Runtime Verification for Autonomous Spacecraft Software

Allen Goldberg
Kestrel Technology
NASA Ames Research Center MS 269-1 T35B
Moffett Field, CA 94035
650-604-4858
goldberg@email.arc.nasa.gov

Klaus Havelund
Kestrel Technology / NASA Ames Research Center
havelund@email.arc.nasa.gov

Conor McGann
QSS Group, Inc. / NASA Ames Research Center
cmcgann@email.arc.nasa.gov

Abstract -

Autonomous systems are systems that can operate without human interference for extended periods of time in changing environments, likely in remote locations. Software is usually an essential part of such systems. However, adaptation of autonomy software is limited by its complexity and the difficulty of verifying and validating it. We describe an approach named runtime verification for testing autonomy software. Runtime verification is a technique for generating test oracles from abstract specifications of expected behavior. We describe its application to the PLASMA planning system, used in the recent Mars Exploration Rover missions. We furthermore discuss alternative autonomy V&V approaches.

TABLE OF CONTENTS

1 INTRODUCTION....1
1 INTRODUCTION....1
2 V&V OF AUTONOMY SOFTWARE....2
3 RUNTIME VERIFICATION....3
4 PLASMA....5
5 RUNTIME VERIFICATION OF PLASMA....6
6 OTHER APPLICATIONS OF RV....7
7 RELATED WORK....7
8 CONCLUSIONS....8
9 REFERENCES....8

1 INTRODUCTION

The difficulty of verifying and validating (V&V) autonomy software has limited its use on spacecraft. In this paper, we overview new approaches to autonomy V&V and describe an experiment with one of the approaches. Our experiment involved regression testing for PLASMA, the next generation of planning technology used to create MAPGEN [4], which in turn was used to plan and schedule MER Rover activities. Planners take as input a set of high level goals to be achieved, such as driving a Rover to a distant location, or taking a picture or measurement, and generate a series of low-level commands that realize the goals, while respecting the flight resource and safety constraints such as not taking pictures while moving, or staying within power budgets. PLASMA is a model-based planner generation system. As shown in Figure 1, PLASMA takes as input a domain model and "compiles" the domain model into a planner specialized to that model. The resulting planner solves planning problems, i.e. it takes as input a set of goals and uses heuristic search to find efficient plans. A model is a declarative description of a domain that defines domain objects, such as cameras and wheels, constraints, and actions such as turning on a camera. Because complex planning problems, in general, are *NP-complete*, heuristic search is required, since finding an optimal solution would require an exponential time algorithm that will not scale to the size of problems regularly solved by these systems.

0-7803-8870-4/05/$20.00© 2005 IEEE
IEEEAC paper #1215

When constructing a planner in this way, the key activities are building the model and ensuring that the heuristic search used by the planner is effective in finding good plans for the goal sets of interest. A common development methodology is to incrementally build, refine, and elaborate the domain model, testing the model against a graded set of challenge examples. However, small changes to the model or the heuristics can have unexpected and dramatic changes in the planner's behavior. Test cases that prior to the modification were solved by the planner may no longer be solvable, or may be solved by a different plan that may or may not be acceptable. Thus, comprehensive regression testing is integral to this development methodology. However an oracle used with a regression test suite cannot simply check that the identical plans are obtained, but rather must determine that an acceptable plan is generated within an acceptable time bound. To automatically execute a regression suite, the acceptance criteria for each test case must be realized as executable code. To do so we employed runtime verification, a V&V technology that allows the creation of test oracles that can check sophisticated properties of a computation. We carried out this approach by formulating, using a runtime verification system called Eagle, both universal properties (true for any computation of the planner) and problem-specific properties. We then analyzed logs generated by the planner to check conformance to these properties.

The rest of the paper is organized as follows. Section 2 describes the general problem of V&V of autonomy software. Section 3 describes Runtime Verification, a specific V&V technique that combines formal methods tools with program testing. Section 4 describes PLASMA and section 5 describes how it was tested using Eagle. Section 6 describes other applications of runtime verification to autonomy testing. In section 7 we describe related work. Finally, section 8 states conclusions and describes future work.

2 V&V OF AUTONOMY SOFTWARE

Autonomy software can both dramatically improve the capability and robustness of spacecraft while reducing the cost of operation. In this paper we consider model-based autonomy software, particularly planning and scheduling systems. This software can expand the capabilities of unmanned missions, allow for greater utilization of spacecraft resources, and significantly reduce the level of ground support. The capabilities of autonomy software are now being successfully demonstrated on a number of deployed missions.

