Slightly revised version of paper that appeared in:
Foundations of Intelligent Systems,
(Eds. Z.W. Ras, A. Skowron),
TenthInternational Symposium on
Methodologies for Intelligent Systems,
Charlotte,North Carolina,
October 15-18, 1997,
Lecture Notes in ArtificialIntelligence,
Springer-Verlag, Vol. 1325.

Verification and Validation of AI Systems that Control Deep-Space Spacecraft

Michael Lowry, Klaus Havelund, and John Penix

Computational Sciences Division
NASA Ames Research Center
M.S. 269-2
Moffett Field, CA 94035

Abstract. NASA is developing technology for the next generation of deep-space robotic spacecraft, with the aim of enabling new types of missions and radically reducing costs. One technology under development is Autonomy: highly capable spacecraft that perform significant scientific missions with little or no commanding and monitoring from Earth. Artificial Intelligence provides a basis for autonomy technology, but raises issues of verification and validation outside the scope of empirical testing technology for conventionally commanded spacecraft. This paper describes research towards extending formal methods verification techniques for the mathematical verification of AI systems controlling deep-space spacecraft. This paper first overviews a planned space mission called DS-1 which includes an AI-based autonomy experiment. It then describes part of this AI system called the executive, which includes an 'intelligent' operating system based on goal-oriented constructs. The paper then describes focused research on applying and extending model-checking technology for verifying both the core services of the executive and the concurrent task programs run by the executive.

1 NASA's New Millennium Program

The successful landing of Mars Pathfinder on Independence day (July 4, 1997) signalled a new era in man's exploration of the solar system: faster, better, and cheaper. The Mars Pathfinder project was completed in four years, delivered widely sampled geological data from a mobile rover and cost just $250 million (1997) dollars. In contrast, the two Viking missions of twenty years ago took over eight years to develop, delivered data from fixed landers and orbiters, and cost over $3 billion (1997) dollars. The Mars pathfinder project took advantage of off-the-shelf technology to reduce development costs.

NASA is preparing for an order-of-magnitude expanded space exploration program in the next decade within the constraints of a flat-lined budget. One key aspect of this plan is the New Millennium program: a series of technology validation flights whose objective is to accelerate the flight-qualification of new spacecraft technology. For example, new generations of radiation-hardened microprocessors, based on commercial designs, will be flight-qualified in New Millennium missions. Up to now the functional performance of space-qualified hardware has often lagged a decade or more behind commercial hardware. New Millennium will greatly accelerate the space-hardening and space qualification of new technology. This will reduce development costs for subsequent science-oriented missions and enhance the technology base for these missions. The New Millennium program is also aimed towards decreasing operations costs while enhancing science return.

Operations costs are largely determined by two factors: launch weight and personnel. Microelectronics and other miniaturization technology can greatly reduce the launch weight of a deep-space robotic spacecraft. This alone can save hundreds of millions of dollars by reducing the size of the required launch rocket; for example, from a Space Shuttle to a Titan rocket to a Delta rocket. Personnel costs are the second major factor in operations costs, with missions such as Voyager requiring hundreds of non-science personnel at the peak of encounters to monitor the health of spacecraft subsystems, sequence the commands executed by the spacecraft, and navigate.

2 Deep-Space 1 and AI-Based Autonomy

As one of its objectives, the New Millennium program is seeking to reduce non-science personnel costs by an order of magnitude through automation technology. The ultimate objective of the automation technology is spacecraft autonomy: deep-space robotic spacecraft that navigate themselves, monitor their own health, monitor the status of mission goals, and take appropriate corrective actions. Autonomous spacecraft will be commanded at the level of mission goals rather than traditional spacecraft sequences, the latter being roughly equivalent to macro-assembler programs for embedded systems. This will be accomplished by extending the feedback loops which on today's spacecraft operate only at the low level of servo-mechanisms (e.g., maintaining attitude control) to feedback loops which operate all the way up to science goals. In addition to greatly reducing operations costs, autonomous spacecraft will also enable new kinds of missions that cannot be accomplished through light-time delayed remote control, such as comet landings.

