead to abandoning the whole plan, whereas portions of the plan could still be safely continued.

Virtual Machine Language (VML) [14] is a sequencing language that has flown on numerous NASA spacecraft. VML is currently in use on the Spitzer space telescope, Mars Odyssey, Mars Reconnaissance Orbiter, Dawn, Genesis, and Stardust. It is slated for future New Frontier and Discovery class missions, including the Mars Telecom Orbiter and possibly the Mars Science Lander.

VML is an execution language that was developed to take into account the needs of spacecraft operations. It provides a "safe-sandbox", with the aim of shielding operations personnel from most of the mistakes possible in contemporary programming languages like C. Sequences are procedural, and have symbolic names. At any time only one instruction is active in a sequence engine (also known as a virtual machine). The language accommodates a variety of spacecraft commanding architectures. It features absolute and relative constraints, event-driven sequencing, programmable delays, arithmetic and bit-level operations, parameters with polymorphism, and a number of numeric and string data types. VML dynamically builds spacecraft commands with values derived from variables, and has reusable blocks that can be called or spawned from sequences. The on-board sequencing component can also

be configured to access telemetry values for use within sequences.

The VML language is compiled to an uplinkable file form in a Unix-based ground system by the VML Compiler. This process translates human-readable text into a binary file for interpretation onboard by the VML Flight Component. The flight component is implemented in C for compatibility with the widest possible range of missions. In addition, a Unix tool known as Offline VM (OLVM) is available for ground-based execution and debugging of developed blocks and sequences. OLVM encapsulates the actual flight code for high fidelity testing with very fast turnaround when developing using VML.

Execution Systems

This section presents several NASA-relevant execution systems, in alphabetical order.

Apex

Apex [10] is an execution system and has been used in numerous large-scale applications including control of real autonomous helicopters, control of simulated aircraft for wildfire detection, and in simulating humans for Human Computer Interface (HCI) analysis.

Apex is a reactive execution system that selects for execution one or more procedures (partial plans) from its library of procedures at each execution step. In most applications Apex has been used as an *execution-only* system. Apex is designed to unify plan-running and mission-management functionality. Planners may be called on to produce or extend a mission plan, to solve a local planning problem within a mission plan or both. Apex may therefore potentially be used as an *execution system coupled with an external planner*.

The execution language used by Apex is the Procedure Description Language (PDL). PDL can represent a hierarchical decomposition of a high-level task into basic primitives, event driven floating contingencies, and also calls to Lisp (the underlying programming language). A PDL procedure consists of a unique identifier, a description of a class of goals the procedure applies to, and one or more step clauses. The step clauses are concurrently executable and may call other procedures (sub-tasks).

The input to Apex is a set of human-fabricated procedures represented in PDL. Apex is a reactive system that chooses an action at every execution step. Key capabilities of the executive (and of PDL) are:

- Monitoring/querying for complex temporal events patterns
- Opportunistic (reactive) task refinement and resource allocation
- Management of concurrent and periodic tasks

Continuous reaction allows Apex to use the most recent measurements to guide the selection of the next action. In addition it allows dynamic update of high level goals. Apex also provides a number of tools for debugging, demonstration, and monitoring.

CRL and C-CRL Executive

The *Contingent Rover Language* (CRL) [4] is a declarative plan execution language that was designed to represent contingent plans. It uses a hierarchical representation and can represent simple and floating branches, nesting, flexible time, and state and resource conditions. The CRL executive has been used on NASA's Marsokhod, ATRV, and K9 rovers as a high-level plan interpreter. It has also been used with the Mission Simulation Facility (MSF) rover simulator. C-CRL is an extension of CRL that is capable of concurrent execution and has been used for the single-cycle instrument placement demonstration on the K9 rover [21].

The CRL executive may be used as an *execution-only* system with manually written CRL constructs. The external planner that generates CRL plans is the PICO contingent planner [5]. CRL does not support loops and periodic tasks, or have a mechanism for providing feedback to planners.

