Formal Analysis of a Space Craft Controller using SPIN

Klaus Havelund, *Mike Lowry* and *John Penix*

NASA Ames Research Center
Moffett Field, California, USA
Email: {havelund,lowry,jpenix}@ptolemy.arc.nasa.gov
URL: http://ic-www.arc.nasa.gov/ic/projects/amphion

Abstract

This paper documents an application of the finite state model checker SPIN *to formally verify a multi–threaded plan execution programming language. The plan execution language is one component of NASA's New Millennium Remote Agent, an artificial intelligence based spacecraft control system architecture that is scheduled to launch in October of 1998 as part of the* DEEP SPACE 1 *mission to Mars. The language is concretely named* ESL *(Executive Support Language) and is basically a language designed to support the construction of reactive control mechanisms for autonomous robots and space crafts. It offers advanced control constructs for managing interacting parallel goal-and-event driven processes, and is currently implemented as an extension to a multi-threaded* COMMON LISP. *A total of* 5 *errors were in fact identified,* 4 *of which were important. This is regarded as a very successful result. According to the Remote Agent programming team the effort has had a major impact, locating errors that would probably not have been located otherwise and identifying a major design flaw. The work additionally motivated the introduction of procedural abstraction in terms of inline procedures in* SPIN.

1 Introduction

SPIN [7] is a verification system that supports the design and verification of finite state asynchronous process systems. Programs are formulated in the PROMELA programming language, which is quite similar to an ordinary programming language, except for certain non-deterministic specification oriented constructs. Processes communicate either via shared variables or via message passing through buffered channels. Properties to be verified are stated in the linear temporal logic LTL. The SPIN *model checker* can automatically determine whether a program satisfies a property, and in case the property does not hold, an error trace is generated.

This paper documents an application of SPIN to formally verify a multi–threaded plan execution programming language (a library really). The plan execution language is one component of NASA's New Millennium Remote Agent (RA) [9], an artificial intelligence based spacecraft control system architecture that is scheduled to launch in October of 1998 as part of the DEEP SPACE 1 mission to Mars. The language is concretely named ESL (Executive Support Language) and is basically a language designed to support the construction of reactive control mechanisms for autonomous robots and space crafts. It offers advanced control constructs for managing interacting parallel goal-and-event driven processes, and is currently implemented as an extension to a multi-threaded COMMON LISP.

ESL is used to program the RA *Executive*, a subcomponent of the RA, responsible for executing jobs safely on board. To analyze a *language* like ESL, which is generic in its nature, we have set up a special situation called the *model* – really a small example RA Executive – with a fixed number of tasks all using constructs of the language, and then observed whether this *model* satisfies various desired properties. The effort has consisted of hand translating parts of the LISP code for ESL into the PROMELA language of SPIN. A total of 5 errors have in fact been identified, 4 of which are important. This is regarded as a very successful result. According to the RA programming team the effort has had a major impact, locating errors that would probably not have been located otherwise and identifying a major design flaw not yet resolved at the time of writing our first report [4].

Section 2 contains an informal description of the RA Executive, while section 3 describes its formalization in PROMELA. Section 4 presents the verification results by first stating the properties to be verified, and then by describing the errors found by applying the model checker to the model and these properties. Each error is described by an error trace leading from the initial system state to a state that breaks the particular property being verified. Finally, sections 5 and 6 contain the RA programming team's evaluation of the project, and our own conclusions respectively. Our own conclusions concern issues such as PROMELA's capabilities seen as a specification notation, and tool support for model building.

Acknowledgments

We would like to thank Erann Gat, who has programmed ESL, for his useful responses to our error reports, and for providing the basic contents of the evaluation in section 5. When we occasionally refer to the *RA programming team*'s response to our work, it is his response that is referred to. We also want to thank Ron Keesing and Barney Pell who are members of the RA programming team. Their comments were more related to explaining the model and suggesting properties to be verified. Finally, we have had an ongoing useful email conversation with SPIN's designer, Gerard Holzmann. A result of this communication was the introduction of "inline" procedures in PROMELA.

2 Informal Description of the RA Executive

In this section, we give an informal description of the RA Executive. After an overview follows a description of the data types and the processes of the system.

2.1 Overview

The RA Executive, Figure 1, is designed to support safe execution of software controlled *tasks* on board the space craft. A task may for example be to run and survey a camera. A task often requires specific *properties* to hold in order to execute correctly. For example, the camera–surveying task may require the camera to be turned on throughout task execution. When a task is started (dynamically), it first tries to *achieve* the properties on which it depends; where after it starts performing its main function. The camera–surveying task will for example try to turn on the camera before running the camera. Properties may, however, be unexpectedly broken (e.g. camera may be turned off) and tasks depending on such broken properties must then be *interrupted*.

To simplify the programming of the individual tasks, the RA Executive models the spacecraft devices in terms of the various properties that they may have, and stores these in a *database*. The executive provides mechanisms for both *achieving* and *maintaining* these properties, and

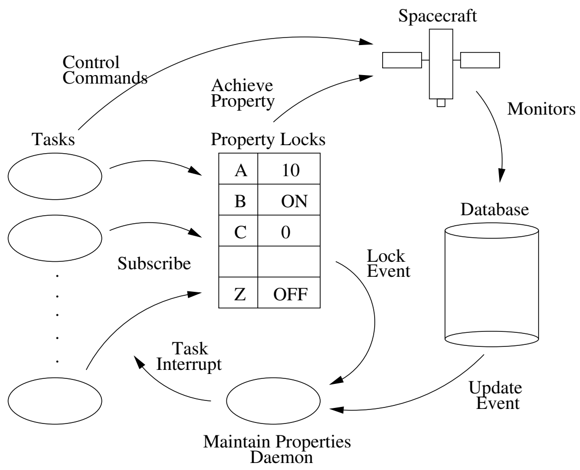

Figure 1. Remote Agent Executive

uses *locks* to prevent tasks with *incompatible* property requests from executing concurrently. Executing concurrently with the tasks is a *"maintain properties" daemon* that monitors the database representing the state of the spacecraft. If there is an *inconsistency* between the database and the locks – meaning that a locked property no longer holds in the database – the daemon suspends all tasks *subscribed* to the property while some action is taken to re-achieve the property. The daemon is normally inactive unless certain *events* happen, such as a change of the database or the lock table.

The Executive permits various *achieve methods* to be associated with a property. Then, when a task makes a request for a property to be achieved, the Executive calls the achieve method that is appropriate for the current state of the system. This aspect will, however, not be subjected to verification, and hence we shall downplay it. Instead, we shall regard the tasks as being able to achieve properties directly themselves.

2.2 Data Types

The Properties

A property describes some state of the space craft. In terms of programming jargon, it basically states that some variable, called the *property name*, has some value, called the *property value*. For example, the following is a property:

CAMERA *is* ON

It states that the property name CAMERA has the property value ON. Hence, a property p is a pairing of a property name pn and a property value pv: $p = (pn, pv)$. The property above can be written as (CAMERA,ON).

The Database

The state of the space craft is constantly monitored, and stored in a database. Since the current state can be regarded as the set of properties that currently hold, the database is basically a set of such properties.

The Property Lock Table

As mentioned, a task can lock a property to prevent other tasks requiring incompatible properties from executing concurrently. Two properties $p_1 = (pn, pv_1)$ and $p_2 = (pn, pv_2)$ are incompatible, if they have the same property name (pn) but different property values ($pv_1 \neq pv_2$). The property lock table contains those properties that have been locked. In addition, it contains information for each property about which tasks subscribe to it (rely on it) and whether it has been achieved or not. That is, the property lock table can be regarded as a set of locks, where a lock is a triple of the form: $(p, subscribers, achieved)$[1].

If there is an inconsistency between the database and the locks, the daemon suspends all tasks subscribed to the property. An inconsistency occurs if the lock table contains a lock $l = (p, sub, true)$ with a property p that has been achieved (achieved field is $true$) but is not in the database.