The first New Millennium mission is Deep-Space 1 (DS-1), which currently includes plans for an asteroid rendezvous, a Mars fly-by, and a comet fly-by. Critical technology being tested includes an ion propulsion system (IPS) and on-board optical navigation. Another experiment is the flight-testing an AI-based control system called Remote Agent (RA), the first towards the goal of spacecraft autonomy. The remote agent has three subsystems: a planner which decomposes goals into task-nets and then sequences tasks along a time-line according to precedence constraints and resource constraints; an executive which concurrently executes these tasks; and a next-generation fault detection and recovery system called MIR. The executive subsystem and MIR provide the feedback loops at higher levels. The executive subsystem combines features of multi-threaded operating systems with AI languages based on sub-goaling, such as Prolog and MRS. This paper describes research aimed at formal verification of the executive subsystem.

3 Analytic Verification Effort

A major concern for spacecraft engineers and mission planners is verification and validation of advanced software architectures such as the remote agent. Traditional approaches to verification and validation include extensive testing and manual review. Furthermore, traditional spacecraft *sequences* - that is, command sequences to control a spacecraft - are time-stamped, straight-line programs that are reviewed by engineering teams from each spacecraft subsystem. For example, the thermal team will review a sequence to check for overheating, the power system team will review a sequence to check for safety of the electrical system. Each command is time-stamped to execute at a particular time, down to millisecond precision. More flexible methods for commanding a spacecraft are controversial - even minimal extensions such as *conditional sequencing*, where the execution timing of a command depends on the spacecraft's environment. An example would be a command to wait until the spacecraft comes out of the shadow of a planet before turning the solar panels towards the sun. Traditionally, all feedback loops above the level of servo-mechanisms go through ground manual control, with human-in-the-loop verification. Verification and validation is a major obstacle in accepting more autonomous architectures.

As an adjunct to the regular testing effort for DS-1, an effort among several NASA centers to apply formal methods to verifying and debugging DS-1 is being pursued. In previous NASA formal method studies the software being analyzed was developed in a waterfall process, with relatively long phases for requirements and design (months to years). This allowed plenty of time for formal methods practitioners to exercise their analysis tools and feedback their results to requirements analysts. In contrast, the DS-1 software has been developed through the spiral model [1]. The spiral model is an iterative software development process with four or more distinct phases for each turn of the spiral: requirements, design, coding, and testing. In contrast to the one-shot waterfall model, the spiral model provides iterative feedback on requirements and design through testing and validation of earlier turns of the spiral. It is a process well-suited to developing novel architectures where there is substantial initial uncertainty on requirements and designs. However, it results in highly time-compressed phases. In the case of DS-1, each turn of the spiral has lasted two to three months, and each phase of each spiral lasts just a matter of weeks. This presents a substantial challenge to formal methods efforts, stressing automation over labor-intensive manual methods.

The NASA-wide effort is being called analytic verification, to stress its complementing empirical validation efforts and to highlight its foundation in mathematical approaches to software engineering. We believe this name is also more informative to the spacecraft engineers and mission planners than the name 'formal methods'. As part of this analytic verification effort, our team at NASA Ames is pursuing formal verification of part of the remote agent. The technology we are developing stresses model-checking over interactive theorem-proving, because the former is much more automated than the latter. The long-term goal of our effort is to make our technology directly usable by the design and development team, as low-overhead tools. In the interim, we are our own users, providing feedback from our efforts to the design and development team.

4 Executive Core Services

The executive subsystem of the remote agent is conceptually composed of two layers: a set of core services that implement a robust operating system for executing concurrent tasks, and a set of mission-specific task programs. The language in which both layers are written is an extension of Lisp called the Executive Sequencing Language (ESL), principally authored by Eran Gatt of JPL. ESL is defined through a set of Lisp macros. ESL code operates in a multi-threaded Lisp.

In this section we overview one aspect of the executive that will serve as examples of the formal verification techniques described in section 5: the *with-maintained-properties* construct in the core executive system. Section 6 describes an example task program for thrusting with the ion propulsion system, which will serve as examples for the abstraction techniques described in subsequent sections.