IDEA

Intelligent Distributed Execution Architecture (IDEA) [20] is a model-based planning and execution system. One of the two glitches experienced by Remote Agent was due to undocumented and subtle differences in semantics between models in the planning, execution and diagnosis layers. IDEA was developed to address this problem by building an architecture that supports controllers/planners at multiple levels of abstraction. Controllers (agents) at every level of abstraction share the same model. The semantics of the structure of a task, the structure of an execution cycle responsible to activate a task in response to an asynchronous or synchronous event, the structure of events communicated between controllers, how the communication of tasks maps into the transport layers responsible of delivering them across agents, are thus uniform. Each controller (control agent) at every level of abstraction is assumed to perform planning as the sole computational process to decide how to respond to events.

IDEA uses the classic sense-plan-act cycle. One of the novel features in IDEA is the use of an on-board planner from first principles (i.e., the sub-goaling model) to plan for a limited horizon into the future and execute the current task at hand simultaneously. The advantage is that this allows it to dynamically update the plan based on the

current state of the world and previous actions, which can yield a wider range of robust behaviors than possible with traditional execution scripts. The disadvantage of using planning from first principle at every execution cycle is that patterns of constraints (temporal and parametric) are always assembled from scratch, causing higher latency than possible when using pre-compiled execution scripts. Consequently, execution may halt if the planner can't deliver a response in time. IDEA agents can also use an arbitrary number of deliberative planners to optimize agent behavior over a long, future horizon. IDEA is thus an *execution system with internal planner* (reactive planners) and may also be used as an *execution system coupled with an external planner* (deliberative planners) at the same time.

XIDDL is the execution language used in IDEA. It is a modeling language amenable to temporal/hybrid planning through subgoaling, used to describe the model of the world, the internal logic and the input/output behavior of each IDEA controller. This uniformity aims to facilitate system-level validation for an autonomy system without the need for understanding the details of each specific controller, since it is expensive and error prone to assume that mission personnel will examine software written in different computer languages in order to ascertain its ability to satisfy mission requirements. IDEA has been used for autonomously controlling a telescope, PSA (personal satellite assistant), and a number of mobile robots.

While IDEA is designed to use any planner that uses a representation that is compatible with the XIDDL modeling language, all of the IDEA systems developed so far use the Europa planning technology [9] both for reactive and deliberative planning.

MPE

Mission Planning and Execution (MPE) [1] is the execution subsystem of the Mission Data System (MDS) [24]. MDS uses an explicit state-based representation. Knowledge about the spacecraft and the environment is provided by state estimates. Knowledge about the behavior of the system is stored in state models. Information is reported via a history of states, measurements, and control commands. The input to MPE is operator "intent" (expressed as temporal constraints, and constraints on states), flight rules, and hard constraints on variables. MPE is an *execution system with internal planner* that can locally adapt the original plan to recover from faults and handle uncertainty.

PROPEL

Program Planning and Execution Language (PROPEL) [17] [18], is a unified planning and execution system that uses a procedural representation. This is different from IDEA, which exclusively uses a declarative action representation.

The motivation was that since most software is not written as a declarative model it tends to be outside the scope of a planner's reasoning. PROPEL was designed to increase the scope of the planner's model to include software in order to address the problem of software failure detection and recovery.

Propel was designed to close the gap between the declarative action model used by a planner and the procedural languages used to develop real-world software. The representation is intended to be expressive enough to be used in system software including the planner and executive software. Motivation for using a procedural representation includes:

- Desire to include all software within the planner's model in order to increase the scope of failure recovery to include infrastructure software failures.
- Desire to represent complex procedures including loops, conditionals, local variables, and multiprocessing.
- Desire to reduce the need to develop and maintain different models for the planner and execution system.
- Reduce risk of loss of information in translation between execution and planning (and vice versa).