The Events

Whenever the lock table or the database is changed, this is signaled to the daemon so that it can examine the renewed system state. In general, application tasks may also wait for such events to happen as described below. For this purpose, event lists are introduced, one for each instance of event: SNARF_EVENT (representing a change of the lock table – *to snarf* is implementers jargon for *to lock*) and MEMORY_EVENT (representing a change of the database). Any process (task or daemon) wanting to wait for an event to happen calls a *wait* procedure, which hooks up the process to the corresponding list. Whenever changes happen to these data structures, the corresponding event lists are signaled, via the *signal* procedure, resulting in the waiting processes being restarted - for example the daemon.

2.3 Processes

The Tasks

Before a task executes its main job, it will try to achieve the properties that the execution depends on. First, however, it will lock the properties in the lock table – this activity is called *snarfing* by implementers. The snarfing of a property can, however, only succeed if it is compatible with the existing locks, and in case it's not, the task is aborted. If there are not conflicting locks, the task will create the lock, if it is not already there. Note that some other task may have locked the exact same property already, and this is not defined as a conflict. If it succeeds, the task also puts itself into the subscribers list of the lock, indicating that now this task depends on this property.

The creator of a lock is called the *owner*, in contrast to tasks that subscribe later to the same property. The owner is responsible for achieving the property, resulting in the database being updated. Upon successful achievement, the achieved field in the lock is set to $true$. If the achievement fails, the task is aborted. Other tasks that subscribe later than the owner must wait for the owner to achieve the property. This is done by simply waiting for a MEMORY_EVENT which successfully achieves the property. Hence, the *wait* procedure takes a property as argument in addition to the event to be waited for.

Once a task has first snarfed and then achieved its required properties, it executes its main job, relying on the properties to be maintained throughout job execution.

Before a task terminates, it releases its locks. That is, it removes itself from the subscribers list, and in case this then becomes empty (no other subscribers), it removes the lock completely. In case there are other subscribers, the lock must of course be maintained.

The "Maintain Properties" Daemon

The purpose of this daemon is to guarantee that achieved properties are maintained while subscribing tasks are executing. A once achieved property in the property lock table is said to be *maintained* as long as it is contained in the database (and hence is a property of the space craft). Hence, from the perspective of a task, the maintained properties are invariants while the task is executing – and the task is aborted by the daemon if not.

The daemon is normally in "sleeping" mode, waiting for an event that modifies the database (MEMORY_EVENT) or the property lock table (SNARF_EVENT). This is implemented by letting the daemon wait in the corresponding event lists. Once started, it examines all locks in the property lock table, and for each lock where the achieved field is $true$, it checks whether the property is contained in the database. If the property is not in the database all tasks in the lock's subscribers list are interrupted, and a recovering procedure is initiated which will re-achieve the property. After having examined all locks, the daemon goes into sleep again by waiting for another MEMORY_EVENT or SNARF_EVENT.

[1] The figure only shows the properties of the lock table.

3 Formalization in PROMELA

In this section we present the PROMELA model of the RA Executive. The basic data type of LISP is that of lists, and we therefore begin our exposition by outlining how we have modeled lists in PROMELA. Then the presentation is divided into subsections corresponding to the following topics: the state space (constants, types and global variables), the operations on events, the tasks, the daemon, the environment that may introduce violations, and finally a section explaining how the system state is initialized.

The LISP program that we want to model in PROMELA is highly structured using procedural abstraction, and hence is divided into a collection of relatively small-sized procedures and functions. We have tried to maintain the same level of structuring, using PROMELA's inline and macro concept. Note furthermore that all communication between processes basically takes place via *shared variables*, since this is how the LISP implementation works. Channels are used to represent lists though, as will be described in the next section.

3.1 Modeling Lists

The fundamental data type in LISP is that of lists. Lists are used heavily in the program, and hence we have tried to find a convenient way to represent them in PROMELA. One solution is to define an abstract data type, implementing lists as arrays and defining the classical operations like *add an element*, *remove an element*, etc. as macros (or inlines in the newest version of SPIN). We didn't do this, mainly due to an early attempt to avoid macros since they are not well integrated into SPIN; they do for example not support local variables very well.

As an experiment (rather than a choice of best solution) we decided early to model lists as channels. Channels have some of the same properties as lists: one can easily *add* elements, and *remove* them (following the FIFO–principle though). In addition, channels make some operations that we need easy. That is, questions like *"does list l contain element x?"*, and operations like *"remove element x from the list l – no matter where it is in the list"*. We shall shortly describe the technique.

First, with the macro definition "`#define list chan`" we define a new symbol `list` to stand for the symbol `chan`, which is the PROMELA keyword for declaring channels. This definition makes it possible to declare a "list variable" as follows:

```
list numbers = [5] of {int}
```

The "list variable" `numbers` is intended to contain lists with a length smaller than or equal to 5. A number of operations are now defined upon lists, which we shall only give the signatures for, see Figure 2.

```
inline append(e,l) {...};
inline remove(e,l) {...};
inline copy(l1,l2) {...};
inline next(l,x)   {...};
```

Figure 2. Signatures for list operations

Informally, the procedures and functions do the following[2]. The procedure `append` appends an element to the front of a list; `remove` removes a particular element (assuming it is there); `copy` copies one list (`l1`) into another (`l2`); `next` removes the first element inserted (FIFO principle) and stores this in the result variable `x` (assuming the list is not empty). Suppose we have the following declarations:

```
int x;
list numbers = [5] of {int};
list temp = [5] of {int};
```

Then Figure 3 illustrates the use of the list operations, and their effect on the variables `x`, `numbers` and `temp` (only changes are shown). All statements execute, hence boolean valued expressions evaluate to $true$.

	x	numbers	temp
	0	[]	[]
append(1,numbers);		[1]	
append(2,numbers);		[2,1]	
append(3,numbers);		[3,2,1]	
next(numbers,x)	1	[3,2]	
x == 1;			
copy(numbers, temp);			[3,2]
remove(3,temp);			[2]
next(temp,x);	2		[]
x == 2			

Figure 3. Examples of list operations

[2]Somewhat more formally, the procedures perform the following channel operations: `append(e,l)` does `l!e`; `remove(e,l)` does `l??e`; `copy(l1,l2)` does combinations of `l1?x` and `l2!x`; and `next(l,x)` does `l?x`. Note however, that some of these PROMELA channel operators do not allow variables as arguments, only constants, hence the implementations of these procedures are sometimes more elaborated.

3.2 The State Space

Three constants define the bounds of the system, Figure 4. That is, they define the size of the state space, an important factor for obtaining efficient model checking.

```
#define NO_PROPS  2
#define NO_EVENTS 2
#define NO_TASKS  3
```

Figure 4. The constants

The constant `NO_PROPS` defines the number of property names, and hence the size of the property lock table and database, which each have an entry for each property name. We shall work with two property names: 0 and 1. The constant `NO_EVENTS` defines the number of events, 2 in our case: `MEMORY_EVENT` and `SNARF_EVENT` as will be formalized below. Finally, the constant `NO_TASKS` defines the number of tasks in the system, including the daemon. This number is set to 3 corresponding to a daemon and two application tasks.

A number of types are defined, see Figure 5[3]. The type `EventId` is an enumerated type defining the two forms of events. `TaskId` is the type of task identifiers. Note, that there are 3 tasks (`NO_TASKS` $= 3$): the daemon, which is given identity 0 and two application tasks, given identity 1 and 2 respectively.

The type `Property_Name` contains the property names, of which there are two (`NO_PROPS` $= 2$): 0 and 1. Correspondingly, the type `Property_Value` contains the property values. There is no constant defining the maximal number of property values, since this bound is not needed for declaring the state space (beyond declaring it as a `byte`). Finally, a `Property` is then defined as a record containing two entries: a property name and a property value.

Now, as we shall see, the property lock table will be modeled as a mapping from property names to locks in the type `Lock`[4]. Hence each property name is mapped to a record containing the following three fields: the property value it is supposed to have; the list of tasks subscribing to the lock; and finally, a flag indicating whether it has been achieved or not.

[3]Note that PROMELA does not have type equations nor enumerated types. Hence, a type equation of the form `type T = ty` stands for `#define T ty` and an enumerated type of the form `type T = {A,B,C}` stands for `#define T byte`, followed by `#define A 0`, `#define B 1` and `#define C 2`.