For example MAPGEN [4] is a ground-based tactical activity planning system used each day for planning the command sequences uplinked to the Mars MER Rovers, that uses EUROPA [9], a precursor of PLASMA. Upwards of 700 activities each day are planned by MAPGEN.

The EO-1 Autonomous Science Agent [5] is autonomy software currently flying onboard the Earth Observing One (EO-1) spacecraft. This software has been flying in a series of tests since fall 2003, is scheduled to fly well into 2005 and may fly well into 2006. The highest level of EO-1 autonomy software is the CASPER planning system [6]. The CASPER planning system has been used in a wide range of applications including but not limited to: spacecraft control, deep space communications station control, rover control, and as a single agent controller in multi-agent testbeds. On the EO-1 the planner schedules observations, communication activities, etc.

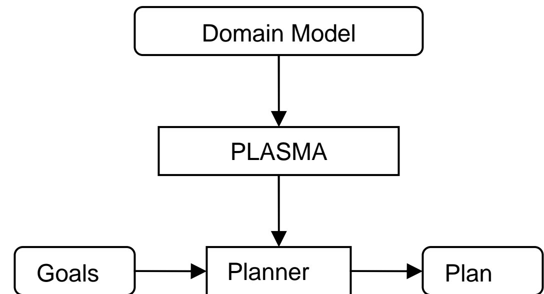


Figure 1: Planning with PLASMA

However, the difficulty of V&V has excluded use of autonomy software on manned aircraft and restricts its use on spacecraft. There are aspects of autonomy software that make verification and validation both different and difficult. Theoretically these problems are *NP-complete*. *NP-complete* problems require, in the worst case, exponential time to solve. Hence, achieving an exact solution within an acceptable response time over all possible inputs is currently impossible. Thus the software designer faces a complicated design trade space of performance, accuracy, and constraints on the input problems to be solved. This in turn implies that there is no simple and "clean" characterization of the elaborated software requirements. Previous work has demonstrated that this design space can be successfully negotiated, since systems capable of generating very good results, often exceeding what can be done by humans, have been built. But validating and verifying these systems raises challenges. Are the approximate solutions produced by the system adequate? Are the real time performance goals met? Another characteristic of *NP-complete* problems is that of discontinuity - a small change to an input can lead to significant changes to the accuracy and performance of the algorithm. This discontinuous behavior makes V&V more difficult, because the central notion of testing is that by

testing “representative” inputs one can inductively infer that similar inputs will also behave correctly.

The planning and scheduling software applications we consider consist of a declarative model that defines constraints, hierarchical goal structures, domain entities, etc. and an “engine” that interprets problem solving tasks using this model. This different structure calls into question the direct applicability of traditional white box structural testing methods and coverage criteria. The model-based approach furthermore leads to a development methodology in which the model is incrementally developed and validated, which in turn requires a V&V methodology that can support such frequent modifications. Empirical observation suggests that it is more difficult for a human to understand the behavior of model-based systems because of the seemingly exponential number of possible interactions of model elements. Because the engine is fairly stable (except for changes to heuristics) and shared between different applications, validating the model is the primary V&V focus.

Finally, almost by definition, autonomy software is required to react to a diverse set of conditions. A test approach based on testing a nominal scenario and off-nominal variants cannot be applied. A larger operational profile must be defined and tested against. Thus one can expect that for autonomy software the test set is very large, and that automated testing procedures will be very important.

In summary, autonomy is a high-stakes technology often used at the highest level of commanding systems with high cost and human risk. The V&V task is more difficult for autonomy software because it

- has an input-output behavior that is difficult to state and verify,
- responds to a wide variety of inputs and not just a single “nominal” scenario, and
- has a declarative model-based architecture that makes it difficult to predict “execution paths”.

In the following we shall investigate a particular V&V technology, runtime verification, which can be used to increase the reliability of model-based autonomy systems.

3 RUNTIME VERIFICATION

Runtime Verification

Formal methods hold out the promise for higher-quality software verification by judicious application of mathematical methods. Unfortunately formal methods have not scaled to be routinely applied to production software code. The challenges of autonomy software, described above, exacerbate this problem making it unlikely that formal methods can be usefully applied to autonomy software in the near to mid term. However, emerging from formal methods research are hybrid V&V methods that apply the technological advances of formal methods to more traditional V&V methods. Runtime verification is such a hybrid approach.

