

Model Checking Artificial Intelligence Based Planners: Even the best laid plans must be verified

Margaret H. Smith
Jet Propulsion Lab
4800 Oak Grove Drive
M/S 198-326
Pasadena, CA 91109
(818) 393-7521
margaret@jpl.nasa.gov

Gordon C. Cucullu, III
Jet Propulsion Lab
4800 Oak Grove Drive
M/S 156-142
Pasadena, CA 91109
(818) 393-3842
Gordon.C.Cucullu-III@jpl.nasa.gov

Gerard J. Holzmann
Jet Propulsion Lab
4800 Oak Grove Drive
M/S 126-110
Pasadena, CA 91109
(818) 393-5937
Gerard.J.Holzmann@jpl.nasa.gov

Benjamin D. Smith
Jet Propulsion Lab
4800 Oak Grove Drive
M/S 126-301
Pasadena, CA 91109
(818) 393-5371
Benjamin.D.Smith@jpl.nasa.gov

Abstract— Automated planning systems (APS) are gaining acceptance for use on NASA missions as evidenced by APS flown on missions such as Earth Orbiter 1 and Deep Space 1, both of which were commanded by onboard planning systems. The planning system takes high level goals and expands them onboard into a detailed plan of action that the spacecraft executes. The system must be verified to ensure that the automatically generated plans achieve the goals as expected and do not generate actions that would harm the spacecraft or mission. These systems are typically tested using empirical methods. Formal methods, such as model checking, offer exhaustive or measurable test coverage which leads to much greater confidence in correctness.

This paper describes a formal method based on the SPIN model checker. This method guarantees that possible plans meet certain desirable properties. We express the input model in Promela, the language of SPIN [1] [2] and express the properties of desirable plans formally. The Promela model is then checked by SPIN to see if it contains violations of the properties, which are reported as errors. We have applied this approach to an APS and found a defect.

TABLE OF CONTENTS

1. INTRODUCTION.....	1
2. CONTRASTING EMPIRICAL TESTING WITH VERIFICATION.....	2
3. MODEL CHECKING CASE STUDY	3
<i>CASPER Overview</i>	3
<i>SPIN Overview</i>	3
<i>Example Planner Domain Model</i>	3
<i>Planner Domain Model Expressed in Promela</i>	4
<i>Model Tractability</i>	6

<i>Expressing the Property of Interest</i>	6
VERIFICATION RESULTS	8
CONCLUSIONS	9
ACKNOWLEDGEMENTS	9
REFERENCES	9
BIOGRAPHY	10

1. INTRODUCTION

Automated Planning Systems (APS) have commanded two NASA technology validation missions: DS1 and EO1. Unlike traditional flight software, which executes a fixed sequence, an automated planning system takes as input a few high level goals and automatically generates a sequence (plan) that achieves them. The plan can be modified onboard in response to faults, new situations, and unexpected execution outcomes. This added flexibility allows the system to respond to unexpected situations and opportunities that a fixed sequence cannot. However, this same flexibility also makes a planner far more difficult to verify. The plan must be shown to generate the correct plan for a vast number of situations. Empirical test cases can cover only a handful of the most likely or critical situations. Formal methods can prove that every plan meets certain properties.

APS systems of a *planner domain models*, the planning engine, or *planner*, and the *executor* that carries out the plan. Each of these components is a potential test target. Verification of the planner domain models is our focus in this work. We want to answer the question: how do we know that an APS will produce only desirable plans when it is flown? The cost of a bad plan can potentially be very high, ranging from loss of science return to loss of an entire multi-million

dollar mission. Once a planner generates a plan, we can prove that the plan is consistent with the planner domain model provided to the planner, but there is currently no method to check that the planner domain model will allow only desirable plans.

Many safety considerations and flight rules can be captured directly as constraints in the planner domain model. However, certain properties, such as what constitutes an acceptable overall plan, can not be enforced directly in the planner domain model. For example, in a system consisting of a camera, solid state recorder and a radio, we would want to ensure that for all plans, if an image is taken and stored, it is eventually uplinked. There is no way to express this type of desirable property directly in the planner domain model and so in this work we have developed a technique to verify AI input model compliance with desirable plan properties.

In other work, the real-time model checker UPPAAL was used to check for violations of mutual exclusion properties and to check for the existence of a plan meeting a set of goals [3]. In contrast, the work reported in this paper shows that for verification of a set of properties of interest, it is not necessary to reason about time. SPIN has also been used to verify plan execution engines [4] [5]. A comparison of three popular model checkers, SPIN, SMV and Murphi showed that these model checkers can be used to check for the existence of a plan meeting a set of goals and to check that from any state in an AI input model it is possible to reach a desired goal state [6]. While this last work demonstrates existence of a single desirable plan or the possibility of reaching a goal state from any model state, it does not seek to check models for the presence of undesirable plans in an AI input model.

CONTRASTING EMPIRICAL TESTING WITH VERIFICATION

As shown in Figure 1, the empirical method for testing AI Models is *test plan generation* that includes these steps: inspect the AI model, request a finite number of sample plans from the APS, and manually inspect the plans to determine if they are good or bad. The number of sample plans requested will correspond to the amount of time available for manual analysis of the plans. A typical number of plans requested may be in the order of 100 plans [7]. When a undesirable plan is discovered in the sample set, the constraints portion of the input model is adjusted to prevent that particular undesirable plan and the sampling and manual analysis is repeated until the sample set produced by the APS contains no undesirable plans.

In contrast, our approach, shown in Figure 2, uses the SPIN model checker to determine that the space defined by the AI input model either contains only desirable plans, or if undesirable plans exist, to expose the them explicitly as errors. If an exhaustive check is not tractable we use an approximation technique. In either case, our technique examines millions of plans, as opposed to the sampling method of the traditional testing process where only 100 or so plans are analyzed.