[4]In the LISP program a property lock table is represented as a list, but we have found the mapping representation to be more convenient from a modeling point of view; although thereby we risk to overlook potential errors.

```
type
  EventId = {MEMORY_EVENT,SNARF_EVENT};
  TaskId = byte;

type
  Property_Name = byte;
  Property_Value = byte;

typedef Property{
  Property_Name name;
  Property_Value value};

typedef Lock{
  Property_Value value;
  list sub = [NO_TASKS] of {TaskId};
  bool achieved};

typedef Event{
  byte count;
  list pending_tasks = [NO_TASKS] of {TaskId}};

typedef Task{
  State state;
  list waiting_for = [NO_EVENTS] of {EventId};
  Property prop};

type
  State = {SUSPENDED,RUNNING,
           ABORTED,TERMINATED};
```

Figure 5. Types

Each event (`MEMORY_EVENT` and `SNARF_EVENT`) is associated with a status record of the type `Event` containing two fields: a counter that is increased each time the event is signaled (used by the daemon); and a list of pending tasks waiting for the event to signaled, and which then will be re-started. Correspondingly, each task is associated with a status record of the type `Task` containing the following three fields: the state of the task (`SUSPENDED`, `RUNNING`, `ABORTED`, or `TERMINATED`); a list of those events it waits for in case the state is `SUSPENDED`; and finally a property called `prop`. This last property represents a condition that has to be satisfied before the task can be re-started in case it waits for an event. It's relevant when a task is not the owner of a lock, and hence some other task is supposed to achieve the property. Then the task must wait for this property to be achieved, hence the property becomes such a condition.

The state space of the model can now be declared, see Figure 6. The database is represented by the variable `db`, which is an array mapping property names into property values. The property lock table is represented by the variable `locks`, which is an array mapping property names into locks. In the LISP code, the property lock table is represented as a list of (property name, lock) pairs. Hence, in the LISP program, the existence of a lock l on a property name pn is represented by the fact that the pair (pn, l) is in the list. Since we model the property lock table as a mapping from property names to locks, the property name *pn* will *always* have an entry, and we therefore have to model the non-existence of a lock differently. We have reserved the property value 0 for those locks that are "non-existent". That is, if a property name maps to a lock with property value 0, it means it is not locked (corresponding to not being in the list in the LISP program). The constant:

```
#define undef_value 0
```

is introduced to denote this undefined property value.

Two variables are introduced which store the status of the events and the tasks. The variable `Ev` maps events into event status records, and similarly, the variable `active_tasks` maps task identifiers into task status records.

```
Property_Value  db[NO_PROPS];
Lock            locks[NO_PROPS];
Event           Ev[NO_EVENTS];
Task            active_tasks[NO_TASKS];
```

Figure 6. Variables

3.3 Events

Two operations are defined on events, corresponding to *waiting* for an event and *signaling* an event. These operations are represented by the procedures `wait_for_event`[5], Figure 7, and `signal_event`, Figure 8.

```
inline wait_for_event(this,a,p) {
  atomic{
     append(this,Ev[a].pending_tasks);
     append(a,active_tasks[this].waiting_for);
     active_tasks[this].prop.name = p.name;
     active_tasks[this].prop.value = p.value;
     active_tasks[this].state = SUSPENDED;
     active_tasks[this].state == RUNNING
  }
}
```

Figure 7. wait_for_event

The procedure `wait_for_event` takes three parameters: the parameter `this` (type `TaskId`) identifies the task that calls the procedure, and hence the task that wants to wait for an event to happen. The parameter `a` (type `EventId`) identifies the event to be waited for; and finally the parameter `p` (type `Property`) represents a property that must be satisfied in addition to the occurrence of the event before the calling task can be re-started. For example, when a task wants to wait for some other task to achieve the property `CAMERA_ON`[6], then it calls this procedure as follows: `wait_for_event(this,MEMORY_EVENT,CAMERA_ON)`. We shall refer to this property as the *restart condition*.

```
inline signal_event(a) {
  atomic{
    TaskId t;
    EventId e;
    list pending = [NO_EVENTS] of {EventId};
    Ev[a].count = Ev[a].count + 1;
    copy(Ev[a].pending_tasks,pending);
    do
    :: pending?t ->
       if
       :: (active_tasks[t].prop.value ==
             undef_value
           ||
           db_query(active_tasks[t].prop) )
          ->
          do
          :: active_tasks[t].waiting_for?e
               -> remove(t,Ev[e].pending_tasks)
          :: empty(active_tasks[t].waiting_for)
               -> break
          od;
          active_tasks[t].state = RUNNING
       :: else
       fi
    :: empty(pending) ->  break
    od
  }
}
```

Figure 8. signal_event

The body of the procedure is executed atomically, as within a critical section. First, the calling task is appended to the event's list of pending tasks (those waiting for the event to occur). Second, the event is appended to the task's list of events it is waiting for. Third, the restart condition `p` is stored in the task's status record in the `prop` field. Note that since PROMELA does not allow for assignments to record variables, each field has to be updated individually. Finally, the task is suspended by updating the task's `state` field. The waiting itself is realized by executing the statement:

```
active_tasks[this].state == RUNNING
```

[5]A procedure `wait_for_events` also exists, but it is very similar to `wait_for_event`.

[6]That is, the property name `CAMERA` must have the value `ON`.

This is a boolean valued expression (without side effects), and according to the semantics of PROMELA, it can only execute, and terminate, if its value is $true$. Hence, the calling task will wait until it becomes $true$, the intention being that the `signal_event` procedure at some later point will assign the value `RUNNING` to `active_tasks[this].state`.

The procedure `signal_event` takes one single parameter, namely the event `a` (type `EventId`) to be signaled, and then basically restarts all tasks waiting for that event, if their restart condition is satisfied that is. Three local variables are declared: `t`, `e` and `pending`, the last intended to hold the list of tasks waiting for the event. First, the event counter in incremented. The event counter is used by the daemon to determine whether a new, and untreated, signal has arrived, see Figure 22 page 11. Then the event's list of pending tasks is copied into the local `pending` variable, which hereafter in a loop is examined, task by task. Each task is extracted by the statement `next(pending,t)`, and hence stored in the local variable `t`.

Now, for each such waiting task `t`, if the task's restart condition `prop` is satisfied it is restarted. The restart condition is satisfied, if either its property value is undefined (equals `undef_value`), or if it indeed is satisfied in the database. The latter is the case if the expression:

```
db_query(active_tasks[t].prop)
```

evaluates to $true$. The function `db_query`, Figure 9, takes as parameter a property `p` (type `Property`), and returns $true$ if the database satisfies it (the property name denotes the property value).

```
#define db_query(p)
  db[p.name] == p.value
```

Figure 9. db_query

Hence, in case the restart condition is satisfied, an inner loop is entered, in which all events in the task's `waiting_for` list are examined, and for each such event: the task is removed from the event's list of pending tasks. In other words, the task is removed from all events since it's now restarted. In the LISP code, the body of the `signal_event` procedure is embedded within a critical section. A direct modeling of this in PROMELA results in an `atomic` construct around the body.