In the rigorous approach of formal methods, precisely-specified software properties are stated and mathematically verified. The precision is obtained by stating properties in a formal logic. A particularly relevant one for expressing properties of software is *temporal logic*. Temporal logic is a logic that enables succinct description of systems that evolve over time, for example, the discrete computational steps of a computer program. Temporal logic is appropriate for programs that are reactive, i.e. those that execute continuously by reacting to an environment, and do not simply take an input at the start of the computation and produce a single output at termination. Using temporal logic properties like “if the reset signal is received then within 1 second the device is reset” are naturally stated. Furthermore, V&V can be more effective if not just the inputs and outputs of a program are examined but also the internals of the computation. There again temporal logic is relevant.

Runtime verification applies temporal logic to program testing. The software is tested for conformance to precisely stated properties specified in the logic. The properties may be universal properties that are expected true of all program inputs, as with traditional formal verification, or can be properties specific to a particular input. Runtime verification acknowledges that proving non-trivial functional and performance properties for all inputs for complex software is beyond current capabilities, but that the specification of those properties using a formal language such as temporal logic is a great advantage in a testing context.

Runtime verification is used as part of writing/generating test cases. A test case consists of a test input and an oracle. The oracle is a predicate that determines if the behavior of the program is the desired or correct behavior with respect to the input. In an automated test suite the oracle must be represented by code that performs the checking. If the correct behavior of the program is described by a set of temporal logic formulas then the oracle is a program for checking those formulas. The value added by runtime verification is the creation of tools to automatically generate such oracles. The key technology is a compiler or interpreter that translates temporal logic formulas into code that checks if a program execution conforms to the property.

A runtime verification system can be architected so that the checking is done independently of the system under test. In this case the system under test is instrumented so that a sequential log is generated. The log can be saved to a file for offline analysis or examined in real time. Alternatively the checking can be done as code that is integrated into the

system under test. There are advantages to both approaches. The main advantage of the independent approach is that the impact on the real time performance of the system under test is minimized to creation of the log. The main advantage of the integrated approach is that the checking can become part of the system under test thus supporting an autonomic computing or Integrated Vehicle Health Management (IVHM) capability.

Figure 2 illustrates the offline architecture we used in this runtime verification application.

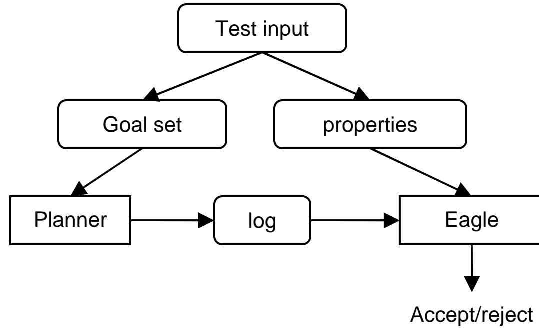


Figure 2: Offline Test Execution Architecture

Eagle

In this section we describe Eagle [2,3,1], the temporal logic framework for runtime verification used in this work. Most temporal logics were developed for use with model checkers, and so had to trade expressiveness for efficient model checking. In designing Eagle for runtime verification we were able to include more features that improve expressiveness because the computational efficiency is a lesser, although still important, issue.

Eagle is a rule-based framework for defining temporal logics. It allows the definition of new temporal operators using a parameterized rule construct. With Eagle, in a few lines of text, we are able to define past, future and interval time temporal logics, real-time temporal logics, regular expression logics, state machine notations, and more. Unlike more common "propositional" temporal logics, the Eagle user can define rules parameterized by data values (e.g. the value of a program variable from the system under test).

The models of the logic are execution traces, where a trace is a sequence of states, each state associating values with a collection of variables. An execution trace can, for example, represent the log file generated by executing the system under test. Assume as an example the following trace consisting of four states, each being a pair giving value to two variables *x* and *y*:

$$(x=0,y=0) \quad (x=1,y=0) \quad (x=0,y=1) \quad (x=1,y=1)$$

In the first state both *x* and *y* are *0*. In the next state *x* is *1* while *y* is still *0*, etc. Our logic allows us to state properties about such a trace, properties that can be checked. Assume for example that we want to state and check the (true) property *"P1: it is always the case that when x is positive then eventually y becomes positive"*. This sentence embeds two temporal operators "always" and "eventually", which we will have to define. Note, once defined they can be reused. The definition in Eagle of these two operators is as follows:

```
max Always(Form F) = F ∧ @ Always(F)

min Even(Form F) = F ∨ @ Even(F)
```

The first definition introduces the temporal operator `Always`, which as argument takes a formula `F`, and is defined as: `F` holds now, and in the next state `Always(F)` holds recursively. The operator `@` points to the next state in the trace, relative to a current position. This definition corresponds to standard equations in classical text books. Similarly, the second definition defines the `Even` (eventually) operator: `F` holds now, or `Even(F)` holds in the next state.