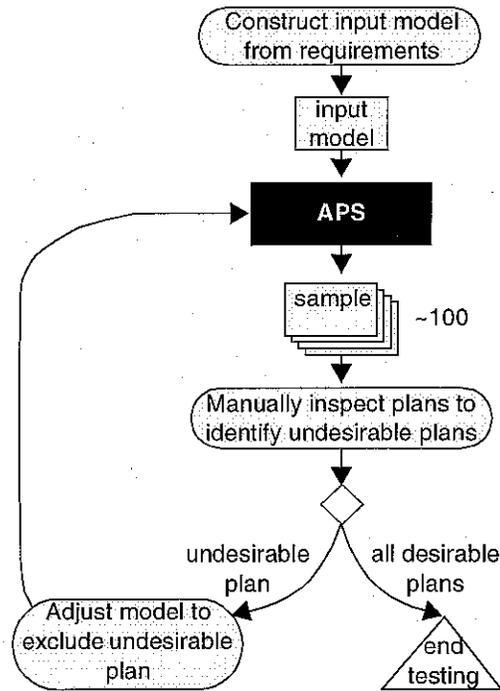


Figure 1 - Empirical Test Plan Generation

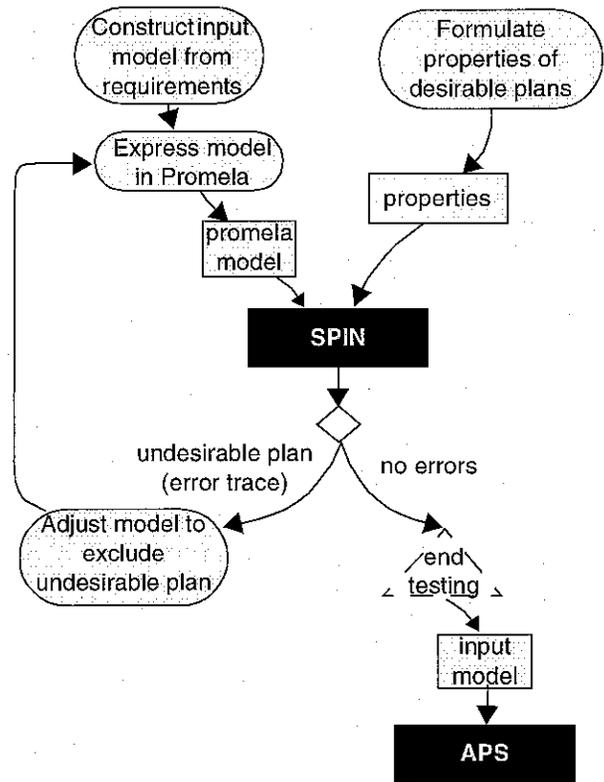


Figure 2 - Test process using the SPIN model checker

The AI Model is expressed first as a Promela model, the language of the SPIN model checker. We develop properties of desirable plans from mission requirements. The property is not an example desirable plan but a more abstract representation of the characteristics of a desirable plan. The properties are expressed formally using the Timeline Editor [8]. The AI input model is then checked exhaustively by SPIN to determine if it contains undesirable plans. If a undesirable plan is found, SPIN reports it as an error in the form of a sequence diagram. Human analysis of the undesirable plan will reveal that the input model is under or over constrained and an adjustment, such as the relaxation or addition of a constraint is made and the process is repeated until SPIN reports that there are no undesirable plans.

2. MODEL CHECKING CASE STUDY

In this section we show how SPIN verifies whether all plans generated by a planning system will meet certain properties. For purposes of this example, we chose a domain model for the CASPER continuous planning system. The model generates plans for a sample acquisition and analysis scenario of a possible comet landing mission. We chose this particular model because it is easily understood and has been documented in the literature [9]. The model generates plans for the sample acquisition phase which consists of collecting, 'baking' and evaluating terrain samples in a gas chromatograph/mass spectrometer and taking images of the comet surface.

CASPER Overview

CASPER is a Continuous Activity Scheduling Execution and Re-planning (CASPER) system build around a modular and reconfigurable application framework known as the Automated Scheduling and Planning Environment (ASPEN) [10]. ASPEN is a modular and reconfigurable application framework capable of supporting a wide variety of planning and scheduling applications that includes these components:

- an modeling language for defining the domain
- a resource management system
- a temporal reasoning system, and
- a graphical interface for visualizing plans

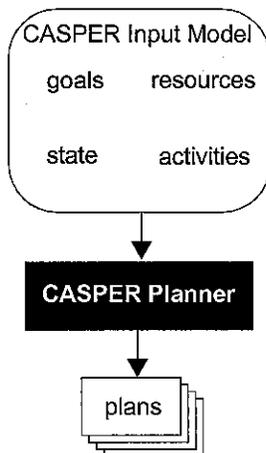


Figure 3 - ASPEN modeling language

The Continuous Activity Scheduling Planning Execution and Re-planning (CASPER) system supports continuous modification and updating of a current working plan in light of a changing operating context. Applications of CASPER have included autonomous spacecraft such as Earth Orbiter 1 and 3 Corner Satellite, Rover Sequence Generation, Distributed Rovers, and Closed Loop Execution and Recovery

An planner domain model to the CASPER system is expressed in the CASPER modeling language that includes the constructs shown in Figure 3. *Goals* are the high level science and

engineering activities performed by a spacecraft. An example goal is performance of a science experiment. *Activities* are actions and are the means by which goals are satisfied. A drilling activity might be one of the activities by which a science experiment goal is satisfied. Activities contain temporal constraints and resource reservations that must be satisfied for the activity to be scheduled. *States*, represented by state variables, are the means by which resource reservations are made and tracked. For instance, before drilling can occur the drill must be at the desired location. A *state* would be the means by which the drill location is tracked. State changes are performed by *activities*. For instance, the sample activity would change the drill location to hole1 if no other activity has a reservation on the drill location state variable. *Resources*, are those items that are necessary to, or used up in the course of, carrying out an activity. Drilling, for instance, will use battery power. Resources are updated by activities. For instance, when the sample activity begins drilling, it decrements the power resource and when drilling ends the power resource is incremented.

SPIN Overview

SPIN is a logic model checker that is used to formally verify distributed software systems. Development of the tool began in 1980 in the original Unix group of the Computing Sciences Research Center at Bell Labs. The software has been freely available since 1991, and continues to evolve to keep pace with new developments in the field.

SPIN is the most widely used logic model checker with over 10,000 users. In April 2002 SPIN was the recipient of the prestigious System Software Award for 2001 by the Association for Computing Machinery (ACM). SPIN verifies software, not hardware and has a high level language for describing systems, called PROMELA (a PROcess MEta LANGUAGE). Spin has been used to trace logical design errors in distributed systems designs, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

Example Planner Domain Model

The sample acquisition phase of model consists of taking three terrain samples and two images. Each sampling activity should contain a set of serial sub activities of: drilling, mining for a sample, moving the drill to the oven, depositing the sample in the oven and baking the sample and taking measurements. Other activities that were not part of the goals were data compression, used to partially free up memory in the data buffer, and data uplinking to the orbiter.

For the portion of the landed phase that we analyzed, the resources included:

- 2 ovens
- 1 camera
- 1 robotic arm with drill
- power (renewable)
- battery power (non-renewable)
- memory (renewable)

State variables included these: oven 1 and oven 2, camera, telecom and drill location. The legal states and state

transitions for state variables are shown in Figure 4. Default states are shaded. All transitions are allowed for the drill location, camera and telecom state variables. For the ovens, oven 1 / 2

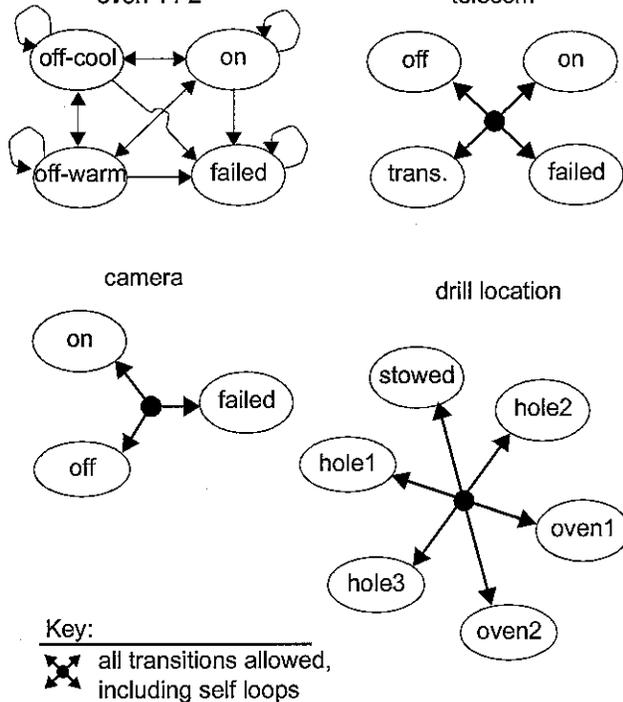


Figure 4 - example CASPER planner domain model state variables and state transitions

all transitions are allowed except for transitions out of the failed state. Once an oven enters the failed state it will stay in the failed state.