3.4 The Tasks

Tasks are modeled as PROMELA processes. Before we define what a task is, we shall, however, introduce a collection of procedures. The procedure `fail_if_incompatible`, Figure 10, is called by a task just before it tries to snarf a property, in order to check whether or not this is in conflict with already existing locks. The procedure takes as parameter the property `p` (type `Property`) to be snarfed, and returns $true$ if some other task has already snarfed the property name, but with a different, and therefore incompatible, property value. Recall that if the property name denotes a value different (`!=`) from `undef_value` in the lock table, then it has been locked.The result of this test is stored in the return variable `err`, which we shall see is used to direct control in the calling context.

```
inline fail_if_incompatible_property(p,err) {
  if
  :: (locks[p.name].value != undef_value &
      locks[p.name].value != p.value) ->
        err = 1
  :: else
  fi
}
```

Figure 10. fail_if_incompatible_property

The procedure `snarf_property_lock`, Figure 11, is called by a task to snarf a property. The procedure takes as parameter the identity, `this` (type `TaskId`), of the calling task; and the property, `p` (type `Property`), to be snarfed. The success of the operation is written back into the result variable `err`.

```
inline snarf_property_lock(this,p,err) {
  atomic{
    fail_if_incompatible_property(p,err);
    append(this,locks[p.name].sub);
    if
    :: locks[p.name].value == undef_value ->
         locks[p.name].value = p.value;
         locks[p.name].achieved = db_query(p)
    :: else
    fi;
    signal_event(SNARF_EVENT)
  }
}
```

Figure 11. snarf_property_lock

The procedure first checks whether the operation is compatible with the already existing locks. That is, there must not be a lock with the same property name, but with a different property value. Note that the result of this check is written into the `err` variable. In the calling context, Figure 17, we shall later see the effect of this result variable becoming $true$: an interrupt will occur and terminate the task.

The task is then appended to the list of subscribers to the property: those that want it to become $true$. Then, in case the property is in fact not already in the lock table, it is "inserted": the property name of `p` is set to denote the property value of `p`; and the `achieved` field is set to $true$ if the property already holds in the database (call of `db_query`), otherwise to $false$. Finally, the `SNARF_EVENT` is signaled with the result that the daemon will be restarted if waiting.

After having snarfed the property, it is now up to the task to achieve the property – if it is the owner that is. A task is the owner of a property, if it was the first to subscribe to it, and hence the first element in the property's subscriber list in the lock table. The procedure `find_owner`, Figure 12, determines this. It takes as parameter the property `p` (type `Property`), and returns in the result variable `owner` (type `TaskId`) the owner of that property in the lock table.

```
inline find_owner(p,owner) {
  locks[p.name].sub?<owner>
}
```

Figure 12. find_owner

When a task finally wants to achieve a property, it calls the procedure `achieve_lock_property`, Figure 13. The procedure takes as parameter the identity, `this` (type `TaskId`), of the calling task; and the property, `p` (type `Property`), to be achieved. The result (success) of the operation is stored in the result variable `err` (type `bool`). The task can only achieve the property if it's the owner. Hence, first it is determined which task is the owner of the property `p`: the procedure call `find_owner(p,owner)` stores the owner in the result variable `owner`. In case the owner equals the calling task (`this`), the property is achieved by a call of the procedure `achieve` (defined in Figure 14 and described below); and the `achieved` field is set to $true$. On the other hand, if the task is not owner, it must wait for the owner (some other task) to achieve the property. This waiting is initiated by a call of `wait_for_event` with the property `p` as restart condition. That is, the calling task will only be restarted on a memory event, if also the property `p` has been achieved, and hence is satisfied in the database.

The procedure `achieve`, Figure 14, is the one that really achieves the property by updating the database in case the property is not already satisfied in the database. The procedure takes as parameter the property `p` (type `Property`) to be achieved. If the property is already satisfied in the database – i.e. `db_query(p)` evaluates to $true$ – the procedure returns successfully[7]. Otherwise (`else`), in case the property is not already satisfied, a non-deterministic choice

```
inline achieve_lock_property(this,p,err) {
  TaskId owner;
  find_owner(p,owner);
  if
  :: owner == this ->
       achieve(p,err);
       locks[p.name].achieved = true
  :: else ->
       wait_for_event(this,MEMORY_EVENT,p);
  fi
}
```

Figure 13. achieve_lock_property

is made between success : updating the database to achieve the property, and failure : setting the boolean result variable `err` to $true$. This non-determinism reflects the fact that achievement can fail, and we abstract away from the details about the possible causes of failure.

```
inline achieve(p,err) {
  if
  :: db_query(p)
  :: else ->
       if
       :: db[p.name] = p.value
       :: err = 1
       fi
  fi
}
```

Figure 14. achieve

Once the task has achieved the property, it is ready to execute its real job while assuming that the property is invariantly satisfied (the daemon must intervene and stop the task if this is not the case). The procedure `closure`, Figure 15, represents this job. Its body is simple: a non-deterministic choice between just a `skip` statement and `false`. In case the first if–branch is chosen, `skip` is executed, and the procedure returns immediately. In case, on the other hand, the second branch is chosen, the execution of `false` will make the calling task block, since `false` cannot execute and terminate due to the semantics of PROMELA. This blocking is supposed to simulate a time consuming computation, and is needed later in order to conveniently formulate a certain correctness property to be verified. The correctness property basically says that *in case the property is broken (i.e.: is no longer in the database), the task will be terminated.* Now, suppose `closure` always terminated, this property would be trivially satisfied – hence the blocking alternative, allowing us to verify that the daemon really explicitly and violently aborts the task.

Assume that the task now has called the `closure`, and

[7] The first if–branch is equivalent to `db_query(p) -> skip`.

```
inline closure() {
  if :: true -> skip :: true -> hang fi
}
```

Figure 15. closure

that this terminates – either by choosing the `skip` branch, or because it has been aborted by the daemon. In this case the snarfed property no longer needs to be satisfied in the database, at least so far as what concerns this task. Hence, our task must release the property, meaning that it must be removed from the property lock table. This will allow other tasks to snarf and lock the same property name but with different property values. The releasing is done by a call of the procedure `release_lock`, Figure 16. It takes as parameter the identity, `this` (type `TaskId`), of the calling task; and the property, `p` (type `Property`), to be released.

```
inline release_lock(this,p) {
  atomic{
    remove(this,locks[p.name].sub);
    if
    :: empty(locks[p.name].sub) ->
         locks[p.name].value = undef_value
    :: nempty(locks[p.name].sub)
    fi
  }
}
```

Figure 16. release_lock

Its body is embedded within an `atomic` to model a critical section in the LISP code. The procedure basically removes the task from the property name's subscriber list in the lock table, since the task no longer subscribes to it. In case the subscriber list thereby becomes empty – no other tasks subscribe – the lock must be removed completely from the lock table. This is done by assigning the `undef_value` as property value to the property name in the table. Recall, that this is the way we model the absence of a lock (a property name maps to `undef_value`), whereas in the LISP program, the lock would simply be removed from the list of locks.

We can now finally define the top-level procedure `execute_task`, Figure 17 – called by a task – which snarfs the property to be maintained, achieves it, executes the body, and finally releases the property again. The procedure takes as parameter the identity, `this` (type `TaskId`), of the task; and the property, `p` (type `Property`), to be achieved and thereafter maintained to the end of the task.

We have up until now seen the variable `err` occurring as result parameter to most of our procedures.

```
inline execute_task(this,p)
{
  bool err = 0;
  {
    snarf_property_lock(this,p,err);
    achieve_lock_property(this,p,err);
    closure()
  }
    unless
  {err || active_tasks[this].state == ABORTED};

  active_tasks[this].state = TERMINATED;

  {release_lock(this,p)}
    unless
  {active_tasks[this].state == ABORTED}
}
```

Figure 17. execute_task

This variable is declared as a local variable at this outermost level, and hence passed as actual parameter to the procedures `snarf_property_lock` and `achieve_lock_property`. The calls of these two procedures are embedded within an `unless` construct of the form

{*snarf;achieve;job*} `unless` {*condition*}.

where the *condition* is that either `err` is *true*, or (||) the task has been aborted by the daemon: `active_tasks[this].state == ABORTED`. As we shall see in the next section, the daemon aborts a task exactly by assigning the value `ABORTED` to the `state` field in the tasks status record. The semantics of the `unless` construct is such that the snarfing, achieving and job is performed to the end, unless the condition becomes *true*, in which case the whole statement terminates abruptly. Hence, in the case that either the snarfing or the achievement goes wrong (`err` becomes *true*), or in the case that the task is aborted by the daemon – the whole operation terminates.