Propel is both an architecture and a language. The architecture provides integrated planning and execution modules that monitor and manipulate application-level processes written in the Propel language. The language is a library of methods for embedding search and temporal constraint information into C++, thus creating a "superset" of C++ like TDL. This library provides an interface from the Propel application code to the supervisory meta-processes (the planning and execution modules), which monitor the application to provide failure detection and recovery.

The language provides an action representation that captures control constructs and can also be projected by a search-based planner. The planner can provide a useful partial plan even when it is interrupted after an arbitrary amount of computation. The planner and the controller share identical data structures and algorithms for interpreting a shared representation of control actions. PROPEL is an *execution system with internal planner*.

PRS

The *Procedural Reasoning System* (PRS) [12] was developed to address the problem encountered in developing autonomous systems that were required to be continuously active and have real-time response. Traditional programming languages imposed an order on

task execution through the language's control structure that makes it difficult to respond quickly to a large set of possible events.

PRS is a reactive goal-driven system that selects procedures (partial plans) from its library of procedures at each execution step. PRS is an *execution-only* system.

PRS has a knowledge-base of procedures. Each procedure requires the specification of an event, the state of the world that will trigger that event, the steps that are executed by the procedure, and the sub-goals that it achieves.

PRS has been used on a number of mobile robots and also in a simulation of the space shuttle. PRS was originally written in Lisp and is now known as PRS-CL. The C version of it is called C-PRS or *Propice* [15].

RAP System

Reactive Action Package System (RAPs) [8] was designed to support reactive planning and execution. It is a representation language for general-purpose execution. It uses a Lisp-based interpreter to manage a task network and to interface to a behavioral layer. RAPs may thus be used as an *execution-only* system or *execution system coupled with an external planner*.

The main idea behind RAPs is that all capabilities of goal-achieving behaviors – task decomposition, different tactics for achieving a goal, monitoring, error recovery, checking of pre- and post-conditions – should be represented in a single "package." Each RAPs is thus a self-contained module that knows how to achieve a particular goal in the face of uncertainty.

The RAPs execution system uses a library of goal-achieving behaviors and a symbolic world database to choose which RAPs to execute, how to decompose them, and when they succeed or fail. The execution system schedules RAPs according to their priority and temporal constraints, interrupting execution of one RAP if higher priority RAPs become active.

Remote Agent (RA) Executive

Remote Agent [22] is an AI system that flew on-board the Deep Space One (DS-1) spacecraft in 1999. The main characteristics of the Remote Agent are that it is model-based with on-board planning, fault detection, identification, and recovery.

The executive in the Remote agent [23] is the central controller. The input to the Remote Agent executive is a high-level state and duration for which the state must be maintained. The executive autonomously calls the planner to generate a plan to satisfy a high-level goal. It uses a domain model to monitor plan execution and commands the planner to generate an updated plan if any of the constraints are violated during execution.

The Remote Agent executive was based on the *Execution Support Language (ESL)* [11]. ESL is a declarative execution language that is an extension of Lisp. It is implemented as a set of macros that expand into Common Lisp and invoke Lisp's multi-tasking library. ESL provides task-level control constructs, resource management, and a database built on Prolog

The novel features demonstrated by the RA executive in the DS-I experiment were integrated planning and execution with low-latency response time to contingencies and deficiencies in the plan and the lack of intervention required by the human operator after issuing high level mission goals. The RA executive is an *execution system with internal planner* and also an *execution system coupled with an external planner* at the same time.

One of the main challenges with this approach is building and maintaining models. The emergent behavior that results from subtle interactions between qualitative models of weakly interacting subsystems is hard to predict since the range of input conditions and responses are extremely large. "Incorrect knowledge in the domain model could endanger or even lose the mission" [2].

RMPL, Titan, Kirk, Moriarty

Titan [29] is a model-based execution system that supports both execution control and model-based goal achievement specifications. The execution control component generates goal states, which are then given to the model-based goal achievement component. The goal achievement component uses automated diagnosis methods to estimate the current state from observable data (*mode identification*), and then uses automated planning (*mode reconfiguration*) to generate command sequences to achieve the given goals from the current state.