The services provided by the core executive system, such as those provided by the *with-maintained-properties* construct described below, provide an API (application programming interface) for defining task programs that achieve diverse goals such as acquiring images and changing trajectory. These programs are executed concurrently in order to achieve the token nets generated by the planner. For the purposes of this paper, assume that each token in the token net corresponds to the invocation of a task program in the executive, with the appropriate parameters. The core services of the executive include as a subset those of an operating system for managing a queue of jobs, which must be assigned resources and synchronized according to their concurrency constraints. The executive keeps an agenda of tasks (jobs) which are broken down into sequences of commands; the executive activates and suspends these tasks. The resources are typically states of a device, more generally, resources are *properties.* A property is any value that is controlled and monitored. The executive differs from a standard operating system in at least two respects: the extensive support for monitoring and recovering from property violations - leading to robust execution, and the high level of abstraction provided by goal-oriented AI constructs.

4.1 Maintaining Properties during Task Execution

This subsection describes the architecture implemented in the executive for assigning and monitoring properties required by task programs for successful execution. The analytic verification and debugging of this architecture is described in the next section. The primitive construct that invokes these services is the (*with-maintained-properties* **prop body***)* construct (implemented as a Lisp macro), which takes as parameters a set of properties and the body of a task program. The specification of this construct is that it guarantees the properties while the body is executing. By having the properties managed in this manner, the task programs can focus on sequencing activities without worrying about maintaining and monitoring the values of the maintained properties.

Further constructs are built using the *with-maintained-properties* construct. For example, the (*with-selected-device* **class** (**do-activity**)) construct selects a device of the class, achieves its ready-state, and then locks the properties of that ready-state and maintains them with the assistance of MIR while it does the activity. This ESL construct is used within the DS-1 executive to maintain properties of devices, which are physical objects on the spacecraft. There are classes of devices which are defined to have various properties. A task can select a device and execute with the knowledge that the desired state of this device is being maintained by the system. Further constructs built using the *with-maintained-properties* construct are used to maintain states during sequencing. For example, thrusting in the ion propulsion system is controlled by creating a thrusting state predicate and associating it with the property IPS-THRUSTING.

The basic architecture in which *with-maintained-properties* operates is a collection of concurrent tasks that require specific values of certain properties in order to execute correctly. If a property value become false, then the associated tasks should be suspended while action is taken to re-achieve the property. Tasks must also be coordinated so they do not require conflicting values of properties. The following are some of the basic protocol requirements for this architecture whose analytic verification and debugging is described in section 5:

1. A task wanting a property to have a specific value subscribes to the property. To coordinate multiple tasks wanting property values, the properties are locked during *subscription*. Property locks are used to coordinate tasks so that they do not try to achieve different values for a single property at the same time. Mutually exclusive access to the locks is guaranteed by placing the code for locking within a critical section. The subscription process can have three outcomes for a task:

1a. No other task is subscribing to that lock, in which case the subscription is successful, this task becomes the owner of the lock, and *snarfs* the property.

1b. Some other task already owns lock, and the values that the two tasks want the property to have are compatible. In this case the task successfully subscribes to the lock but does not become the lock's owner - it is a secondary subscriber. It also *snarfs* the property.
1c. Some other task is subscribing to the lock and the values are incompatible. In this case the subscription FAILS.

2. The owner of a lock, after successfully subscribing, attempts to actually make the property true by calling the **achieve** method on the property. All secondary subscribers wait for the property to be achieved by the owner. If the owner's attempt to achieve the property fails, then all of the lock's subscribers **fail**.

3. Once a lock property has been achieved, the lock's subscribers, which were waiting for the owner to achieve the property, are signaled and continue to run. The body of these tasks are executed.

4. If a locked property becomes false (is violated) during execution of a body (through events exogenous to the executive, such as device failures), signal this loss, suspend execution of the body, and attempt to recover the property. Maintained property violations are detected by a daemon. This daemon will attempt to restore or recover violated properties, using recovery actions that are either explicitly stated in the task definition, or default recovery actions that can involve invocation of MIR. If the recovery action is not successful, then all tasks which have snarfed the property are aborted.

5. When a task terminates, either normally or aborted, it unsubscribes to all properties it has snarfed. If the owner of a locked property unsubscribes to the property, then ownership is passed to the next task subscribing to the property. If there are no further tasks subscribing to the property, then the lock on the property is released.