The keywords `max` and `min` (referring to the mathematical semantics as a maximal versus minimal fix-point interpretation) define the value of the respective formulas `Always(F)` and `Even(F)` at the end of the trace, when evaluation terminates: a maximal rule evaluates to true while a minimal evaluates to false. This carries the natural intuition that `Always(F)` is true at the end if it has been true all along, while `Even(F)` is false at the end, since apparently an obligation `F` did not occur that should have occurred.

The monitor for property P1 can now be stated as follows:

```
mon P1 = Always(x>0 → Even(y>0))
```

The defined monitor is composed of temporal operators and of classical propositional logic operators, such as here implication (→). The logic also allows for the definition of past time operators, using the previous-state operator `#`. For example, dual past time versions of the `Always` and `Even` operators, which we name `Sofar` and `Prev`, are defined as follows:

```
max Sofar(Form F) = F ∧ # Sofar(F)

min Prev(Form F) = F ∨ # Prev(F)
```

We have now seen examples of rules being parameterized with formulas, useful for defining new temporal operators. However, rules can also be parameterized with data such as integers, floats, strings, etc. Consider for example that we

want to state and check the (false) property: "*P2: whenever x is positive with some value k, then sometime in the past y had that value, and sometime in the future x is less than k*". This property can be stated as follows, using an extra rule `R` to capture the value *k*:

```
min R(int k) = Prev(y=k) ∧ Even(x<k)

mon M2 = Always(x>0 → R(x))
```

The rule `R` is here introduced to capture the value of `x` at the moment where `x>0`, binding it to the formal parameter `k`.

Eagle rules are converted into oracles that check the trace, state by state, without storing the entire trace. This means that very large traces can be examined on-the-fly while the system under test is executing. This works by for each monitor to maintain a "current formula" that represents the value of the original formula on the so-far processed prefix of the trace. Consider for example the monitor for *P1* above. After having processed the first state *(x=0,y=0)*, the current formula is unchanged since no triggering positive *x* was detected. After processing the second state *(x=1,y=0)* however, the formula now changes to:

```
Always(x>0 → Even(y>0)) ∧ Even(y>0)
```

The crucial change is the addition of the conjunct `Even(y>0)`, which now is an obligation to be fulfilled. It will be discharged at the third state *(x=0,y=1)*, where after the current formula will reduce to its original form again. The fourth state *(x=1,y=1)* re-generates the `Even(y>0)` obligation since *x=1*, but it is immediately discharged since at the same time *y=1>0*.

4 PLASMA

In this section we describe PLASMA, a C++ library for building planning systems, that was tested using Eagle.

MAPGEN is a ground-based tactical activity planning system used on the ground each day for planning the command sequences uplinked to the Mars Rovers. It is based on EUROPA [9] which provided the core constraint-based planning technology to the MAPGEN team. PLASMA (**PLA**n **S**tate **M**anagement **A**rchitecture) is a successor to EUROPA, developed to provide increased performance, usability, and flexibility to this proven planning paradigm. PLASMA is intended for use in off-board and on-board planning and plan execution applications arising in the domain of space exploration. In addition to support of the development activities of PLASMA itself, it is an important goal to provide comprehensive verification and validation support for engineering mission applications with PLASMA. In this section we briefly present the underlying semantics of plans and planning in PLASMA, and describe the components of a typical application using PLASMA.

Plans and Planning in PLASMA

Consider the problem of controlling an imaging system on a satellite. In this example, we are concerned with attitude adjustment to position a camera, and control of the camera to take pictures. One could imagine a planning system which received input of initial attitude (i.e. initial state) and a set of imaging requests (i.e. goals). The solution is a sequence of states and actions reflecting the camera and satellite attitude control over time to accomplish the given requests (i.e. a plan). For simplicity, an imaging request is given by a position, and attitude is controlled to point to a position. Imaging requests are serviced by a *Camera*, and attitude control is accomplished by an *Attitude Controller*. These are two objects of the system whose behavior varies over time. The notion of predicates applying over intervals of time is fundamental to the description of object behavior in the PLASMA planning paradigm. The interval of time over which a predicate occurs is described using temporal variables i.e. *start, end* and *duration*. Use of variables provides flexibility in when a predicate might start or end. A relationship exists between these variables, namely, *start + duration = end*. This relationship is enforced via a *constraint* such that the bounds of the variables are automatically adjusted to eliminate illegal values from the domain of each variable. The actions and states of this domain are given by the following predicates which hold true over said intervals of time:

- Camera::off() – this predicate is true when the Camera is off.
- Camera::ready() – this predicate is true when the Camera is ready to take a picture.
- Camera::takePic(Position p) – this predicate is true when an action to take a picture at position p is taking place. The parameter p is an additional variable added to the built in temporal variables of this predicate.
- Attitude::pointAt(Position p) – this predicate is true when the Attitude Controller is pointing at position p. The parameter p is an additional variable added to the built in temporal variables of this predicate.
- Attitude::turn(Position from, Position to) – this predicate is true when an action to turn the satellite from one attitude to another is occurring. Parameters from and to are additional variables of this predicate.

Constraints must be specified in the model. For example there is the obvious constraint that a camera cannot be off while it is taking a picture. A number of additional model rules are indicated in Figure 3a. For example, the camera must be ready before it is used to take a picture. Such a rule

is naturally enforced as a constraint equating the end time of the ready predicate and the start time of the takePic predicate. Furthermore, we see that relations exist between parameter variables of predicate instances. For example, a turn action must meet a pointing state such that the target position of the turn is equal to the position to which we are pointing.

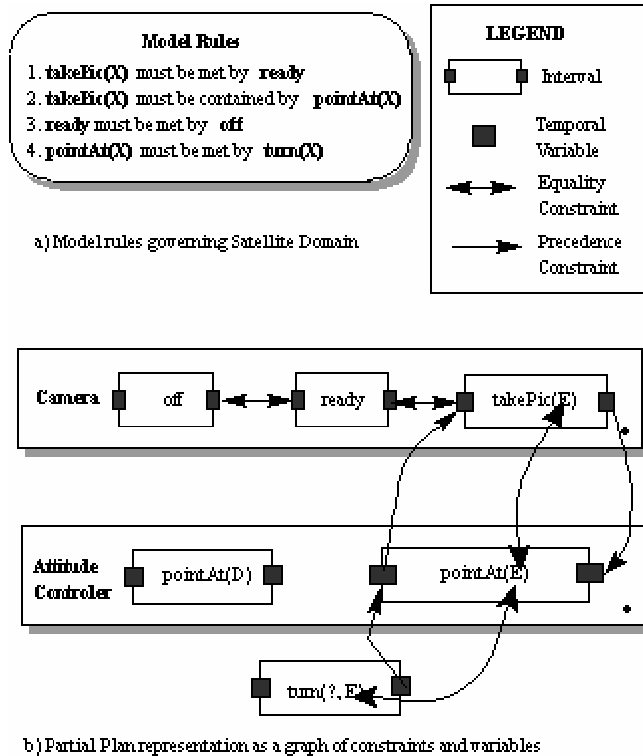


Figure 3: Plasma Planning

In PLASMA, planning is a process of elaboration of an initial partial plan into a final complete plan. Initial problem descriptions are specified in terms of predicate instances. For example, we might initially be in a state at the beginning of the mission where the *Camera* is off, the *Attitude Controller* is pointing at position *D*, and there is a goal to take a picture at position *E*. Model rules are used to express the causal relationships between actions and states in the problem domain. PLASMA uses these rules to automatically infer additional actions and states which must be introduced into the plan and to further introduce constraints among predicate variables to enforce model relationships. Figure 3(b) illustrates an interim stage of the planning process where some causal rules have introduced additional predicate instances (e.g. pointAt(E) and turn(?, E)) and constraints. A constraint reasoning system is used to propagate the relationships among variables in the plan, removing illegal values proactively to avoid unnecessary search. The planner is used to select values for variables and to place predicate instances in sequence on objects. If all values of a variable have been removed, the system is inconsistent and the planner must backtrack and try an alternative solution path. When all relevant variables have values, and all required predicate instances have been placed in the plan the planner will terminate.