Once the snarfing, achieving and job has been terminated, either normally or abnormally, the statement:

```
active_tasks[this].state = TERMINATED;
```

is executed. This is part of the modeling of the LISP `unwind-protect` construct. The purpose of the assignment is to "restore" the value of the `state` field in case the task has been aborted by the daemon; and hence this field had got the value `ABORTED`. Restoring here means assigning a value different from `ABORTED`, since the value `ABORTED` will result in an immediate termination of the statement that follows. The last statement namely releases the property from the lock table, but is abruptly terminated in case the `state` field has, or gets the value `ABORTED` by the daemon, in case the daemon at this point discovers a violation. This is hence the second example of how the PROMELA `unless` interrupt construct is used to model task abortion.

We are now able to define the process type `Achieving_Task`, Figure 18, of which a process is spawned/instantiated for each task. It takes as parameter its own identity, `this` (type `TaskId`), which will be determined in the initialization section, Figure 25. A local variable `p` is declared, which is assigned the property to be snarfed and achieved by the task. In order to reduce the state space to model check, we have focused on property name 0 (`p.name = 0`), and we arbitrarily let the task achieve a property *value* which is identical to the task's identity: 1 or 2 since, as we shall see, only two tasks will be spawned. Finally the main procedure is called, which performs the snarfing, achievement, job and release. Note that all tasks in this model perform the same job (`closure`). This is an example of an abstraction from the LISP code, where details regarded as unimportant for the verification have been omitted.

```
proctype Achieving_Task(TaskId this)
{ Property p;
  p.name = 0;
  if
  :: this == 1 -> p.value = 1;
  :: this == 2 -> p.value = 2
  fi;
  execute_task(this,p);
};
```

Figure 18. Achieving_Task

3.5 The Daemon

The daemon is responsible for detecting whether violations of locks occur in the database. That is, it must react in case a property name pn in the lock table is locked to a property value pv_1, and the corresponding `achieved` field is set to $true$ (hence a task relies on it and is executing its job), but pn denotes a value $pv_2 \neq pv_1$ in the database. In that case the daemon must interrupt the tasks relying on the property (pn, pv_1) and repair the violation by updating the database, assigning pv_1 to pn again. The procedure `interrupt_task`, Figure 19, takes as parameter a task, `t` (type `TaskId`), to be aborted, and does this by simply assigning the value `ABORTED` to the `state` field of the task's status record. This will cause the relevant `unless` construct to terminate the task (Figure 17).

```
inline interrupt_task(t) {
  active_tasks[t].state = ABORTED
}
```

Figure 19. interrupt_task

The procedure `property_violated`, Figure 20, is used to determine whether locks have been violated. It is called for each property name having an entry in the lock table (0 and 1 in our reduced case), and takes as parameter this property name `pn` (type `Property_Name`); returning the result back into the variable `lock_violation` (type `bool`). The body consists of a single assignment to the result variable, which becomes $true$ iff. the property name is locked (property value is defined), has been achieved, but has a property value different from the one in the database.

```
inline property_violated(pn,violation) {
  atomic{
    violation =
      (locks[pn].value != undef_value &
       locks[pn].achieved &
       db[pn] != locks[pn].value)
  }
}
```

Figure 20. property_violated

The procedure `property_violated` is called from the procedure `check_locks`, Figure 21, which checks the whole property lock table for violations. This is done in a loop that iterates over all the property names $\{0 \ldots \texttt{NO\_PROPS} - 1\}$. In fact, there are two such loops. In the first loop, in case of a property name `pn` being violated (denoting something different in the database than in the lock table), all the subscribers to that property name are interrupted. This is done by first taking a copy of this subscriber list, storing it in the local variable `sub`, and then extracting each task `t` from `sub`, one by one (`next(sub,t)`), and interrupting it.

In the second loop, a `break` statement causes termination as soon as a violation is found, the purpose being just to examine whether there are any violations left. This result is returned in the result variable `violation` of the `check_locks` procedure. The result will then be used in the calling context to decide whether the database should be recovered.

The two loops are also present in the LISP code, and since they result in an unexpected behavior found during verification, to be explained in section 4.4, we quote Erann Gat's explanation of the code:

```
inline check_locks(lock_violation) {
  Property_Name pn;
  list sub = [NO_TASKS] of {TaskId};
  TaskId t;
  pn = 0;
  do
  :: pn < NO_PROPS ->
       property_violated(pn,lock_violation);
       if
       :: lock_violation ->
            atomic{copy(locks[pn].sub,sub)};
            do
            :: sub?t -> interrupt_task(t);
            :: empty(sub) -> break
            od
       :: else
       fi;
       pn++
  :: else ->  break
  od;
  pn = 0;
  do
  :: pn < NO_PROPS ->
       property_violated(pn,lock_violation);
       if
       :: lock_violation -> break
       :: else
       fi;
       pn++
  :: else -> break
  od
}
```

Figure 21. check_locks

The structure of this code is complicated by the design requirement that an external process may be responsible for restoring violated properties. (In the case of the DS1 RA this is the MIR process.) So tasks need to be able to decide, when a property that they want maintained is violated, if they want to wait for the external process to restore the property or if they want to fail right away. If all the tasks that rely on a violated property fail right away then there is no need to restore the property, since no one is relying on it any more. So check-locks makes one pass through the property locks and injects failures into all tasks that rely on violated properties. It then yields to give all those tasks a chance to abort themselves if they choose to. Then it checks to see if there are any violated properties left. This is returned as a boolean to the first part of the maintain-properties-daemon, which runs in an infinite loop.

The daemon process itself will be an instance of the process type `Daemon`, Figure 22, which as parameter takes its own identity, `this` (type `TaskId`). It declares three local variables: `lock_violation`, to hold the result of `check_locks`; `event_count`, to keep track of new events; and `first_time`, which is *true* only when the daemon starts. The body consists of an infinite loop, which for each iteration does the following. The procedure `check_locks` is called to determine if there are any violations. If there are, the procedure `do_automatic_recovery` is called, which has not been shown here, but which basically repairs the database by making it consistent with the lock table. That is, `do_automatic_recovery` performs the update `db[pn] = pv` for each property name `pn`, where the lock table maps `pn` to `pv`, but the database `db` does not.

```
proctype Daemon(TaskId this) {
  bit lock_violation;
  byte event_count = 0;
  bit first_time = true;
  do
  :: check_locks(lock_violation);
     if
     :: lock_violation ->
          do_automatic_recovery()
     :: else
     fi;
     if
     :: (!first_time &&
         Ev[MEMORY_EVENT].count +
         Ev[SNARF_EVENT].count != event_count )
           ->
           event_count =
             Ev[MEMORY_EVENT].count +
             Ev[SNARF_EVENT].count
     :: else ->
         first_time = false;
         wait_for_events(this,
           MEMORY_EVENT,SNARF_EVENT)
     fi
  od
};
```

Figure 22. Daemon

Then, in the second if construct, it is decided whether the daemon should stop and wait for a new memory or snarf event to occur (call of `wait_for_events`), or whether it should continue with yet another iteration, calling `check_locks` and perhaps `do_automatic_recovery`. Another iteration is needed if a memory event or a snarf event has occurred since the daemon was restarted last time. This is expressed as follows: when `first_time` is *true* (initial state), the daemon simply calls `wait_for_events`, and then waits for either a `MEMORY_EVENT` or a `SNARF_EVENT` to occur. The procedure `wait_for_events` has not been shown, but is like `wait_for_event`, Figure 7, except that not *one* – but either of *two* events are waited for. A second difference is that a boolean variable daemon_ready is set to *true* as the last thing before the daemon starts waiting. This is used during initialization, Figure 25, as we shall see. Now, in case it's not the first iteration, the test:

```
Ev[MEMORY_EVENT].count +
Ev[SNARF_EVENT].count
        !=
    event_count
```

is executed. It evaluates to *true* in case the event counter event_count differs from the sum of the event counters for the memory and snarf events. If there is a difference, it means that there has been an event since last time event_count was updated, and this must result in yet another iteration before calling wait_for_events. Before this extra iteration, the event_count variable is, however, updated.