The execution language used in Titan is the *Reactive Model-based Programming Language* (RMPL) [28]. It is used to specify both the control information used by the control component and the model-based state estimation and planning component. The control information supports control constructs such as loops, conditions, iterations and contingencies, over model-based specifications of concurrent and sequenced goals. The control elements of RMPL are compiled into hybrid control automata (HCA), while the mode identification and reconfiguration is specified in terms of concurrent control automata (CCA).

Titan differs from Propel because it compiles procedural constructs into a declarative model, which is then interpreted by during execution. Titan is similar to IDEA this way, but differs from IDEA by using an explicit description of control behavior.

The core Titan system and the RMPL language have been extended to handle hybrid (continuous/discrete) state information, resulting in a system called Moriarty. A different extension, implemented in the Kirk execution system [30], supports distributed cooperative execution. Titan, Moriarty and Kirk may be described as *execution systems with internal planner*.

RPL

Reactive Plan Language (RPL) [19] was inspired by RAP and PRS. It is a Lisp-like language and includes rich set of control constructs, such as conditionals, looping, and the ability to specify "policies" that hold during the execution of particular sub-tasks.

RPL was designed to support replanning and debugging of task definitions [2]. Based on experience obtained during execution and Monte-Carlo simulations of task execution, situations can be identified where tasks are likely to fail. Heuristic "critics" are then used modify the task (e.g., adding new constraints, adding new policies) in order to fix the bugs found. RPL is an *execution system with internal planner*.

TDL

The Task Description Language (TDL) [25] uses a procedural representation to support plan execution. It is an extension of C++, adding syntax for specifying high-level control. A Java-based compiler translates TDL into pure C++, together with calls to a domain-independent task management library. The resulting code can then be compiled with any existing compiler and linked with existing C++ code. There are options in the language to specify that the resulting code should be threaded and/or distributed (the latter used for coordinating multiple robots.

TDL provides the ability to represent high-level control constructs including task decomposition, task coordination and synchronization, execution monitoring and exception handling, as well as distributed coordination between multiple agents. Being an extension of C++ makes it very easy to integrate TDL into projects – developers can use as much, or as little, of the TDL functionality as they need to augment the standard C++ functionality.

High level control constructs are represented in TDL as task-trees. Task trees represent the execution trace of hierarchical plans and are created dynamically at run time. The task-tree decomposition can be created from conditional and recursive task representations. The temporal constraints in the task-tree decomposition (partially) order task execution. Planning and sensing are treated as schedulable activities. In other words, the executive runs the main loop and calls the planner when required. TDL is an *execution system coupled with an external planner*. In several projects, a symbolic *Plan Representation Language (PRL)* was used to transfer data between a planner and a TDL-based executive [13] To date, TDL has been used in about a dozen mobile robot and autonomous system projects at various universities and institutions, including several NASA rovers [7].

Universal-Executive

The Universal-Executive is currently under development in a collaborative effort of researchers at NASA Ames Research Center, NASA's Jet Propulsion Laboratory and Carnegie-Mellon University. It is being designed to facilitate reuse and inter-operability of execution and planning frameworks. Plan execution systems often have a close relation to the planners that they are associated with, which makes information sharing between different execution and decision-making systems difficult.

The Universal-Executive builds on the Coupled Layer Architecture for Robotic Autonomy (CLARAty) [27], which is a two layer software architecture that was developed to enable both a plug-and-play capability and a tighter coupling of high level decision making planners and the interface to hardware. The CLARAty architecture has successfully enabled interoperability at the Functional Layer, which is the interface to the hardware. Current work, including the development of the Universal Executive, is addressing this same goal at the Decision Layer.

The execution language to be used in the Universal-Executive is called *Plan Execution Interchange Language (PLEXIL)*. PLEXIL extends many execution control capabilities of other systems. The key characteristics of PLEXIL are that it is compact, semantically clear, and deterministic given the same sequence of events. At the same time, the language is quite expressive and can represent simple branches, floating branches, loops, time and event driven activities, concurrent activities, sequences, and temporal constraints.