PLASMA Architecture

PLASMA is a library for building constraint-based planning applications. The key provisions of the PLASMA architecture illustrated in Figure 4 are:

- A modeling language for describing the artifacts of the problem domain (i.e. objects and predicates) and the rules governing their interactions. Model development is a critical engineering task in deploying or developing an application in PLASMA.
- A component for representing plan state. The primary component for doing this is referred to as a *plan database*. The plan database accepts transactions to 1) create or delete objects; 2) create or delete predicate instances; 3) create or delete constraints; 4) restrict or relax variables. The plan database uses a *schema*, populated from the model, to enforce type restrictions.
- Inference and consistency management services. Plan state management is coordinated by the plan database but aided by both the *rules engine* and the *constraint engine*. The former uses rules from the model to create or delete predicate instances and constraints based on changes in plan state. The latter co-ordinates an extendible range of specialized algorithms (e.g. resource envelope calculations, temporal network propagation) for propagating relationships among variables to prune illegal values and detect inconsistencies.
- Search services. The problem of searching is essentially one of deciding which of the available open decisions should be made (or retracted) next, and which choice to make for that decision. Different applications will employ different search strategies. PLASMA provides a baseline planner using chronological backtracking. It further provides facilities to track open decisions and apply heuristics for decision ordering. Heuristics can be scripted as part of application development and/or deployment.
- Event publication for external integration. PLASMA exposes a rich set of events for all components of the system. Listeners can be registered for these events for a variety of reasons. Most relevant in the context of this work is the use of a listener to log events to an *event log* which can be monitored by a runtime verification system.

5 RUNTIME VERIFICATION OF PLASMA

In this section we describe the application of runtime verification to regression testing.

When using model-based autonomy software for applications such as planning and scheduling the major development focus is construction of a model representing the application domain. The model must be accurate and effective, meaning that the model must reflect the physical reality of the objects, and the model when linked to the engine(s) and heuristics must be capable of generating effective plans for the problem set of interest. Construction of such a model is often done as an evolutionary process. First a simple model is constructed and tried on simple examples. The model is then modified or enhanced. Sometimes the search strategy or heuristics are modified. Whenever any such change is made there is a significant likelihood that the prior test cases will not execute exactly as they did previously. While this situation is not essentially different then any other regression test situation in which retesting is required when the code is changed, there are some significant new and different factors in the autonomy case. First, regression testing is required to support the central development activity – namely model construction. Second, unlike traditional software, there is not a single correct output with an easily specifiable oracle. Changes in the model may yield plans that may differ dramatically but still be acceptable. (Recall the "discontinuity" of autonomy problems.) In other words, the acceptability criteria to be represented in an oracle may be quite complex. Recalling the first difference, that regression testing is part of a development activity, it is appropriate and in fact crucial, to frame the problem as building an oracle that determines if *the computation* and not just the output of the computation is acceptable. For example, a developer may wish to assert that the planner should never need to backtrack on the problem, or that the plan database have certain properties at various points through the planning process. For these reasons, the power of a runtime verification system such as Eagle can be usefully leveraged.

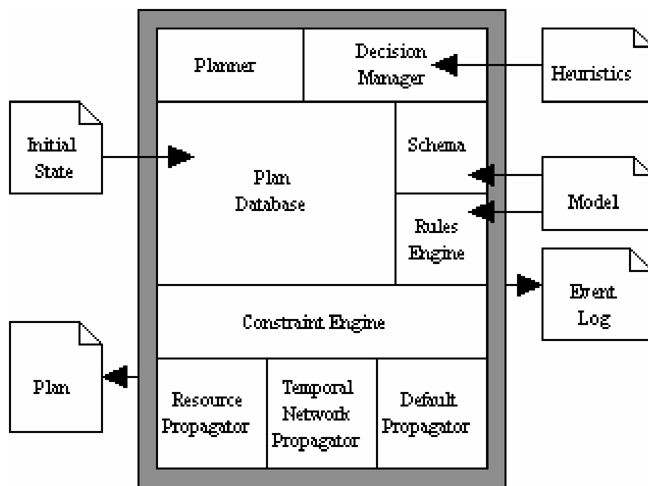


Figure 4: PLASMA Architecture

As described above, offline runtime verification processes a log containing the pertinent data for testing the asserted properties. In this application, since PLASMA already generates sufficiently detailed trace logs usable by Eagle code instrumentation is not necessary. Instead the logs are simply parsed and used to update the Eagle state. To date we have prototyped this approach on a number of simple examples and found it effective.

6 OTHER APPLICATIONS OF RUNTIME VERIFICATION

Autonomy software differs in significant ways from more traditional software and these differences have impact on V&V. We describe some additional applications of runtime verification to autonomy V&V that we have developed or plan to do.

Planners produce a plan, and, of course, at some later stage the plan is executed by an execution executive. Verification of the execution executive requires checking that plans are executed according to their intended meaning. This too can be checked using runtime verification, by translating the plan semantics into Eagle properties, logging the behavior of the execution engine and monitoring that against the properties. This was done as part of earlier work using a test-case generation and runtime verification system called X9 [1], which not only monitored plan executions but combined that with a test case generator to generate test plans.