A small portion of the planner domain model is shown in Figure 5. The goals shown are two pictures, *take1*, and *take2*. An initial start time for this activity is seven hours after the beginning of the plan. The CASPER planner may move the activity start time but can not delete the activity as indicated by the *no_permissions* variable.

The resources shown in Figure 5 are the camera, the telecom device and the data buffer. The camera and telecom are atomic devices, meaning that only one activity can utilize the resource at a time. The data buffer is a shared resource that can be depleted, and has a capacity and a minimum value.

The *take_picture* activity may start between 10 minutes after the start of the plan and infinity and its duration may be between 1 and 10 minutes. Reservations that are needed in order for the *take_picture* activity to be scheduled are:

- exclusive access to the telecom device (comm)
- exclusive access to the camera (civa)
- use of 5 Mb of the data buffer
- and the camera (civa) must be "on" for the duration of the activity.

Planner Domain Model Expressed in Promela

To express the planner domain model in Promela we first observed that each activity in the CASPER model should

requests	resources
<pre>take_picture take1 { start_time = 7h; file = "IMAGE1"; no_permissions = ("delete"); }; take_picture take2 { start_time = 18h; file = "IMAGE2"; no_permissions = ("delete"); };</pre>	<pre>resource civa { //camera type = atomic; }; resource comm { type = atomic; }; resource data_buffer { type = depletable; capacity = 30; min_value = 0; };</pre>
activity	states
<pre>activity take_picture { start_time = [10, infinity]; duration = 1m, 10m]; reservations = comm, civa, data_buffer use 5, civa_sv must_be "on"; };</pre>	<pre>state_variable civa_sv { states = ("on", "off", "failed"); default_state = "off; };</pre>

Figure 5 - CASPER/ASPEN model for taking a picture

correspond to an instance of a Promela proctype. A Promela *proctype* is a process type that defines the behavior of a process instance. The *take_picture* activity, for instance, would be defined in Promela as a proctype, *take_picture*. In an initialization step, two instances of *take_picture* would be created to correspond to the two images desired in the model's goals. In the model checking step, SPIN explores all possible interleavings of the *take_picture* activities with all other activities.

The behavior we desired for each activity is that after the activity is created the activity may schedule itself as soon as its constraints are satisfied. This semantics coincides precisely with Promela semantics where each Promela statement is either executable or blocked. As soon as a Promela statement is executable it may be passed and if it is not executable, the process containing the statement blocks until the statement evaluates to true.

The *take_picture* activity proctype is defined on lines 11 through 29 of Figure 6. The guard, or block, on the executability of the *take_picture* activity is on lines 12 through 14. The guard stipulates that either state variable *civa_on* must be set to on (line 12), or there should be no other reservation for the *civa_sv* state variable (line 13). Also, the camera should be on (line 14) and there should be available space in *data_buffer* (line 15) to record the picture.

```
1 unsigned data_buffer : 3 = 4;
2 mtype = {on, off, failed, picture};
3 bool civa = 1; /* atomic resource:
4 1 is available,
5 0 is in use */
6 unsigned count : 3; /* # of memory using
7 activities scheduled */
8 unsigned power : 6 = 32; /* non-depletable
resource with capacity=35000/1094= 32and
min_value=0
9 ** power values scaled to fit in an
unsigned with 6 bits .. i.e. 32 */
```

```

10  mtype civa_sv = off;
11  chan mutex_civa = [2] of {pid};
12  chan plan = [0] of {mtype};
13
14  proctype take_picture() {
15      atomic {(((civa_sv == on) || \
16          empty(mutex_civa)) && \
17          civa && \
18          ((data_buffer-1) >= 0)) ->
19          if
20              :: (civa_sv != on) ->
21                  civa_sv = on;
22                  power = power - 9;
23              :: else
24                  fi;
25          mutex_civa!_pid;
26          data_buffer--;
27          civa = 0; /* camera in use */
28          plan!picture; /* take picture */
29          count++;
30      }
31
32      d_step {
33          civa = 1;
34          mutex_civa??eval(_pid);
35      }
36  }
37
38  proctype civa_off() {
39      unsigned i : 2;
40  start:
41      do
42          :: atomic {
43              ((civa_sv == on) && \
44              empty(mutex_civa)) ->
45              civa_sv = off;
46              power = power + 9;
47          }
48      od
49  }
50
51  proctype server() {
52      /* prints scheduled activity on MSC, pre-
53      serving order */
54      mtype x;
55      do
56          :: atomic {
57              plan?x ->
58              printf("MSC: %e\n", x);
59          }
60      od;
61  }
62  init {
63      atomic {
64
65
66          run take_picture();
67          run take_picture();
68          run civa_off();
69          run server();
70          printf("MSC: %d\n", _nr_pr);
71      }
72  }

```

Figure 6 - Promela model fragment for taking a picture

At the beginning of the model several variables are declared and initialized. On line 1, the `data_buffer` is declared to store a 3 bit value and is initialized to 4. The `mtype` declaration on line 2 causes integer values to be assigned to represent some state names. On line 3, the `civa` camera is declared as a boolean to capture whether the `civa`, an atomic resource, is available or in use. On line 6, `count` is declared to store a 3 bit value and is initialized to 0. The `count` variable will be used by our property as will be described later. On line 8, `power`, measured in watts is a renewable resource and is declared to hold an 6 bit integer with a default value of 32. On line 10, `civa_sv`, that tracks the state of the camera, is declared to store an `mtype` and is initialized to 'off' to

correspond to the default state of the camera. On line 11, `mutex_civa` is declared as a channel, or queue, with a capacity of 2 messages of size `pid`, which corresponds to a byte. `mutex_civa` is used to track reservations of state values, place by activities, for `civa_sv` variable. On line 12 another channel, `plan`, is used to make activity scheduling explicit and easy to see on XSPIN's sequence chart output. The `plan` channel is a special rendezvous channel that has a capacity of 0.

The `take_picture` proctype is defined on lines 14 through 36. `take_picture` is a meta type. Two copies of `take_picture` are created in the initialization step on lines 66 and 67. The first few statements on lines 15 through 18 form the guard for the activity. The guard ensures that activity won't get scheduled until the reservations and resources are available. In the case of the camera, for example, the state variable `civa_sv` tracks the state of the camera, which can be either 'on' or 'off'. The `take_picture` activities' reservation requirement (line 15) on `civa_sv` is that `civa_sv` 'must be' on, which means that the camera must be 'on' at the onset of and during the entire `take_picture` activity. If the camera is not already on, the `take_picture` activity needs to turn the camera on. It can only do so if no other activity has a reservation on the state of the camera (line 16). The `civa` variable must be 1 indicating that the camera is available and not being used by another activity. Finally, before a picture is taken we need to ensure that there is enough room in the data buffer to store the results. This check is made on line 18.

On lines 19 through 27 state variables and resources are modified. The `civa_sv` variable is set to 'on' and `power` is decremented to reflect the power draw of the camera. A reservation on the value of `civa_sv` is created on line 25. The general method for handling reservations will be explained presently. The available capacity of the data buffer is decremented on line 26. The lock on the camera is modified to show that the camera is in use on line 27. On line 28 a message is sent to the rendezvous channel containing the `mtype` 'picture.' This model artifact appears only to improve our ability to interpret the error traces that XSPIN displays as sequence charts. Each scheduled activity appears as a distinct message with ordering preserved.