The next section describes a simple error trace found through analytic verification that violates requirement 5 and then a more complex error trace that violates requirement 4.

5 Analytic Verification of Executive Core Services

This section describes ongoing work to verify and debug key components of the core services of the executive. An abstracted model of the Lisp code was manually developed of the implementation for locking, snarfing, and releasing properties. This model includes the *with-maintained-properties* construct described in the previous section. The model was written in PROMELA, the language for the model-checker SPIN [2]. Even this highly abstracted model was at the limits of computational complexity for large workstations (i.e., a workstation with 268MB of memory).

Model-checking is a formal methods technique for verifying and debugging concurrent or real-time systems modeled as interacting finite state machines. Given a model and a property, a model-checker searches for *traces* of the model that violate the property. Properties can be invariants, temporal properties (i.e., defined through model operators such as *eventually*), or in the case of real-time model-checkers, metric time constraints defined through linear relations. A trace is an interleaved sequence of states (or dually, transitions) of the finite state machines. Model checkers differ from simulators in that they explore all possible traces, in other words all realizable paths through the reachability graph. They also enable checking much richer concurrency properties than is typical of simulators. Some model checkers are similar to theorem-provers in that they manipulate symbolic descriptions of the transition relation. However, model-checkers do not perform induction, which typically needs to be guided manually in real-world verification proofs. Trading off from this deficiency, they are completely automatic, and thus more practical for verification in a spiral development process.

Different model checkers provide different languages for defining the interacting finite state machines. The PROMELA language is a C-like language with facilities for creating processes and for interprocess communication through buffered channels. SPIN compiles a PROMELA model into a C program which is then run to find error traces. To formulate the model of the executive core services in PROMELA, a number of trade-offs had to be made. For example, the spring 1997 release of PROMELA did not directly support procedures. Modeling procedure calls as process invocations led to state explosions, a subsequent attempt at modeling procedure calls as macro calls avoided the state explosion but required careful separation of variable names. Neither the precise details of our abstracted PROMELA model nor the various trade-offs are enumerated in this paper. However, the reader should be aware that formulating computationally tractable models of complex systems requires expertise, artistry, and experimentation.[1]

This section focuses on two subtle bugs that were found regarding the protocol requirements described in section 4. These bugs were first found through error traces of our abstracted PROMELA model and then confirmed in the executive Lisp code. The designers of the executive believe these bugs would not have been detected through means other than formal verification. Empirical testing would not have uncovered these bugs because they arise only in a multithreaded environment, and only in unusual circumstances that are unlikely to arise during testing. Model-checking is able to find these bugs because it considers all possible interleavings of concurrent processes.

We should first note that the executive has performed well under empirical testing, and moreover we have been able to analytically verify that the implementation satisfies various requirements. For example, the following requirements have been verified, in the abstracted PROMELA model, of the property protocols described in section 4:

1. If a property is snarfed by a task, it can only be snarfed by another task if the property values are compatible (requirement 1b).
2. When a task terminates normally, it releases all locks (requirement 5, normal termination).

The verification of these requirements was achieved by placing invariants in the model at the appropriate locations, and running the model over all interleavings to demonstrate that there are no traces where the invariants are violated.

The subtle bugs which have been found through model-checking concern traces in abnormal situations, for example, where a property is violated in a situation that was not considered by the executive design team. The first bug is demonstrated by an error trace where a task may abort without releasing its property locks. The *with-maintained-property* construct uses the Lisp construct *unwind-protect* to provide a wrapper around the execution of the body. This ensures that errors signalled during the execution of the body (e.g., an error signalled because a locked property is violated) are caught and han-

1. The authors thank Gerard Holzmann, the creator of SPIN, for his advise and help.

dled by recovery procedures. However, after the body is executed, the *with-maintained-property* construct exits the *unwind-protect* wrapper and executes code to release the locks. If after the *unwind-protect* wrapper is exited a property violation occurs, and then the daemon which monitors property violations wakes up, it will signal an error. Because this error occurs outside the dynamic context of an *unwind-protect,* the error falls through and causes the task to be aborted - whether or not the release locks code has been fully executed.