The input to the Universal-Executive will be a PLEXIL representation of an execution control instance and a description of relevant domain information. Execution nodes describe both initiation of real-world actions, and the control of execution. The nodes are arranged into hierarchical trees where leaf nodes are action nodes and internal nodes are control nodes. This is different from TDL, where task trees are a type rather than an instance.

The execution of each node is governed by a set of conditions, such as when the node gets activated and when it is done. The Universal-Executive will be capable of executing multiple nodes concurrently. When action nodes are executed, commands are sent to the rover, whereas

when internal nodes are executed, they are expanded to the next level of nodes in the tree.

The expressiveness of the language enables the Universal Executive to handle dynamic outcomes and environmental uncertainty. The executive can also provide execution information and outcomes back to higher-level systems. Consequently, it can be used both as a stand-alone *execution-only system*, and as an *execution system coupled with an external planner*.

Conclusions

The demands of future NASA spacecraft and robotic missions dictate an execution system that has great flexibility, expressiveness, and ease of use. This paper has presented a number of execution systems and execution languages that are relevant to NASA-type missions.

Acknowledgements: We thank Emmanuel Benazera, Howard Cannon, Mike Freed, Nicola Muscettola, Corina Pasareanu, and Rich Volpe for many useful discussions and all the attendees of the "Workshop on Existing Planning and Execution Systems", Nov 16, 2004 at NASA Ames Research Center, which motivated this paper. In addition, Mike Freed, Chris Grasso, and Nicola Muscettola provided invaluable comments on this paper.

References

1. Barrett A., Knight R., Morris R., Rasmussen R., *Mission Planning and Execution Within the Mission Data System*, International Workshop on Planning and Scheduling for Space (IWPSS 2004). Darmstadt, Germany, June 2004.
2. Beetz M. and McDermott D., *Declarative goals in reactive plans*, In James Hendler (ed.), *Proc. First Int. Conf. on AI Planning Systems*, San Mateo: Morgan Kaufmann, pp.~3--12
3. Bernard D. et al. *Final Report on the Remote Agent Experiment*, NMP DS-1 Technology Validation Symposium Feb 8th and 9th 2000, Pasadena, CA
4. Bresina J.L. and Washington, R., *Robustness via Run-time Adaptation of Contingent Plans*, In Proceedings of the AAAI-2001 Spring Syposium: Robust Autonomy. Stanford, CA
5. Dearden R., Meuleau N., Ramakrishnan S., Smith D., and Washington R., *Incremental Contingency Planning, ICAPS-03 Workshop on Planning under Uncertainty*, Trento, Italy, June 2003.
6. Drummond M., Bresina J., Kedar S., The Entropy Reduction Engine: Integrating Planning, Scheduling, and Control, in SIGART Bulletin 2, 1991, 48-52
7. Estlin T., Gaines D., Chouinard C., Fisher F., Castano R., Judd M., Anderson R., and Nesnas I.. "Enabling Autonomous Rover Science Through Dynamic Planning and Scheduling," *Proceedings of 2005 IEEE Aerospace Conference*, Big Sky, Montana, March, 2005.
8. Firby J, *Adaptive Execution in Complex Dynamic Domains***,** Ph.D. Thesis, Yale University Technical Report YALEU/CSD/RR #672 *January 1989*
9. Frank J., and Jonsson A. K., *Constraint-based Attribute and Interval Planning*, in Constraints, 8(4), p 339-364, 2003.
10. Freed M., *Managing Multiple Tasks in Complex, Dynamic Environments*. In Proceedings of the 1998 National Conference on Artificial Intelligence. Madison, WI. 1998
11. Gat E.. *ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents*, Proc. AAAI Fall Symposium on Plan Execution, Boston MA, October 1996.
12. Georgeff M. and Lansky A., *Procedural Knowledge*, in Proceedings of the IEEE Special Issue on Knowledge Representation, Volume 74, pages 1383-1398, 1986.
13. Goldberg D., Cicirello V., Dias M. B., Simmons R., Smith S., and Stentz A., *Market-Based Multi-Robot Planning in a Distributed Layered Architecture*, In Proceedings of the Multi-Robot Systems Workshop, Washington, D.C., March 17-19, 2003
14. Grasso C., *The Fully Programmable Spacecraft: Procedural Sequencing for JPL Deep Space Missions Using VML (Virtual Machine Language)*, IEEE Aerospace Applications Conference Proceedings, march 2002
15. Ingrand F., R. Chatila, R. Alami and F. Robert, *PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots*, IEEE ICRA 96, Minneapolis, USA.
16. Kim P., Williams B., Abramson M., 2001. *Executing Reactive, Model-based Programs through Graph-based Temporal Planning*. IJCAI '01. AAAI, Menlo Park, CA.
17. Levinson R., *A General Programming Language for Unified Planning and Control.* Artificial Intelligence, special issue on Planning and Scheduling, Vol. 76. Elsevier Press. July 1995.
18. Levinson R., Unified Planning and Execution for Autonomous Software Repair. ICAPS 2005. Workshop on Plan Execution: A Reality Check. 2005.
19. McDermott D., *A Reactive Plan Language*, Research Report YALEU/DCS/RR864 Yale University 1991
20. Muscettola N., Dorais G., Fry C., Levinson R., and Plaunt C., "IDEA: Planning at the Core of Autonomous Agents," (AAAI 2001)
21. Pedersen L., Smith D., Deans M., Sargent R., Kunz C., Lees D. and Rajagopalan S., *Mission planning and target tracking for autonomous instrument placement*, 2005 IEEE Aerospace Conference.
22. Pell B., Bernard D.E., Chien S. A., Gat E., Muscettola N., Nayak P. P., Wagner M. D., and Williams B. C.. *A Remote Agent Prototype for Spacecraft Autonomy.* In Proceedings of the