To track reservations on state variables we use Promela *channels*, which are similar to message queues. Each state variable has a corresponding channel that can hold several messages. For instance the state variable `civa_sv` has an associated channel `mutex_civa` for tracking reservations on its value. When an activity wants to control the value of a state variable it may only do so if it can pass this guard: (1) the value of the state variable is already the value desired by the activity, or (2) there are no reservations on the state variable. The guard for the `take_picture` activity appears on lines 15 through 18. If the value of the state variable is already the value desired by the activity, the activity sends its process id (`pid`) to the state variable's reservation channel. The check of the guard and the message send must be performed in an atomic step to ensure that no other activity obtains a reservation between the guard check and the message send. The `take_picture` activity, for instance, sends `mutex_civa` its `pid` on line 25 after passing the guard expression. The `take_picture` activity now has a reservation on the value of `civa_sv` and no other activity can change the value of `civa_sv` until the `take_picture` activity, and all other activities that have reservations on the value of `civa_sv`, have removed their reservations. A reservation is removed after the activity has completed. For instance, `take_picture`

transitions for state variables are shown in Figure 4. Default states are shaded. All transitions are allowed for the drill location, camera and telecom state variables. For the ovens,

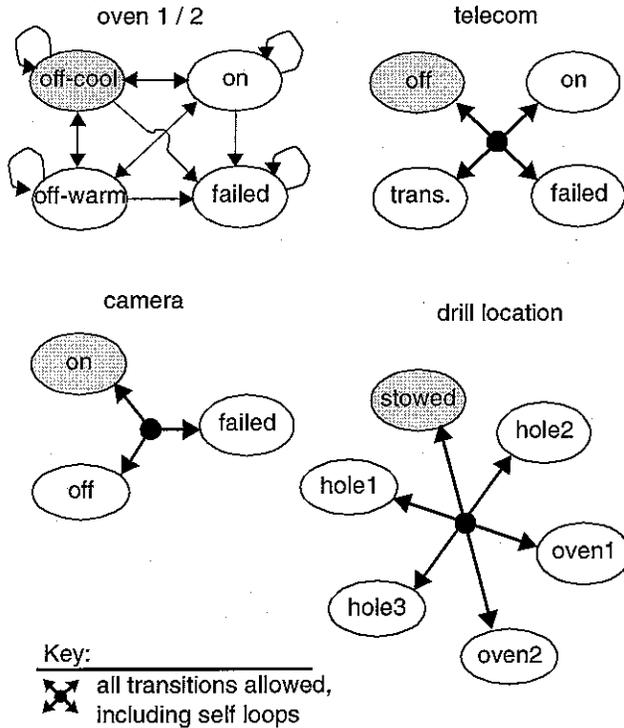


Figure 4 - example CASPER planner domain model state variables and state transitions

all transitions are allowed except for transitions out of the failed state. Once an oven enters the failed state it will stay in the failed state.

A small portion of the planner domain model is shown in Figure 5. The goals shown are two pictures, *take1*, and *take2*. An initial start time for this activity is seven hours after the beginning of the plan. The CASPER planner may move the activity start time but can not delete the activity as indicated by the *no_permissions* variable.

The resources shown in Figure 5 are the camera, the telecom device and the data buffer. The camera and telecom are atomic devices, meaning that only one activity can utilize the resource at a time. The data buffer is a shared resource that can be depleted, and has a capacity and a minimum value.

The *take_picture* activity may start between 10 minutes after the start of the plan and infinity and its duration may be between 1 and 10 minutes. Reservations that are needed in order for the *take_picture* activity to be scheduled are:

- exclusive access to the telecom device (comm)
- exclusive access to the camera (civa)
- use of 5 Mb of the data buffer
- and the camera (civa) must be "on" for the duration of the activity.

Planner Domain Model Expressed in Promela

To express the planner domain model in Promela we first observed that each activity in the CASPER model should

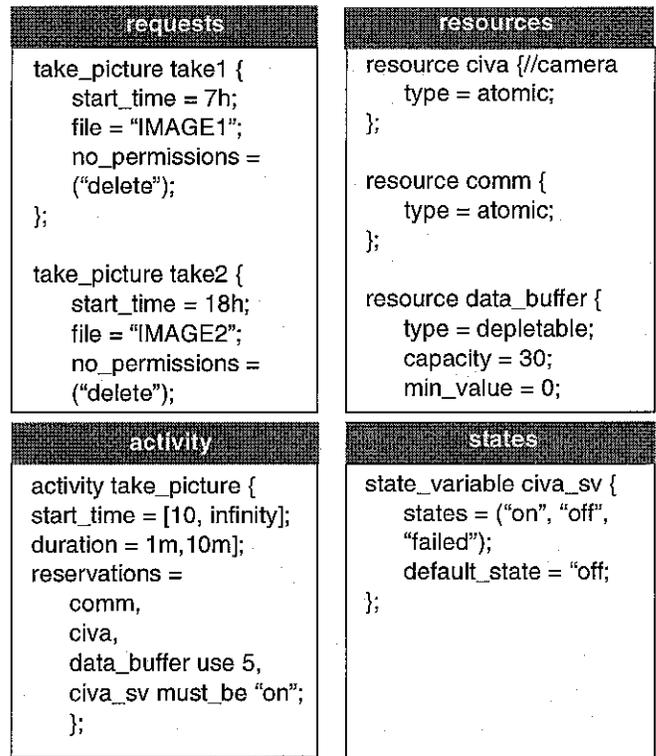


Figure 5 - CASPER/ASPEN model for taking a picture

correspond to an instance of a Promela proctype. A Promela *proctype* is a process type that defines the behavior of a process instance. The *take picture* activity, for instance, would be defined in Promela as a proctype, *take_picture*. In an initialization step, two instances of *take_picture* would be created to correspond to the two images desired in the model's goals. In the model checking step, SPIN explores all possible interleavings of the *take_picture* activities with all other activities.

The behavior we desired for each activity is that after the activity is created the activity may schedule itself as soon as its constraints are satisfied. This semantics coincides precisely with Promela semantics where each Promela statement is either executable or blocked. As soon as a Promela statement is executable it may be passed and if it is not executable, the process containing the statement blocks until the statement evaluates to true.

The *take_picture* activity proctype is defined on lines 11 through 27 of Figure 6. The guard, or block, on the executability of the *take_picture* activity is on lines 13 through 16. The guard stipulates that either state variable *civa_sv* must be set to on (line 13), or there should be no other reservation for the *civa_sv* state variable (line 14). Also, the camera should be on (line 15) and there should be available space in *data_buffer* (line 16) to record the picture.

```

1  unsigned data_buffer : 3 = 4;
2  mtype = {on, off, failed, picture};
3  bool civa = 1; /* atomic resource:
4     1 is available,
5     0 is in use */
6  unsigned count : 3; /* # of memory using
7     activities scheduled */
8  mtype civa_sv = off;
9  ...
10 chan mutex_civa = [2] of {pid};
11
12 proctype take_picture() {

```

```

13  atomic {(((civa_sv == on) || \
14      empty(mutex_civa)) && \
15      civa && \
16      ((data_buffer-1) >= 0)) ->
17  if
18      :: (civa_sv != on) ->
19      civa_sv = on;
20      :: else
21  fi;
22  mutex_civa!_pid;
23  data_buffer--;
24  civa = 0; /* camera in use */
25  plan!picture; /* take picture */
26  count++;
27  }
28
29  d_step {
30  civa = 1;
31  mutex_civa??eval(_pid);
32  }
33  }
34
35  init {
36  atomic {
37  ...
38  run take_picture();
39  run take_picture();
40  ...
41  }
42  }

```

Figure 6 - Promela model fragment for taking a picture

At the beginning of the model several variables are declared and initialized. On line 1, the `data_buffer` is declared to store a 3 bit value and is initialized to 4. The `mttype` declaration on line 2 causes integer values to be assigned to represent some state names. On line 3, the `civa` camera is declared as a boolean to capture whether the `civa`, an atomic resource, is available or in use. On line 6, `count` is declared to store a 3 bit value and is initialized to 0. The `count` variable will be used by our property as will be described later. On line 8, `civa_sv`, that tracks the state of the camera, is declared to store an `mttype` and is initialized to 'off' to correspond to the default state of the camera. On line 10, `mutex_civa` is declared as a channel, or queue, with a capacity of 2 messages of size `pid`, which corresponds to a byte. `mutex_civa` is used to track reservations of state values, place by activities, for `civa_sv` variable.