This error trace was communicated to the development team, resulting in experiments with a critical section being placed in the *with-maintained-property* construct around the code to release locks. This prevents any other thread from executing (including the daemon monitoring property violations) when any instance of *with-maintained-properties* enters the release-locks section of code. This still leaves an even more unlikely (by over an order of magnitude) error trace that was found through model-checking: a property violation occurs exactly at the point where the *unwind-protect* wrapper is exited and the gap before the critical section for release locks is entered. The best means of handling this unlikely error trace is not obvious, because it is necessary to be conservative in creating critical sections of code. Excluding other threads from executing can itself lead to timing bugs. This brief overview of this error trace illustrates the nature of the interactions between the analytic verification team and the executive development team, and also the subtle nature of concurrency bugs.

An even more subtle error trace concerns the interaction of the internal operation of the daemon which monitors the properties and the code for achieving property locks. Simplified versions of the Lisp code for both, circa the spring of 1997, are given below in Algol-like notation, preceded by explanations. The simplified versions are sufficient for understanding the nature of the error trace, and for understanding how the verification technology is applied to AI programming constructs.

The maintain-properties-daemon is an infinite loop, and is normally sleeping. First it checks the locks data structure; if there is an inconsistency with the database for the actual property values, it invokes recovery procedures. Then it determines whether there has been any new memory-event occurrences (induced by an external event) or snarf-event occurrences (induced by a task snarfing a property). If not, it goes back to sleep until a memory-event or snarf-event occurs. Hence, if no event occurs for a while, then the damon will not perform check-locks.

maintain-properties-daemon

```
 loop-forever do
        if  check-locks
            do-automatic-recovery;
        if  not(changed? (memory-event-count + pl-snarf-event-count))
            then wait-for(memory-event or snarf-event);
   od
```

The **achieve-lock-property** routine is called within the dynamic scope of a task, which is the value of $this-task. This routine is called after a task successfully snarfs a property-lock; the routine is called with the lock as a parameter. If the task is the owner, then it first achieves the property and then sets the achieved-field to boolean TRUE. If the task is not the owner of the lock, it waits for the property to be achieved.

```
achieve-lock-properties (lock)
   p = property-lock-property(lock);
   if  owner(lock) =  $this-task then do
       achieve p;
       property-lock-achieved?(lock) = TRUE;
       od
    else wait-for(p);
```

An error trace arises in the SPIN model for the following requirement, which is part of requirement 4 listed in the previous section. In contrast to the previous requirements, which were formulated as invariants, this requirement is formulated as a temporal property:

> If a task relies on some property to hold, and the database is modified
> such that it no longer holds, then the task will eventually terminate,
> either by itself or by an abort from the daemon.

The error trace is summarized by the following sequence of events which refer to the interaction between the environment, the daemon, the **achieve-lock-properties** routine, and multiple tasks:

1. A task calls **snarf-property-locks,** becomes the owner of the lock, and then calls **achieve-lock-properties.** In the call of the latter, first achieve is called (because the task is the owner of the lock). After this succeeds, but before the achieved-value is set to TRUE, the task is suspended (put to sleep) by the excutive.That is, the property has been achieved, but the achieved-field in the property lock has not yet been set to TRUE. (A second task is activated by the executive and starts to run).

2. The database is now modified by the environment in such a way that it becomes inconsistent with the property lock just created by the first task. This causes a memory event.

3. The daemon is awakened. The daemon starts looking for an inconsistency, but finds none since the achieved-field has not been set yet. Specifically, check-locks signals an inconsistency only if the achieved-field is set to TRUE. Hence, the daemon discovers nothing and goes back to sleep.

4. The first task is awakened and finishes executing the **achieve-lock-properties.** It assigns TRUE to the achieved-field of the lock, and continues execution as if everything is consistent.

5. Unless another memory-event or snarf-event occurs, the daemon remains asleep. The task is not stopped, and continues executing (if it is an infinite loop, perhaps indefinitely) even though properties it depends on are not valid.