In rich and complicated environments such as those encountered by planetary rovers, plans often fail due to differences between the actual encountered environment and the assumed environment. This often leads to a re-planning activity, and a tight integration of the planner and the execution engine. Furthermore, these consideration leads to an architecture in which planning occurs at multiple levels with different time horizons. A planner may be used to globally plan the day's activity for a rover and a second, reactive planner may be used to plan on the minute-or-less time scale, taking into account execution failures. Should the goal of the long time horizon planner fail to be achieved, then re-planning at the higher level is initiated. Our future work includes applying runtime verification to verify this architecture. The approach is to translate the high level planning model into Eagle and then monitor a trace of the low-level execution against the model. This does not validate the model but rather checks the consistency of the multiple planners and execution executive against the model semantics.

7 RELATED WORK

In essence, the focus of the work presented in this paper is model-based planning systems, and specifically how runtime verification can be used to observe the process of a

planner interpreting a model, with the purpose of analyzing the quality of the model. There has been relatively little work on V&V of autonomy software, and nothing to our knowledge on such regression testing for autonomy model validation.

A model can alternatively be analyzed in isolation, without invoking a planner, for various properties, such as completeness and soundness. In [13] and [11] are described two attempts to use model checking to analyze models. Model checking is a technique for exploring all possible paths through the model, using various techniques for avoiding re-exploration of paths. The earliest work described in [13] applies discrete state (without real-time constraints) model checkers to the analysis of planning models. A planning model is formulated in the input notation of the model checker. Various goals can then be stated and it can be determined for example whether a plan exists for achieving a specific goal, whether there are initial states where a plan cannot be generated for a specific goal, etc. The approach described in [11] goes beyond this work by using a real-time model checker, allowing to represent more realistic models with time constraints. These attempts demonstrate that inconsistencies and missing constraints can be detected. However, the experiments with especially real-time model checking show scalability issues for larger models.

Model checking has also been used to analyze executives, systems that execute plans. In [10] is described work applying model checking to analyze part of the executive of the Remote Agent planning and execution system that for a small period controlled the Deep-Space 1 (DS-1) space craft during flight in 1998. Although labor intensive, the analysis demonstrated, before flight, serious concurrency problems, problems that actually caused a deadlock during flight, resulting in a system shut-down for several hours. The errors found before flight were corrected but the deadlock was re-introduced later.

In [7] Chien et.al. describe their efforts to validate autonomy software on the EO-1 mission. This includes walkthroughs for model validation, and test case generation. Safety constraints were checkable against a lower level safety monitor.

Feather and Smith [8] note that it is easier to check that a plan is correct with respect to a model than it is to produce the plan, and have applied this idea to the DS-1 Remote Agent planner. They load a plan resulting from execution of the planner into a database and then check the database against constraints generated from the model. This is intended to verify the planner but not to validate the model.

The Livingstone PathFinder (LPF) [12] is a system for testing Livingstone models. Livingstone is a model-based diagnosis system. LPF consists of a test driver that generates a sequence consisting of either commands or injected faults, a simulator of the modeled device and the Livingstone Engine. The system checks whether the diagnosis system can detect the faults injected into the input stream.

8 CONCLUSIONS

The difficulty of verification has slowed the incorporation of autonomy software in flight systems in space and on aircraft. However due to the advanced functionality and significant cost savings autonomy provides, there is great incentive to utilize autonomy. In this paper we demonstrated new V&V capabilities that exploit the structure and needs of autonomy software. In particular we addressed the problem of V&V of model-based software by prototyping a new capability that allows a developer to incrementally evolve a domain model and an automated regression suite. Constructing a domain model that is an accurately models the domain and that is effective in solving planning problems is the key development activity. Since there are many acceptable solutions to a planning problem, defining an oracle for a test is non-trivial. We showed that by use of runtime verification called Eagle oracle can be constructed that are based not just on the resulting plan but on the computation of the planner to arrive at the plan, and that rich temporal properties of the computation can be easily stated..

REFERENCES

[1] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser and Rich Washington, “Combining Test Case Generation and Runtime Verification”, submitted to the journal Theoretical Computer Science.

[2] Howard Barringer, Allen Goldberg, Klaus Havelund and Koushik Sen, “Rule-Based Runtime Verification”, 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI’04), LNCS 2937, Springer, pages 44-57, Editors B. Steffen and G. Levi, Venice, Italy, January 2004.

[3] Howard Barringer, Allen Goldberg, Klaus Havelund and Koushik Sen, “Program Monitoring with LTL in Eagle”, Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD’04), IEEE Digital Library, Santa Fe, New Mexico, USA, April 2004.