The `take_picture` proctype is defined on lines 12 through 33. `take_picture` is a meta type. Two copies of `take_picture` are created in the initialization step on lines 38 and 39. The first few statements on lines 13 through 16 form the guard for the activity. The guard ensures that activity won't get scheduled until the reservations and resources are available. In the case of the camera, for example, the state variable `civa_sv` tracks the state of the camera, which can be either 'on' or 'off'. The `take_picture` activities' reservation requirement (line 13) on `civa_sv` is that `civa_sv` 'must be' on, which means that the camera must be 'on' at the onset of and during the entire `take_picture` activity. If the camera is not already on, the `take_picture` activity needs to turn the camera on. It can only do so if no other activity has a reservation on the state of the camera (line 14). The `civa` variable must be 1 (line 15) indicating that the camera is available and not being used by another activity. Finally, before a picture is taken we need to ensure that there is enough room in the data buffer to store the results. This check is made on line 16.

On lines 17 through 24, state variables and resources are modified. The `civa_sv` variable is set to 'on'. A reservation on the value of `civa_sv` is created on line 22. The general method for handling reservations will be explained presently. The

available capacity of the data buffer is decremented on line 23. The lock on the camera is modified to show that the camera is in use on line 24. On line 25 a message is sent to the rendezvous channel containing the `mttype` 'picture.' This model artifact appears only to improve our ability to interpret the error traces that XSPIN displays as sequence charts. Each scheduled activity appears as a distinct message with ordering preserved.

To track reservations on state variables we use Promela *channels*, which are similar to message queues. Each state variable has a corresponding channel that can hold several messages. For instance the state variable `civa_sv` has an associated channel `mutex_civa` for tracking reservations on its value. When an activity wants to control the value of a state variable it may only do so if it can pass this guard: (1) the value of the state variable is already the value desired by the activity, or (2) there are no reservations on the state variable. The guard for the `take_picture` activity appears on lines 13 through 16. If the value of the state variable is already the value desired by the activity, the activity sends its process id (`pid`) to the state variable's reservation channel. The check of the guard and the message send must be performed in an atomic step to ensure that no other activity obtains a reservation between the guard check and the message send. The `take_picture` activity, for instance, sends `mutex_civa` its `pid` on line 22 after passing the guard expression. The `take_picture` activity now has a reservation on the value of `civa_sv` and no other activity can change the value of `civa_sv` until the `take_picture` activity, and all other activities that have reservations on the value of `civa_sv`, have removed their reservations. A reservation is removed after the activity has completed. For instance, `take_picture` removes its reservation of the value of `civa_sv` by removing its `pid` from the `mutex_civa` channel on line 31.

The `count` variable that is incremented on line 26 is used in the correctness property and will be described in more detail later. The guard and the steps that follow it are placed in an *atomic* statement to ensure that no other activity coopts the resources and reservations between the step when reservations are made and when they are claimed.

Lines 29 to 32 are the termination of the activity and return and release of reservations and resources.

The `civa_off` proctype is used to model the turning off of the camera when it has no reservation on its state. The server proctype handles the rendezvous receive of the messages activity send when the activity is scheduled. The `init` is a special type of process that is scheduled as the first step(s) and in our model is used to create instances of the proctypes.

Model Tractability

For our model checking task to be tractable, that is; possible within the constraints of desktop computing power and reasonable response time, we employed several modeling and abstraction techniques. We abstracted the timeline to the minimum number of timepoints needed to check the property of interest. As a result, the check we performed was more robust, in a sense, because it checked all plans, not just those that fit on particular timeline. But the increase in robustness comes with a potential penalty; reports of false positives. The false positives would be plans, flagged as errors, that would not fit on the actual timeline. We did not experience any false positives, but if we had, they could have been eliminated with a simple post-processing check

removes its reservation of the value of `civa_sv` by removing its `pid` from the `mutex_civa` channel on line 34.

The count variable that is incremented on line 29 is used in the correctness property and will be described in more detail later. The guard and the steps that follow it are placed in an *atomic* statement to ensure that no other activity coopts the resources and reservations between the step when reservations are made and when they are claimed.

Lines 32 to 35 are the termination of the activity and return and release of reservations and resources.

The `civa_off` proctype is used to model the turning off of the camera when it has no reservation on its state. The server proctype handles the rendezvous receive of the messages activity send when the activity is scheduled. The `init` is a special type of process that is scheduled as the first step(s) and in our model is used to create instances of the proctypes.

Model Tractability

For our model checking task to be tractable, that is; possible within the constraints of desktop computing power and reasonable response time, we employed several modeling and abstraction techniques. We abstracted the timeline to the minimum number of timepoints needed to check the property of interest. As a result, the check we performed was more robust, in a sense, because it checked all plans, not just those that fit on particular timeline. But the increase in robustness comes with a potential penalty; reports of false positives. The false positives would be plans, flagged as errors, that would not fit on the actual timeline. We did not experience any false positives, but if we had, they could have been eliminated with a simple post-processing check

Another method we used to avoid the state space explosion problem was to scale integer variables to fit in a byte or several bits. We used this technique for resources such as power and memory. We also used atomic sequences as much as possible. Atomic sequences are sequences of execution steps that can not be interleaved with execution steps from

other processes. Use of these sequences reduces the number of states that SPIN needs to explore.

Expressing the Property of Interest

The test concern for the example planner domain model was the question of whether it might permit the APS to select undesirable plans. There are two types of undesirable plans: plans that imperil the safety of the mission, and plans that waste resources resulting in a reduction in science return. Although this technique can be applied to check for both types of undesirable plans, we used it to check the latter type. The concern we addressed was that the AI input model would permit the APS to select plans that would waste resources and therefore not meet the mission's science goals.

It is much easier to specify how a system should work rather than all the ways in which a system can break. Similarly, we wanted to specify the characteristics of a desirable plan rather than try to enumerate all the undesirable plans since the AI input model is so complex that we would inevitably miss some undesirable plans. Fortunately, the model checking paradigm explicitly supports specifying the *desired* properties of a system and letting the model checker do all the work to find exceptions to the desired properties.