This error trace could be eliminated through a critical region in the **achieve-lock-properties** routine. However, it also reinforces a known design flaw of the spring 1997 implementation of the executive: there can be a significant time lag between a property being violated and a task being informed of the violation. Prior to model-checking, it was not known that this time lag could be indefinite. This significant time lag is being eliminated in the design of the next version of the executive by changing where the property lock daemon resides and when it is invoked.

This section has overviewed aspects of the analytic verification and debugging of the DS-1 remote agent executive. It demonstrates that formal methods technology, specifically model-checking, which has been successfully applied in the past to verification and debugging of digital hardware, communication protocols, and operating system protocols; can be extended to the verification and debugging of AI-based concurrent systems. However, it has not described the manual effort in hand translating the Lisp code to the language of a model-checker (e.g., PROMELA), the much more considerable effort in abstracting the model so it is computationally tractable even for a large workstation. This required manual effort has made it somewhat difficult to stay current with the executive design and development team working under the rapid spiral model. Nonetheless, we have made a contribution to the overall verification effort for the DS-1 remote agent.

6 Executive Task Programs

This section and the next section together describe research towards automating the abstraction of AI programs for computationally tractable model-checking. The context for this research is verification of the task-specific executive programs, which run on top of the API provided by the core executive services.

The task-specific programs need to be verified both against the goals they are meant to achieve as well as the constraints they cannot violate while executing. A subclass of these constraints are *flight rules* which are formulated by experts to guide the safe operation of the spacecraft. One class of flight rules have the form that a class of events never occur in states satisfying particular predicates:

[never] state-predicate -> event

In other words, the event should only occur in states that satisfy the negation of the state-predicate. An example of such a flight rule concerns the requirements for the spacecraft to have directional control over the propulsion. The Engine Gimbal Actuator (EGA) must be activated to provide directional control over the thrust vector. Thus one implication of the requirement is the following flight rule: the EGA must be on whenever the attitude control system (ACS) is in Thrust Vector Control (TVC) mode. This can be formulated as two rules, which together are equivalent to the constraint that a time interval in which the ACS is in TVC mode is contained by a time interval when the EGA is on:

[never] EGA-off -> turn-on-TVC-mode
[never] TVC-mode-on -> turn-off-EGA

The executive controls devices such as the EGA and the ACS by broadcasting messages to processes that subscribe (listen) to particular classes of messages, these processes implement the low-level device control protocols. Thus from the viewpoint of the task-specific executive programs, flight rules such as the one above are equivalent to predicates on the state in which particular messages are allowed to be sent by the executive.

The rest of this section describes a procedure within the executive task program that achieves a thrust level of the ion propulsion system. It contains a call to change the ACS into TVC-mode. It is a simplified version of the procedure as it existed in the spring of 1997. The part of the procedure which sends out a message to the ACS is the subroutine call **change-acs-mode**.

```
to-achieve-IPS-THRUSTING (level)
    if (ips-thrusting-state-p or ips-standby-state-p)
      then do
          check-type( level, ips-thrust-level-type);
          command-with-mir-confirmation(send_fsc_ips_set_thrust_level(level));
          wait-for(memory-event(ips-state = steady-state and thrust-level = level));
          with-guardian(warp-safe (monitor(ips-thrust-duration-achieved))),
                                                change-acs-mode(tvc_mode));
     od
     else do
          achieve (ips-standby-state);
          achieve (ips-thrusting(level));
     od
```

The procedure first checks that the IPS is either in a thrusting state or a standby state. If neither (the else clause), it first achieves a standby state, and then tries to achieve the thrusting level. Otherwise (the then clause), it performs a sequence of steps that do not themselves change whether the EGA is turned on. These commands instead set-up and monitor the thrust-level given as parameter. Finally, the ACS is switched to TVC-mode.

Note that this procedure does not itself guarantee that the EGA is turned on when the ACS is switched to TVC-mode. (In fact, achieving IPS standby-mode turns the EGA off.) Thus the existence of a possible error trace for the flight rule is not excluded by the code in this procedure, but rather depends on the context in which this procedure is called. However, the program is more complicated than need be to determine whether or not there is an error trace. The superfluous steps can greatly increase the number of interleavings during model-checking.