SPIE Conference on Optical Science, Engineering, and Instrumentation, 1996.
23. Pell B., Gamble E., Gat E., Keesing R., Kurien J., Millar B., Nayak P. P., Plaunt C., and Williams B., *A Hybrid Procedural/Deductive Executive For Autonomous Spacecraft.* In Proceedings of the Second International Conference on Autonomous Agents, Minneapolis, MI 1998
24. Rasmussen R., *Goal-Based Tolerance for Space Systems Using the Mission Data System*, In Proceedings of the 2001 IEEE Aerospace Conference.
25. Simmons R. and Apfelbaum D.. A Task Description Language for Robot Control, Proceedings of Conference on Intelligent Robotics and Systems, Vancouver Canada, October 1998.
26. Simon D., Espiau B., Kapellos K., Pissard-Gibollet R. , *Orccad: Software Engineering for Real-time Robotics, A Technical Insight*, Robotica, Special issues on Languages and Software in Robotics, vol 15, no 1, pp 111-116
27. Volpe R., Nesnas I. A. D., Estlin T., Mutz D., Petras R., Das H., *The CLARAty Architecture for Robotic Autonomy.* Proceedings of the 2001 IEEE Aerospace Conference, Big Sky Montana, March 10-17 2001.
28. Williams B. C., Ingham M., Chung S. H., and Elliott P. H., January 2003. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers, invited paper in Proceedings of the IEEE: Special Issue on Modeling and Design of Embedded Software, vol. 9, no. 1, pp. 212-237.
29. Williams B. C., Ingham M., Chung S., Elliott P., and Hofbaur M., *Model-based Programming of Fault-Aware Systems*, AI Magazine, vol. 24, no. 4, Winter 2004, pp. 61-75.
30. Kim P., Williams B. C. and Abramson M, 2001, *Executing Reactive, Model-based Programs through Graph-based Temporal Planning*, Proceedings of the International Joint Conference on Artificial Intelligence, Seattle, Wa.