For the example planner domain model, a desirable plan was one that achieved all the goals: 2 images and 3 samples. An example of one such desirable plan that was produced by SPIN in a random simulation run is shown in Figure 7. Time progresses to the right. The occurrences of the activities, sample and image, that satisfy the goals are shown in green. Uplink and compress data are permitted activities that do not directly satisfy the goals. Uplink transmits data to the orbiter part of the mission and compress data is used to free up memory so that additional data products can be stored. The state variables `oven1`, `oven2`, `camera` and `drill location` and their values over time are shown beneath the goals. The values of resources power use and memory use are shown at the bottom of the timeline. This presentation of a plan closely resembles the visual output for plans generated by CASPER.

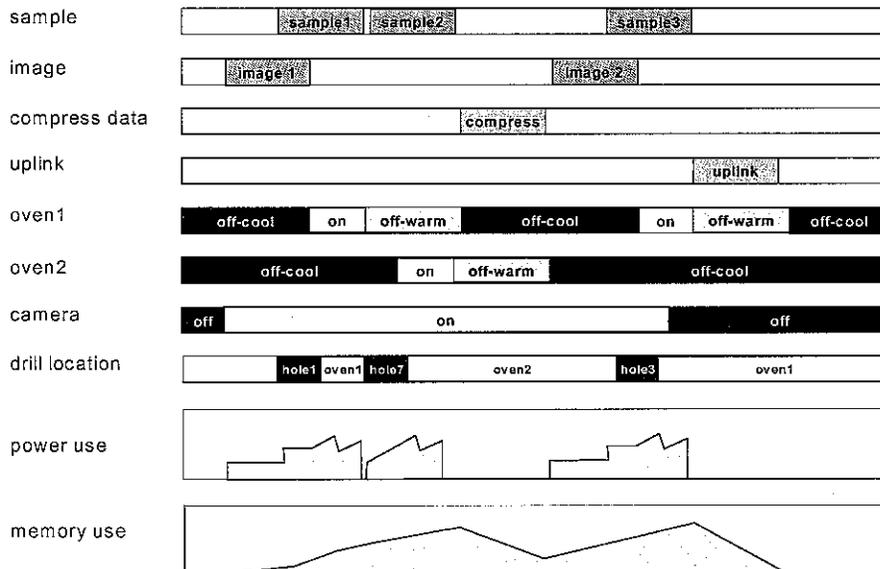


Figure 7 - Example desirable plan for the example planner domain model

Although the simulated desirable plan shows that at least one desirable plan exists in the AI input model, we need to show that all possible plans are good. To do this we first defined the desired property formally using the Timeline Editor [8]. The Timeline Editor is a visual tool for expressing properties and automatically converting properties to SPIN never claims. A never claim is simply something that should never happen. The desirable plan property, expressed as a timeline, is shown on the top-left of Figure 8. Time progresses from left to right. The vertical bars labelled '0', '1', and '2' are

marks or locations where interesting events can be placed. The '0' mark indicates the beginning of an execution. In between two marks zero or more execution steps may occur. To specify what should happen between marks we may use constraints, which would appear as horizontal lines between marks. We do not need to use constraints to express this particular property.

The first event on the timeline is *both ovens in default state*.

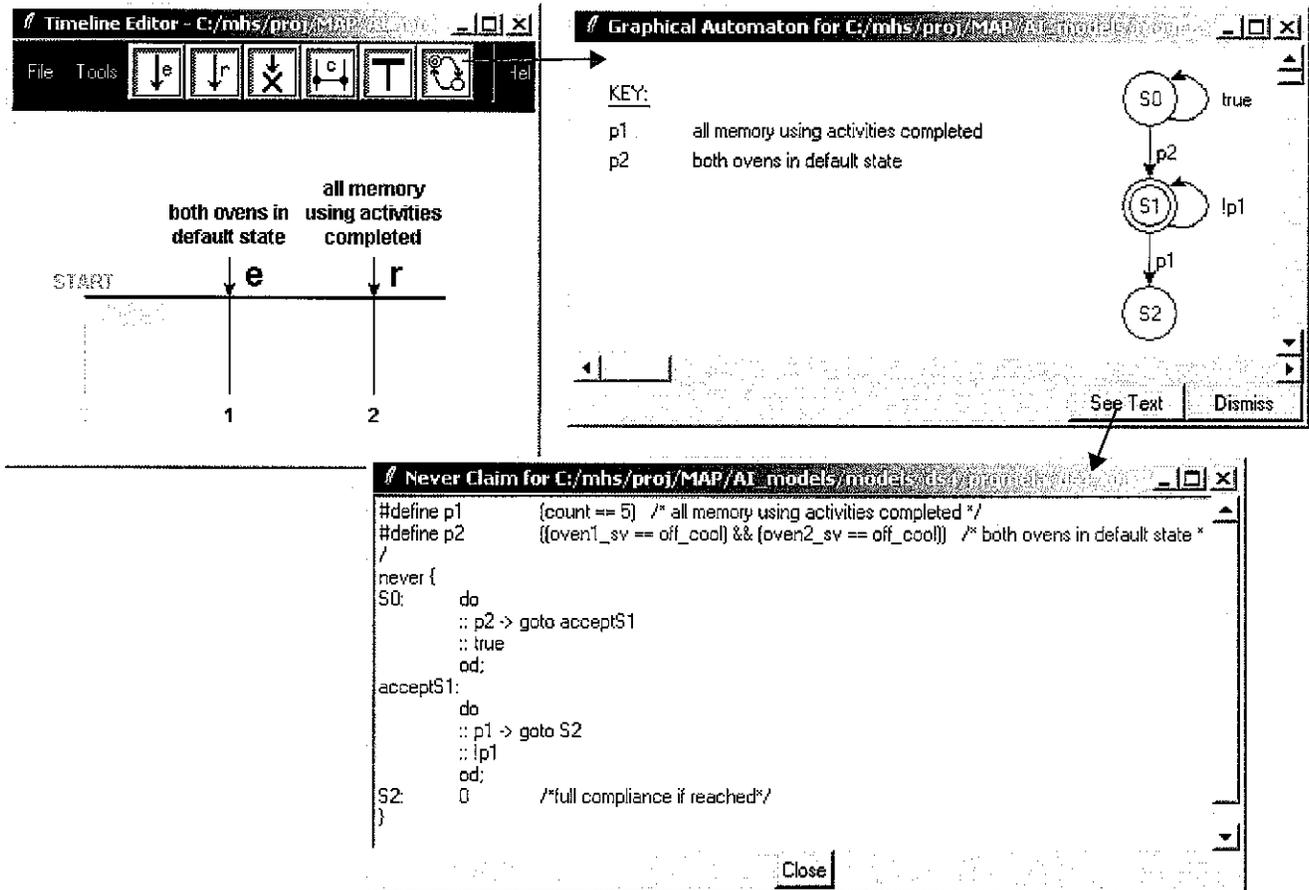


Figure 8 - 'desirable plan' Property for the example planner domain model expressed using the Timeline Editor

This event is an **e** event, denoting a regular event that is neither required or an error. By including the **e** event on the timeline we have specified that we are interested in executions where sometime after the beginning of the execution both ovens are in their default state. The second event, *all memory using activities completed*, is an **r** event, meaning that it is required. If the first event occurs but the second event never occurs this is an error. The two events shown are expressed informally and formally. The informal representation is the English prose that appears as an event label. The formal definition of the event expresses the event label in terms of the values of variables in the model. For instance, the formal definition of *both ovens in default state* is:

$((oven1_sv == off_cool) \ \&\& \ (oven2_sv == off_cool))$

where *oven1_sv* and *oven2_sv* are both state variables in the model, and *off_cool* is the oven state variable's default value.