3.6 The Environment

Violations are introduced by the environment, here modeled by the process type Environment, Figure 23. An instantiation of this will run in parallel with the tasks and the daemon, and may cause a database change at any moment in time. The change is here fixed to property name 0 getting property value 0. This will introduce a violation in case a lock has been created for property name 0 with a value different from 0. The MEMORY_EVENT is furthermore signaled to wake up the daemon, in case it's not already running. The daemon shall then hopefully discover the violation just introduced.

```
proctype Environment()
{ atomic{
    db[0] = 0;
    signal_event(MEMORY_EVENT)
  }
};
```

Figure 23. Environment

3.7 Initialization

All processes, the daemon and the tasks, are all instantiated with the procedure spawn, which takes as parameter the parameterized task (a proctype) to be spawned; and as a second parameter it takes the task identity t (type TaskId) of the task to be spawned. The second parameter is then fed as actual parameter to the first parameter in a

```
#define spawn(task,t)
  atomic{
    active_tasks[t].state = RUNNING;
    run task(t)
  }
```

Figure 24. spawn

run statement. Before that happens, the task's state field gets the value RUNNING.

Finally, the system is initialized by spawning the daemon with identity 0, the two tasks with identity 1 respectively 2, and then the environment, see Figure 25. Before the tasks are spawned, however, the daemon is waited for to terminate its own local initialization. This is done by waiting for the variable daemon_ready to become *true*. In fact, this models the fact that the daemon will be started before any other task in the system.[8]

```
init
{
  spawn(Daemon,0);
  daemon_ready == true;

  spawn(Achieving_Task,1);
  spawn(Achieving_Task,2);
  run Environment()
}
```

Figure 25. initialization

4 Analysis wrt. Selected Properties

4.1 Identifying Properties to be Verified

The model has been analyzed wrt. the following two properties, here expressed informally:

RELEASE Property: *A task releases all its locks before it terminates.*

ABORT Property: *If an inconsistency occurs between the database and an entry in the lock table, then all tasks that*

[8] In an early model, the tasks were spawned without waiting for the daemon, but that lead to the discovery of an error by the model checker, see section 4.7. The error was basically that a lock violation could occur before the daemon got to its initial waiting point, which the first time is unconditional!; and hence the daemon would just ignore the violation and call wait_for_events.

rely on the lock will be terminated, either by themselves or by the daemon in terms of an abort.

In the following we shall demonstrate how we have formulated these properties in terms of PROMELA assertions (assert–statements) and LTL formulae, and we shall show the results of applying the SPIN model checker to verify these properties. It turns out that none of them are satisfied in the presented model, a discovery that has lead the RA programmers to make corrections in the LISP code.

The verification of the two properties lead to the direct discovery of four errors (wrong code) – one breaking the RELEASE property, and three breaking the ABORT property. All of these errors are classical in the sense that they arise due to processes interleaving in unexpected ways. Hence, for example, two errors can be corrected by introducing critical sections around the troubled code. Furthermore, a less serious, but at that time yet undiscovered efficiency error (code executed twice instead of once) was discovered just by observing generated traces from the model checking. Hence, a total of five errors were identified in the LISP code, four of which being important. In addition to this, a verification *"highlighted the need for a mechanism to insure that the daemon has reached 'steady state' before proceeding"*. Although this was not considered as a direct error, we have reported it here.

4.2 Error 1 – The RELEASE Property

RELEASE Property: *A task releases all its locks before it terminates.*

4.2.1 Formalizing The Property

In order to formalize this property, we need to define what it means for a task to have released its locks. The function `not_subscriber` in Figure 26 returns *true* if task `t` does not subscribe to property name `pn`, hence has released it's lock on `pn`.

```
#define not_subscriber(this,pn)
  !locks[pn].sub??[eval(this)]
```

Figure 26. RELEASE **predicate**

To state the RELEASE property, we modify the definition of the process `Achieving_Task`, Figure 18, adding an assert–statement after the call of `execute_task`. This modification is shown in Figure 27.

When a task terminates (end of `execute_task`), we expect that it is no longer subscriber of the property name it has snarfed (`p.name`), and hence we expect the assertion to be satisfied.

```
proctype Achieving_Task(TaskId this)
{ Property p;
  p.name = 0;
  if
  :: this == 1 -> p.value = 1;
  :: this == 2 -> p.value = 2
  fi;
  execute_task(this,p);
  assert(not_subscriber(this,p.name))
};
```

Figure 27. Formalization of RELEASE **property**

4.2.2 Error Detection

Running the SPIN model checker on the modified program yields an error trace illustrating that the assertion is not always satisfied. The trace (shortened) describes the following sequence of events:

1. A task starts, running process `Achieving_Task` in Figure 27. This implies a call of the procedure `execute_task`, defined in Figure 17.
2. The procedure `execute_task` does the snarfing, the achieving, the closure call, and then executes the `active_tasks[this].state = TERMINATED` statement, ready to release its lock by calling the `release_lock` procedure.
3. At this point, just before the call of `release_lock`, the `Environment`, defined in Figure 23, introduces an inconsistency in the database such that the property value of property name 0 becomes 0 in the database, while it is expected to be different from 0 by the running task.
4. The Daemon, Figure 22, detects this inconsistency and aborts the task in the `check_locks` procedure, Figure 21, by calling the procedure `interrupt_task` defined in Figure 19. That is, the status of the task becomes `ABORTED`.

The way the `execute_task` is programmed, this abortion will at this point result in an exit of this procedure, hence skipping `release_lock`. This is caused by the PROMELA semantics of the `unless` construct as occurring in (Figure 17):

```
{release_lock(this,p)} unless
  {active_tasks[this].state == ABORTED}
```

Hence, even though the snarfing, achieving, and closure is protected against abortion (if an abort occurs there, `release_locks` will be called anyway), the lock releasing itself is not protected: if an abort occurs here, the lock releasing is abandoned. This reflects the semantics of the applied ESL construct of the form "`Protect P Exit Q End`" executing `P` and then `Q` (the lock releasing), with the addition, that if an abort occurs during the execution of `P`, the remainder of `P` is skipped, and `Q` gets executed. Hence, the idea is that `Q` always gets executed, even if an abort occurs during the execution of `P`. The unexpected situation is that an abort can occur during the execution of `Q`, with the result that the rest of `Q` will not be executed.

4.2.3 Error Correction

The identified error can be corrected by protecting the lock releasing itself against abortion. This we have done in a modified version of the PROMELA model[9], such that lock releasing cannot be aborted. Hereafter the RELEASE property is verified correct using the SPIN model checker. How the modification is done in the LISP program is beyond the scope of the present paper.

4.3 Error 2 – The ABORT Property

As already mentioned, three verifications of this property were performed, each demonstrating an error in the model causing the falsification of the property. We will present the first verification in this section.

ABORT Property: *If an inconsistency occurs between the database and an entry in the lock table, then all tasks that rely on the lock will be terminated, either by themselves or by the daemon in terms of an abort.*

4.3.1 Formalizing The Property

Our verification will be concrete in that we shall focus on task 1. We shall state, that if task 1 has snarfed and achieved property name 0, assuming it to denote property value 1 in the database (as stated in Figure 18) then if this assumption is broken by the environment, task 1 will be terminated. First of all, we formally define what it means for task 1's assumption to be broken, and what it means for task 1 to be terminated. Figure 28 shows two such predicates.

The predicate `task1_property_broken` returns *true* in case of an inconsistency between `locks` (mapping 0 to 1) and `db` (mapping 0 to 0) in a situation where the task assumes the property to have been achieved. The predicate `task1_terminated` is *true* when the state of task 1 is either `TERMINATED`, set by itself, or `ABORTED`, set by the daemon. The ABORT property can now be stated as an LTL formula as shown in Figure 29. The property states that *"in all states (* `[]` *), if* `task1_property_broken` *holds, then eventually (<>), at some future point in time,* `task1_terminated` *will hold"*.

```
#define task1_property_broken
  (locks[0].value == 1 &
   locks[0].achieved &
   db[0] == 0)

#define task1_terminated
  (active_tasks[1].state == TERMINATED ||
   active_tasks[1].state == ABORTED)
```

Figure 28. ABORT predicates

```
[](task1_property_broken -> <>task1_terminated)
```

Figure 29. Formalization of ABORT property

It's relevant here to note that this property only makes sense to verify if task 1 has the potential of not terminating at all in case it's not aborted. This is the reason why the closure is defined as in Figure 15. The closure can arbitrarily choose the `true -> false` branch whereby it will hang on the `false` expression without being able to progress according to the semantics of PROMELA. Of course, in the real LISP program a task will probably always terminate, and we are therefore really interested in the task being terminated within a certain time frame. However, since PROMELA cannot deal explicitly with time, we have chosen only to focus on the distinction between termination (at some future unspecified time) and non-termination.