7 Abstraction-based Verification

Model-checking is an attractive method for verifying flight rules, specifically that state constraints (invariants) are satisfied at those points in the code where messages are broadcast. However, the executive task descriptions are moderately complex (tens to hundreds of pages), and have a large state space. The code described in section 6 is only a small part of the overall program to control the IPS. This makes it computationally infeasible as a general methodology to run a model-checker on a direct translation of the ESL programs. But for each flight rule, only a small portion of the program is relevant. However, given the number of flight rules, manually abstracting a model of the task programs for each flight rule is not cost effective. Thus we are developing automated abstraction methods for generating reduced models. As applied to the code described in section 6, the abstraction methods would yield the simplified program below with respect to the EGA constraint:

```
to-achieve-IPS-THRUSTING (level)
    if (ips-thrusting-state-p or ips-standby-state-p)
      then do
         change-acs-mode(tvc_mode);
     od
     else do
          achieve (ips-standby-state);
          achieve (ips-thrusting(level));
     od
```

7.1 WP-Based Program Slicing

Program slicing [4,5] is a technique to extract a partial program that is equivalent to an original program over a subset of the program variables. The input to a traditional slicing algorithm is a program and a designated subset of variables, the output of the slicing algorithm is a partial program that for every program execution has identical values assigned to the designated subset of variables upon program termination. The key idea of slicing algorithms is to work backwards from the program end-point, keeping statements that have an effect on the designated variables and removing statements that have no effect. As the algorithm works backwards over the program statements, additional variables might be added to the designated set if they have occur in expressions which modify the designated set.

The concept of program slicing can be extended to abstract ESL programs for the purpose of model checking. However, instead of slicing with respect to variables, the programs are sliced with respect to state predicates [6], starting from a statement which contains the operation(s) which are required to only be executed in particular states. In other words, the programs are sliced with respect to a never property of the form described in section 6. The partial program that is generated from such a never property has the following guarantee: its error traces for the never property are in direct correspondence to the error traces of the original program. This is called WP-program slicing, as the algorithm is defined using Dijkstra's weakest-precondition calculus. In the example above, the slicing begins at the statement which changes the ACS to TVC-mode.

Two rules for this slicing calculus are described. **Slice** takes a code-sequence terminated by a statement, and a state predicate, and returns a reduced code-sequence. It determines where it can substitute the IDLE statement without affecting error traces by calculating when a statement "passes through" a state predicate. This occurs when the weakest precondition is the same as the state predicate. The IDLE statements can later be removed.

The slicing rule for straight-line sequences is:

```
Slice({code-sequence ; statement} ,  state-predicate)  ->
If WP(statement, state-predicate) = state-predicate
   then {Slice(code-sequence,state-predicate); IDLE}
   else  {Slice(code-sequence,WP(state-predicate,statement)) ; statement}
```

The rule states that IDLE can be substituted for a statement if the statement has no effect on the state predicate. Otherwise, the weakest precondition of the state-predicate is substituted for the state-predicate and slicing continues backwards. A more elaborate calculus would allow substituting a simpler statement that had the same weakest precondition.

The rule for conditionals requires the definition of a weakest predicate reduction:

Definition R is a *predicate reduction* of Q by P iff R & P => Q

Definition R is a *weakest predicate reduction* (WPR) of Q by P iff for any S which is a predicate reduction of Q by P, S=> R

The rule for conditionals is defined on the following pattern:
Slice({code-sequence; if *P* then *then-statement* else *else-statement*}, Q}

Let *then-WP* = WP(*then-statment, Q)*
Let *else-WP* = WP(*else-statment, Q)*

Let *reduced-then-WP* = WPR(*P,then-WP)*
Let *reduced-else-WP* = WPR(not(*P),else-WP)*

Note that when the condition *P* implies *then-WP,* the IDLE statement can be substituted for *then-statement.* In other words, the *then-statement* is executed only in a context where it will not generate an error trace. Define *reduced-then-statement* as IDLE when *P* implies *then-WP*, otherwise it is the same as *then-statement.* Similarly, when not*(P)* implies *else-WP,* the IDLE statement can be substituted for *else-statement.* Define *reduced-else-statement* accordingly.