The formal definition of *all memory using activities completed* is:

$(count == 5)$

The count variable is not part of the input model but a global variable that we added to enable the check. As shown in the *take_picture* Promela proctype in Figure 6, the count variable is incremented when the *take_picture* activity occurs. An increment to count is also made when *experiment* occurs. *Experiment* is a sub activity within the *sample* activity. When count is equal to 5 the property is satisfied. If SPIN can find an execution where count does not eventually reach 5 it will report it as an error.

The graphical automaton version of the property on the right-top of Figure 8 shows the property in a form similar to a finite state machine. Symbols p1 and p2 are assigned to represent the two events of interest. When execution begins we are in the start state, S0. At each execution step in the model we must take a transition. The true self loop on S0 can

always be taken. When p2 becomes true, corresponding to both ovens in default state, we can transition to S1 or remain in S0 by taking the true self loop. Thus we check both the first occurrence of p2 and all subsequent occurrences of p2. S1 is a special state, called an *accepting state* and is denoted by the double circle. If we can return to accepting state S1 infinitely often in an execution, that execution is an error. Hence, if for the remainder of the execution, we can return to S1 by taking the p2 transition corresponding to *all memory using activities not completed*, then SPIN will report that execution as an error. If p1 occurs, corresponding to *all memory using activities completed*, we take transition p1 to S2 and the execution under consideration is not an error. The SPIN never claim version of the property that is generated by the Timeline Editor is shown at the bottom of Figure 8. The never claim can be appended to the model or saved to a file and included by reference from XSPIN (SPIN's graphical user interface).

VERIFICATION RESULTS

We used two of SPIN's numerous strategies for verification of large models; *partial order reduction* to reduce the number of system states that must be searched, and *collapse compression* to reduce the amount of memory needed to store each state [2]. In the verification run with SPIN, an error (undesirable plan) was reported within 1 second after checking only 43 states. The undesirable plan, which was reported as a sequence diagram by SPIN, is depicted in the output form used by CASPER. The undesirable plan found by SPIN omits the third sample activity and therefore contains only four of the required five goal activities. In the

undesirable plan the second imaging activity could not be scheduled because all the activities that clear out memory were scheduled at the beginning of the plan when no memory had yet been used. Both imaging and sampling activities use memory and there is only enough memory to store the results of four instances of these activity types. Sometime after the first memory activity but before the fourth memory using activity, either data compression or uplinking should take place to make room in memory to store the results of the fifth memory using activity.

To fix the model we observed that the undesirable plan occurred because data compression and uplinking were allowed to occur when memory was empty. The fix we chose was to add a guard to prevent data compression from occurring when memory is empty. In Figure 9, this guard, omitted from the original model containing the undesirable plan, is shown in bold on line 4. This guard has been added to the Promela version of the `compress_data` proctype so that the process will block until the memory (`data_buffer`) is non-empty. In this case `data_buffer`, that tracks unused capacity, is 4 when memory is empty and is 0 when memory is full.

```

1  proctype compress_data() {
2  atomic
3  {
4  (data_buffer < 4) ->
5  data_buffer = data_buffer + 1;
6  plan!compress;
7  }
8  }

```

Figure 9 - `compress_data` activity specification in Promela

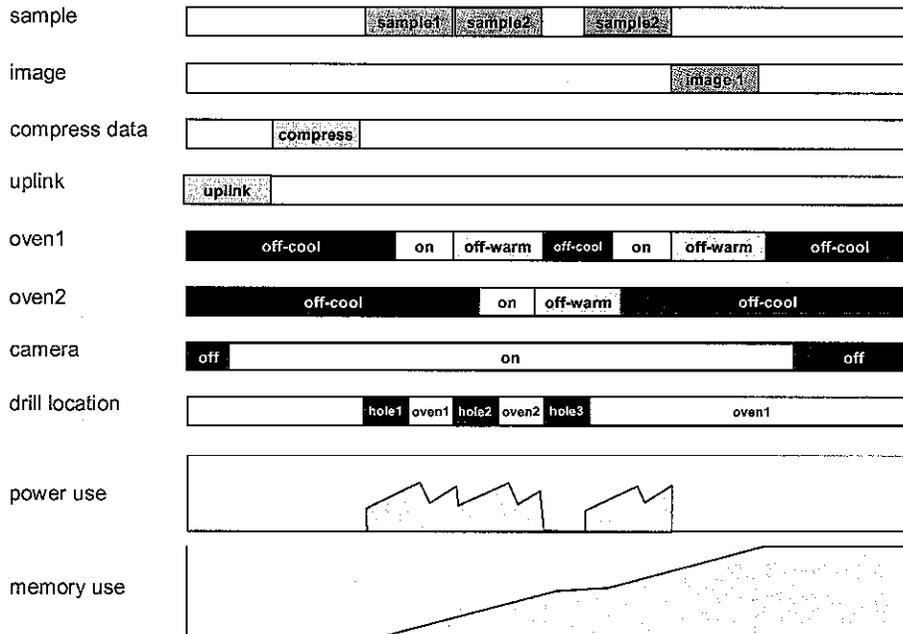


Figure 10 - The undesirable plan found by SPIN shown in CASPER-like output format