4.3.2 Error Detection

Applying the SPIN model checker to the above property yields an error trace demonstrating, that the property is not satisfied in the model. The trace illustrates the following sequence of events.

1. The daemon, Figure 22, starts and reaches a waiting position. That is, it calls `wait_for_events`, where after it waits for an event to occur.
2. A task, Figure 18, starts; snarfs and achieves successfully, thereby signaling `SNARF_EVENT` from

[9] Basically by removing the `unless` construct attached to the call of `release_lock`.

snarf_property_lock, Figure 11, and then starts executing its closure. This closure chooses the `true -> false` branch. Hence if it is not aborted it will never terminate (corresponding to a time consuming computation in a real setting).

3. The daemon has been woken up by the signaling of the SNARF_EVENT. No inconsistencies are found, and the daemon then decides to wait again. That is, it takes the decision to call wait_for_events, but delays a bit before doing it. Note the delay between "decision" and "action" here. The decision to wait is taken in the PROMELA model in Figure 22 at the last `else` branch.

4. The environment, Figure 23, introduces an inconsistency, and signals the MEMORY_EVENT. However, this signal will not affect the daemon since it already *has* decided to call wait_for_events. It will for example not check whether the event counters have been updated.

5. The daemon now calls wait_for_events unconditionally, and hence, starts waiting. The task hence does not get aborted, and continues with its "big" computation.

4.3.3 Error Correction

A solution to the detected problem is to embed the decision to wait and the waiting itself into a critical section, that cannot be interrupted by other processes. In PROMELA, the `atomic` construct can be used to define a critical section, and Figure 30 shows how the `Daemon` has been extended with such a critical section around the code portion that decides whether to wait or not (the last if–statement).

Reapplying the SPIN model checker to verify the ABORT property formulated in Figure 29 for the modified model, however, shows that there is still an error in the system, as described in the next section.

4.4 Error 3 – The ABORT Property

With the corrected model, we re-apply the SPIN model checker to the same property, hoping that it now holds. As already mentioned and as will be demonstrated, it still does not hold.

4.4.1 Formalizing The Property

The property to be verified is as before, namely the one pictured in Figure 29.

```
proctype Daemon(TaskId this) {
  ...

   atomic{  -- added
     if
     :: (!first_time &&
         Ev[MEMORY_EVENT].count +
         Ev[SNARF_EVENT].count != event_count )
           ->
           event_count =
             Ev[MEMORY_EVENT].count +
             Ev[SNARF_EVENT].count
     :: else ->
         first_time = false;
         wait_for_events(this,
           MEMORY_EVENT,SNARF_EVENT)
     fi
   }
  ...
};
```

Figure 30. New `Daemon`

4.4.2 Error Detection

Applying the SPIN model checker yields an error trace demonstrating, that the property is not satisfied in the model. The trace illustrates the following sequence of events.

1. The daemon, Figure 30, starts and reaches a waiting position. That is, it calls wait_for_events, where after it waits for an event to occur.

2. A task, Figure 18, starts; snarfs and achieves successfully, thereby signaling SNARF_EVENT from snarf_property_lock, Figure 11, and then starts executing its closure. This closure chooses the `true -> false` branch. Hence if it is not aborted it will never terminate (corresponding to a time consuming computation in a real setting).

3. The daemon, Figure 30, has been awakened by the signaling of the SNARF_EVENT, and calls check_locks[10], Figure 21. Now check_locks consists of two loops, one executed before the other. The first loop looks for violations and interrupts tasks depending on violated properties. The second loop just checks for violations (and does *not* interrupt tasks). Hence, the daemon executes the first loop – finds no violation – and then is now *ready* for executing the second loop.

[10] In fact, check_locks is called twice, see section 4.6, and it's the second – and last – call which is referred to.

4. The environment, Figure 23, introduces an inconsistency, and signals the MEMORY_EVENT. However, the daemon is already running. Hence, the only effect is that the MEMORY_EVENT counter is increased.

5. The daemon now executes the second loop of check_locks, and finds the violation. Hence, it calls do_automatic_recovery, which repairs the violation by updating the database.

6. Due to the signaling of the MEMORY_EVENT in item 4 by the environment, the MEMORY_EVENT counter has been increased, and hence the daemon will execute check_locks again. However, since the violation has been repaired, the daemon will not find anything wrong, and will therefore finally call wait_for_events and then wait for a new event to occur. The task is still executing, and has not been aborted.

4.4.3 Error Correction

At the time when this error trace was generated, we believed that it was in fact an intended behavior, and only later was it confirmed to be an unexpected and undesired behavior – an error. Hence, we did not correct it; and even with the knowledge we have now, it is not evident for us how to correct this.

4.5 Error 4 – The Abort Property

4.5.1 Formalizing The Property

Since we originally did not regard the above situation as an error, we continued the verification as if it was a correct behavior. That is, in order to investigate the existence of additional errors, we had to reformulate the Abort property such that the above situation was allowed[11]. Hence, since the model may repair an inconsistency without aborting tasks, the property shall state this: in case of a broken property, then either this is repaired by the daemon, or the task is terminated (by itself or the daemon). For this purpose we introduce the predicate task1_property_repaired in Figure 31. This predicate returns *true* if the database and the lock table match wrt. to property name 0 (recall that we have focused on task 1 that snarfs property name 0).

The new correctness property using this new predicate is shown in Figure 32. The property states that *"in all states, if* task1_property_broken *holds, then eventually either* task1_terminated *or* task1_property_repaired *will hold"*.

[11] Even, when it later was confirmed as an error, we did not know how to correct it, and hence a reformulation of the property was still needed in order to avoid the repair situation to be identified by the model checker as an error.

```
#define task1_property_repaired
  locks[0].value == db[0]
```

Figure 31. Abort predicate

```
[](task1_property_broken ->
  <>(task1_terminated ||
     task1_property_repaired))
```

Figure 32. Re-formalization of Abort property

4.5.2 Error Detection

Applying the Spin model checker to the above property yields an error trace demonstrating, that the property is not satisfied in the model. The trace illustrates the following sequence of events.

1. Task 1, Figure 18, starts, and eventually calls achieve_lock_property, Figure 13. This procedure contains the two lines:

   ```
   achieve(p,err);
   locks[p.name].achieved = true
   ```

   That is, a call of achieve, which updates the database, and then an assignment to the achieved field. In the trace, the achieve procedure is called, and then the task execution is delayed, hence, the assignment to the achieved field is delayed.

2. At this point, the Environment, Figure 23, introduces an inconsistency in the database such that the property value of property name 0 becomes 0 in the database, hence, destroys the just achieved property.

3. The daemon, Figure 30, awakened by the environment change starts looking for an inconsistency, but finds none since the achieved field has not been set yet, and the daemon requires this to be *true* in order for an inconsistency to be existing, see the definition of procedure property_violated Figure 20. Hence, the daemon discovers nothing and goes to sleep again.

4. The task from above now assigns *true* to the achieved field, and continues as if everything was consistent.

Hence, an inconsistency has been introduced, but it has not been discovered by the daemon, and hence, is not repaired, neither is the task aborted.

4.5.3 Error Correction

A solution to the problem is the embedding of the two lines of code in the `achieve_lock_property` procedure into a critical section, such that updating the database and the `achieved` field is always done in one indivisible action. For this purpose we introduce an `atomic` construct around the two lines in the PROMELA model, as shown in Figure 33.

```
inline achieve_lock_property(this,p,err) {
  TaskId owner;
  find_owner(p,owner);
  if
  :: owner == this ->
       atomic{  -- added
         achieve(p,err);
         locks[p.name].achieved = true
       }
  :: else ->
       wait_for_event(this,MEMORY_EVENT,p);
  fi
}
```

Figure 33. New `achieve_lock_property`

The SPIN model checker now certifies that the ABORT property in Figure 32 is satisfied in this new model.