[4] John Bresina, Ari Jonsson, Paul Morris and Kanna Rajan, "Constraint Maintenance with Preferences and Underlying Flexible Solution", Online-2003, Online Constraint Solving: Handling Change and Uncertainty, A CP2003 workshop, Kinsale, Co. Cork, Ireland, September 29th , 2003.

[5] Steve Chien, et.al. "Autonomous Science on the EO-1 Mission", Proceedings of the International Symposium on Artificial Intelligence Robotics and Automation in Space (i-SAIRAS), Nara, Japan, May, 2003.

[6] Steve Chien, Russell Knight, Andre Stechert, Rob Sherwood and Gregg Rabideau, "Using Iterative Repair to Increase the Responsiveness of Planning and Scheduling for Autonomous Spacecraft", IJCAI99 Workshop on Scheduling and Planning meet Real-time Monitoring in a Dynamic and Uncertain World, Stockholm, Sweden, August 1999.

[7] Benjamin Cichy, Steve Chien, Steve Schaffer, Daniel Tran, Gegg Rabideau and Rob Sherwood, "Validating the Autonomous EO-1 Science Agent", International Workshop on Planning and Scheduling for Space (IWPSS 2004). Darmstadt, Germany, June 2004.

[8] Martin Feather and Ben Smith, "Automatic Generation of Test 0rac1es - From Pilot Studies to Application", Automated Software Engineering, 8(1):31-61, January 2001.

[9] Jeremy Frank and Ari Jonsson, "Constraint-Based Interval and Attribute Planning", Journal of Constraints Special Issue on Constraints and Planning. 2003.

[10] Klaus Havelund, Mike Lowry and John Penix, "Formal Analysis of a Space Craft Controller using SPIN", IEEE Transactions on Software Engineering, Volume 27, Number 8, August 2001.

[11] Lina Khatib, Nicola Muscettola and Klaus Havelund, "Mapping Temporal Planning Constraints into Timed Automata", TIME'01 (IEEE Press), The 8th International Symposium on Temporal Representation and Reasoning, Cividale Del Friuli, Italy, 2001.

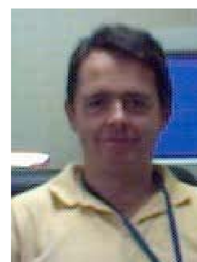
[12] Tony Lindsey and Charles Pecheur, "Simulation-Based Verification of Autonomous Controllers with Livingstone PathFinder", Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis Of Systems (TACAS'04), Barcelona, Spain, March-April 2004, Lecture Notes in Computer Science, vol. 2988, Springer Verlag.

[13] John Penix, Charles Pecheur and Klaus Havelund, "Using Model Checking to Validate AI Planner Domain Models", 23rd Annual Software Engineering Workshop, 1998.

BIOGRAPHY

Klaus Havelund, Kestrel Technology at NASA Ames Research Center, received his Ph.D. in Computer Science from Copenhagen University, Denmark, in 1994 (carried out at Ecole Normale Superieure in Paris, France). Havelund conducts research in program verification and testing techniques, including runtime verification. He initiated and has organized/steered a series of international workshops on runtime verification (RV'01-04). Previous work has included such subjects as specification language design, concurrent language theory, theorem proving, model checking, and dynamic program analysis. He co-developed the formal specification language RSL, and wrote the majority of the textbook "The RAISE Specification Language", published in the BCS Practitioners Series, Prentice Hall 1992. He conceptualized and constructed the first prototype of the Java PathFinder model checker for Java, a second version of which won the NASA TGIR award in 2003.

Allen Goldberg, works for Kestrel Technology at NASA Ames Research Center. He received a Ph.D. in Computer Science from Courant Institute, at New York University. His interests include runtime verification, program testing, and program transformation.



Dr. Conor McGann is a computer scientist in the planning and scheduling group at NASA Ames Research Center. Dr. McGann's research in Constraint-based Planning has contributed essential planning and scheduling capabilities to a wide variety of research and mission applications. He is the architect of PLASMA, a constraint-based planning system, and has been a member of the MAPGEN development team for the Mars Exploration Rover (MER) mission, which received a "Turning Goals Into Reality" NASA Administrator's Award in 2004. Dr. McGann received his doctorate in Computer Science (1995) in the area of model-based reasoning and his undergraduate degree in Computer Engineering (1990) from Trinity

College, Dublin. His principal research interest is in the confluence of artificial intelligence, database and distributed systems technologies to build robust intelligent systems.