The rule for slicing conditionals can now be defined:
Slice({code-sequence; if *P* then *then-statement* else *else-statement*}, Q} ->

{Slice(code-sequence, (*P* and *reduced-then-WP)* or (not(*P*) and *reduced-else-WP)))*
; if *P* then *reduced-then-statement* else *reduced-else-statement*}

Note that the weakest precondition for the conditional is composed of reduced weakest preconditions for the then-statement and the else-statement.

7.2 Future Research on Abstraction-based Verification

Future work will include a full calculus for WP-slicing of ESL constructs. Many of the ESL constructs such as *with-maintained-properties* greatly simplify WP-slicing. In other words, the API provided by core executive services is designed to simplify ensuring that requirements and constraints for task program execution are met. For example, *with-maintained-properties* is specified to guarantee that properties are achieved and maintained throughout the execution of the body. The weakest precondition of the body for a state-predicate implied by these properties is simply **true.**

Another method under investigation for abstracting ESL task programs is equivalence-based abstract interpretation.WP-program slicing can substantially reduce the size of programs. However, it does not reduce the size of the full state space, defined as the product of the number of values that can be assigned to each program variable. By reducing some state transitions to IDLE, WP-program slicing reduces the number of reachable states, defined as the subset of the state space that can be reached through sequences of valid transitions from the initial state. Nonetheless, non-symbolic model-checking is required to enumerate all the possible reachable values for each program variable, even if many of these values result in equivalent traces with respect to an invariant or temporal property.

The state space can be greatly reduced by merging equivalent values, and redefining the operations according to the equivalence classes. Mapping values to equivalence classes and operations to abstracted operations over the equivalence classes is a type of *abstract interpretation*, [3]. Abstract interpretation is defined as a homomorphism from the concrete domain of values and operations in a programming language to an abstract domain, typically with values ordered in a lattice and abstracted operations defined through lattice operations (e.g., meet and join). Abstract interpretation is often applied through fixed mappings for various types of analysis used in compiler optimization, such as dead code detection.

In this ongoing research, we are applying abstract interpretation through dynamically determined mappings to reduce the state space for model-checking. To date, we have only considered mappings based on explicit values: discrete enumerated values and convex intervals over the real numbers. The method is similar to WP-based program slicing: starting from an operation constrained by a flight rule, the state space is merged into one equivalence class. Walking backwards through the code from this operation, this one equivalence class is then recursively partitioned into distinct classes according to those states satisfying the weakest precondition for each statement. At worst, the original state space is regenerated.

8 Summary

Verification technology is important for next generation of autonomous spacecraft. This paper has described ongoing work in applying and extending formal methods techniques, specifically model-checking, to the verification of an AI-based autonomy architecture. The paper presented examples of the core services provided by the DS-1 remote agent executive and task programs executed by the executive. Verification and debugging of the core services protocol requirements through model-checking were described. The paper then described on-going research to automate the abstraction of task programs in order to enable model-checking of flight rules and other requirements.

Acknowledgments

This research was supported by NASA Code S and NASA Code Q . We wish to thank our colleagues at other NASA centers participating in the analytic verification effort for DS-1 for their helpful suggestions in this research. We wish also to thank various members of the model-checking research community for their advise.

References

1. B. W. Boehm: A Spiral Model of Software Development and Enhancement. *ACM Sigsoft Software Eng. Notes 11(4):22-42*.
2. G. J. Holzmann: The Model Checker SPIN, *IEEE Transactions on Software Engineering*, Vol 23, No. 5, May 1997.
3. N. Jones, C. Gomard,and P. Sestoft: *Partial Evaluation and Automatic Program Generation,* ed. C.A.R. Hoare, Prentice Hall, 1993.
4. B. Korel and J. Laski: Dynamic Slicing of Computer Programs, *J. Systems Software,* Vol 13, 1990, pp 187-195.
5. M. Weiser: Program Slicing, *Proc. Fifth International Conference on Software Engineering,* 1981, pp. 439-449.
6. J.J. Comuzzi and J.M. Hart, “Program Slicing Using Weakest Preconditions”, *3rd Intl. Symposium of Formal Methods Europe (FME’96)*, M. C. Gaudel, J. Woodcock, eds., Oxford, UK March 1996. Springer Verlag Lecture Notes in Computer Science 1051.