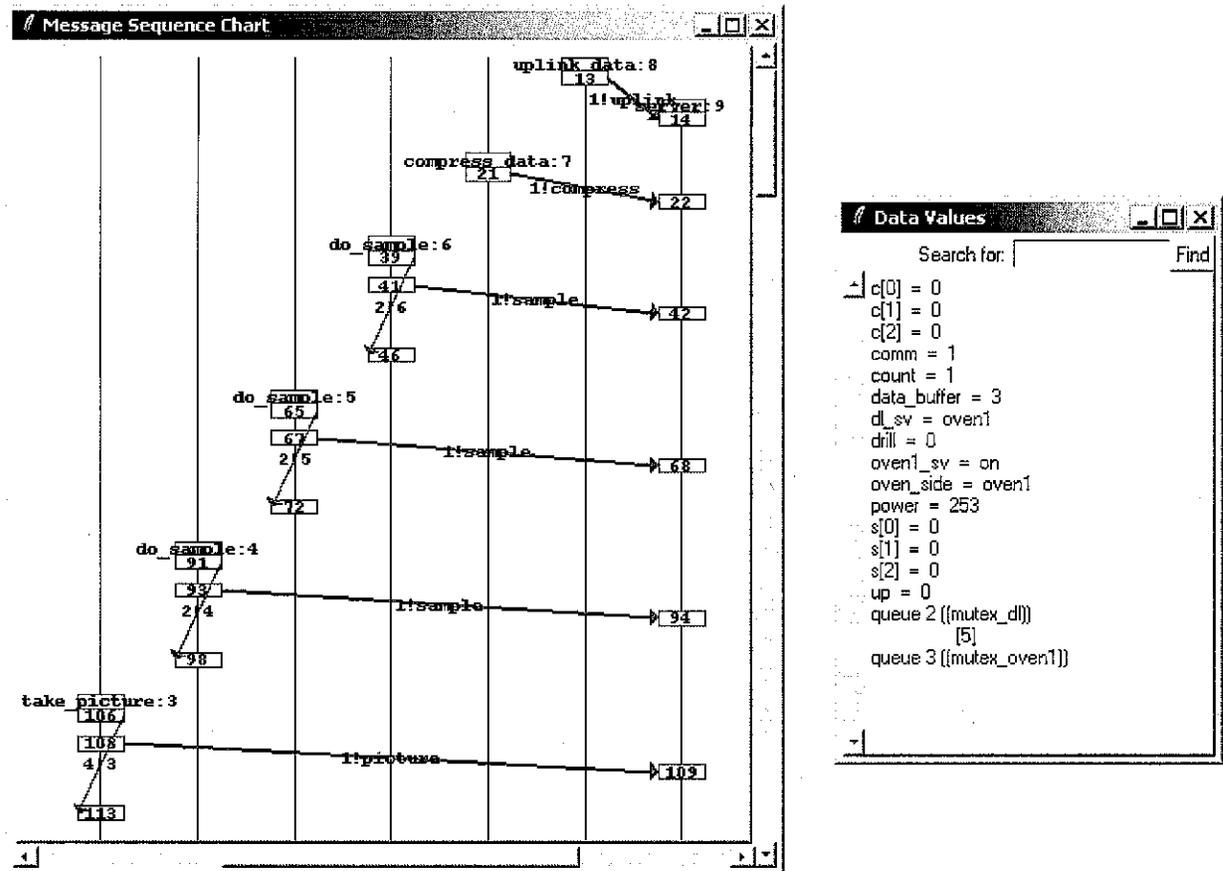


Figure 11 - The bad plan found by SPIN shown as an XSPIN Message Sequence Chart and example Data Values during second Sample activity

We repeated the model checking step on the repaired model and SPIN did an exhaustive search, checking 670 million states and reported no errors. Hence, we can conclude that the input model with the added constraint would not allow the APS to select a undesirable plan.

input models of the Earth Observer 1 mission that employs the CASPER planner and has been in autonomous operation since May 2004. We will also explore possible tool support needed to incorporate SPIN verification of AI input models into the existing process of developing AI input models.

CONCLUSIONS

Using an example planner domain model we demonstrated the ability of the SPIN model checker to verify planner domain models. Specifically, we converted the planner domain model to Promela, the language of the SPIN model checker, formulated a correctness property for desirable plans, and asked SPIN to find and report undesirable plans. SPIN quickly found and reported a undesirable plan that arose due to a missing constraint in the AI input model. We analyzed the error report then added a constraint and repeated the check. In an exhaustive search of the model, SPIN found no additional undesirable plans in the planner domain model.

Testing AI input models using SPIN can dramatically increase the confidence that AI input models are safe to fly. In this case in our verification using SPIN we were able to check millions of plans, replacing a sampling based test technique that checks in the order of 100 plans. In the next phase of our work we plan to use this technique to check AI

ACKNOWLEDGEMENTS

This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the National Aeronautics and Space Administration (NASA) Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP). We are grateful to Kenneth McGill, Martha Wetherholt, and Timothy Menzies of the NASA Independent Validation and Verification center for their insightful feedback and guidance.

REFERENCES

[1] Gerard Holzmann, "The Model Checker Spin," IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.

- [2] Gerard Holzmann, "The Spin Model Checker: Primer and Reference Manual," 2003, Addison-Wesley, ISBN 0-321-22862-6, 608 pgs.
- [3] L. Khatib, N. Muscettola, K. Havelund "Verification of Plan Models using UPPAAL," First Goddard Workshop on Formal Approaches to Agent-Based Systems. NASA's Goddard Space Center, Maryland. March 2000.
- [4] K. Havelund, M. Lowry, J. Penix, "Formal Analysis of a Space Craft Controller using SPIN," IEEE Transactions on Software Engineering, Vol. 27, No. 8, August, 2001. Originally appeared in proceedings of the 4th SPIN workshop, Paris, France. November 1999.
- [5] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, J. L. White, "Formal Analysis of the Remote Agent - Before and After Flight," The Fifth NASA Langley Formal Methods Workshop, Virginia. June 2000.
- [6] J. Penix, C. Pecheur, K. Havelund, "Using Model Checking to Validate AI Planner Domain Models," 23 Annual NASA Goddard Software Engineering Workshop, Goddard, Maryland, Dec 1998.
- [7] B. Cichy, S. Chien, S. Schaffer, D. Tran, G. Rabideau, R. Sherwood, "Validating the Autonomous EO-1 Science Agent," International Workshop on Planning and Scheduling for Space (IWSS 2004). Darmstadt, Germany. June 2004.
- [8] M. Smith, G. Holzmann and K. Ettessami, "Events and Constraints: a graphical editor for capturing logic properties of programs," 5th International Symposium on Requirements Engineering, pp 14-22, Toronto, Canada. August 2001.
- [9] S. Chien, R. Knight, A. Stechert, R. Sherwood, G. Rabideau, "Using Iterative Repair to Improve Responsiveness of Planning and Scheduling," International Conference on Artificial Intelligence Planning Systems (AIPS 2000). Breckenridge, CO. April 2000.
- [10] Alex Fukunaga, Gregg Rabideau, Steve Chien, "ASPEN: An Application Framework for Automated Planning and Scheduling of Spacecraft Control and Operations," Proceedings of International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS97), Tokyo, Japan, 1997, pp. 181-187.
- [11] <http://www.spinroot.com>
- [12] B. Smith, R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, S. Chien, A. Fukunaga, "Representing Spacecraft mission planning knowledge in Aspen," AIPS-98 Workshop on Knowledge Engineering and Acquisition

for Planning, June 1998. Workshop notes published as AAI Technical Report WS-98-03.

BIOGRAPHY



Margaret H. Smith is a Senior engineer at JPL in the Systems Engineering and Technology Infusion group. Before joining JPL Margaret spent five years at Bell Labs Research applying model checking tools to commercial software products, such as Lucent Technologies' Pathstar Access Server. She has developed, with Gerard Holzmann, several software analysis tools for Lucent Technologies, as well as a visual tool for capturing formal correctness properties for use in model checking. Margaret has over ten years experience as a systems engineer at AT&T Bell Labs where she led requirements teams and applications of new requirements tools. Margaret holds four patents. She received a BSE and MSE from University of Michigan, School of Engineering in Ann Arbor, Michigan.



Gerard J. Holzmann is a Principal researcher at JPL, and the primary developer of the SPIN model checker. He has participated in the development of several successful requirements capture and analysis tools. Jointly with Margaret Smith, Holzmann developed the verification technology that was used to exhaustively verify the call processing software for the Pathstar Access Server product at Lucent Technologies with a specially developed extended version of the SPIN model checker. In 2001 Dr. Holzmann was the recipient of both the prestigious ACM System Software Award and the SIGSOFT Outstanding Researcher Award, for the development of the SPIN system. He holds six patents, and has written four books. Dr. Holzmann received his PhD from Delft University, the Netherlands.



Benjamin Smith is a senior member of the Exploration Systems Autonomy section at JPL. His research interests are in the development and verification of autonomous systems. He was the deputy project element lead for the Remote Agent Experiment, which autonomously operated the NASA Deep Space 1 spacecraft, and he led the development and verification of the autonomous mission planning system for the NASA Mod-

ified Antarctic Mapping Mission. He received his Ph.D. in 1995 from the University of Southern California.



Gordon Cucullu, III is a Senior engineer at JPL in the Systems Engineering and Technology Infusion group. Since 1995, he has worked at JPL as a flight system engineer on various flight projects: the Autonomous Rendezvous Experiment, the Mars 2001 Lander instruments, the EOS Microwave Limb Sounder, and as a mechanical integration engineer on Cassini. He has a MS in aerospace engineering from the University of Southern California, and a BSME from the University of New Orleans (Tau Beta Pi, Pi Tau Sigma).