4.6 Error 5 – An Efficiency Problem

During the examination of the error traces generated by the verifications above, yet a fifth error has been discovered in the LISP code. In the PROMELA model it concerns the process `Daemon` in Figure 22.

It occurs that `check_locks` is called twice whenever the daemon has hung after a call of `wait_for_events`, and then is restarted after a signal to one of the events it waits for. That is, when one of these events is signaled by a call of `signal_event`, Figure 8, the event counter for that event is incremented in addition to the restart of waiting tasks. This means that when the daemon has executed `check_locks` (and perhaps `do_automatic_recovery`) once, then the test:

```
Ev[MEMORY_EVENT].count +
Ev[SNARF_EVENT].count
          !=
     event_count
```

will evaluate to $true$, and hence another iteration of the loop is begun, re-executing `check_locks`. The RA programming team has confirmed this as an error, although one of low priority.

4.7 A "Daemon–Ready" Flag Perhaps Needed

In an early model, the tasks were spawned without waiting for the daemon to initialize itself. That lead to the discovery of an error by the model checker. The error was basically that a lock violation could occur before the daemon got to its initial waiting point, which the first time is unconditional!; and hence the daemon would just ignore the violation and call `wait_for_events`. This was not considered an error, because the daemon will always start before everything else. However, the following response from Erann Gat shows that a change to the LISP program could be needed.

> This would be a problem if the daemon were started late. However, I don't think this is a problem in practice because all the daemons are started long before anything else happens. But this does highlight the need for a mechanism to insure that all the daemons have reached "steady state" before proceeding.

Hence, we don't consider this as a caught error, but we regard it as an increased insight given to the RA programming team.

5 Evaluation by the RA Programming Team

This section contains Erann Gat's evaluation of our work. His comments were given during email communications, which were not originally intended to be published. He, however, later approved their publication.

A first sub–section contains his responses to our error reports. A second sub–section contains his responses to three general questions posed after our work had been terminated.

5.1 The Programmer's Remarks to Our Error Reports

In this section we quote Erann Gat on his remarks to our error reports. That is, for each error we discovered, and which has been explained in section 4, we quote his response to our report to him. We present the quotations in the order they appeared in time, although this in certain cases differs from the order of presentation in section 4.

Error 1 – RELEASE Property (section 4.2):

I think this is a real error. It would only arise if a task gets a timer interrupt in between exiting the body of the unwind-protect and entering the critical section of the *release-locks, but I don't know of any reason why that should not happen on occasion. This is a particularly pernicious bug. It arises only because you are in a multi-threaded environment, and only in very obscure circumstances that are very unlikely to arise during testing. Congratulations! You have just converted me into a believer in formal methods.*

Error 4 – ABORT Property (section 4.5):

Ah, good point. You are correct, this is a bug. I'm impressed! This makes two bugs you guys have discovered through formal methods that we almost certainly would never have caught any other way.

Error 2 – ABORT Property (section 4.3):

Yep, another bug. This one is an instance of a classic pattern: not wrapping a conditional wait-for-events inside a critical section. This sort of mistake is very easy to make and happens all the time in our code. Thanks for catching this one!

Error 5 – EFFICIENCY Problem (section 4.6):

No, it's a bug, but since it's just an efficiency problem it's pretty low priority.

Error 3 – ABORT Property (section 4.4):

You have, however, found a (already known) design flaw. There can be a significant time lag between a property being violated and a task being informed of the violation. The property lock daemons should really reside in the property database and be triggered automatically whenever contradictory information is asserted. This is on the list of things to do.

Question: Is it not the case, that a task might **never** be informed?

Ah, good point! I had neglected to consider the case where a new assertion that violates a lock happens in the middle of check-locks. It's hard to get out of a single-threaded mind set! Thanks for pointing this out.

Question: But is it an error? Or is it "just" unexpected?

... Seriously though, the intent was that tasks would be notified whenever a locked property was violated after initial achievement. In some cases this can be important. For example, if a pointing constraint is violated it might be important to know, even if the constraint is automatically restored.

5.2 The Programmer's Answers to 3 General Questions

We asked Erann Gat three general questions about the model checking effort we had carried out. Below we quote his answers to each of them.

Question 1:

Did our work have any impact on your work?

Answer:

You've found a number of bugs that I am fairly confident would not have been found otherwise. One of the bugs revealed a major design flaw (which has not been resolved yet). So I'd say you have had a substantial impact. If nothing else you have helped us improve the quality of our product well beyond what we otherwise would have produced.

Question 2:

How serious were the errors we found? Any examples of what could have gone wrong? Would they only occur rarely or be harmless?

Answer:

The errors you found were the sort that would manifest themselves only under very particular sets of circumstances involving precise timing, so these errors rarely manifest themselves. This makes them both more and less serious – less serious because they are unlikely to actually occur, more serious because if they occur at all they are likely to occur for the first time under actual flight conditions. The overall architecture is designed to be robust in the face of such errors (we have multiple layers of software redundancy) so it is unlikely that these errors would have caused problems more serious than lost time, but one never knows. Every bug is potentially a mission-killer, and generally the ones that do kill the mission do so in ways that one never imagines until it happens.

Question 3:

What was/is your general attitude towards formal methods, before and after this exercise?

Answer:

I used to be very skeptical of the utility of formal methods. This is at least partly due to the fact that I had a misconception about the way in which formal methods would be used. I thought that formal-methods advocates wanted to "prove correctness" of software systems. I believed (and still believe) that that is impossible. However, what you have been doing is finding places where software violates design assumptions, which is not the same thing as proving correctness. To me you have demonstrated the utility of this approach beyond any question. I would like very much to learn more about your work.

6 Conclusion

In this paper the results of verifying the RA Executive have been described, and we shall now try to present some of our derived reflections.

6.1 Analysis of the Effort

The major effort without doubt went into the modeling, hence in obtaining a PROMELA program from the LISP program. This modeling activity can be regarded as consisting of three sub-activities: *comprehension*, *abstraction* and *translation*, see Figure 34. By *abstraction* we mean the activity of reducing the program to become a finite state system, small enough for efficient verification. This task consists of removing irrelevant code, replacing infinite types with interval types, limiting the number of tasks running, etc. By *translation* we mean the activity of writing the actual PROMELA code, for example mapping the property lock list in the LISP program into an array representation in the PROMELA program. A pre-requisite for modeling is a certain *comprehension* of the source program, the LISP program in this case. That is, an understanding of the program that makes it possible to perform good abstractions.

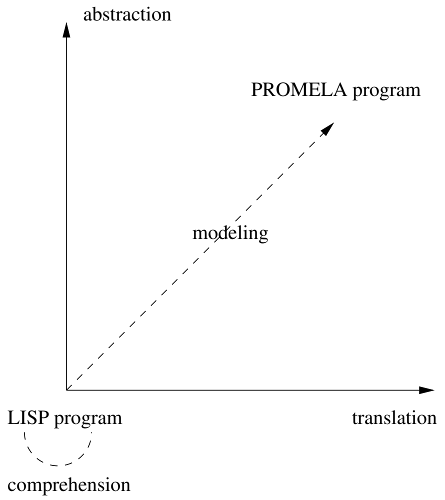

Figure 34. Modeling = comprehension + abstraction + translation

The *comprehension* activity was clearly the hardest, since the LISP program used many macro-definitions, and since we did not have direct access to the programmer for explanations. The *translation* phase was also non-trivial due to the strength of the LISP language compared to the weaker PROMELA language. Basically LISP is probably one of the most powerful languages around since it provides a combination of untyped functional programming and imperative object oriented programming. Hence, the mapping often resulted in code "blow up". Interestingly enough, the *abstraction* activity was the easiest. Once a piece of code was understood, deciding what to keep and what to remove was often quite clear.

Of course, the notion of translation is only relevant in the situation where model checking is applied to an already existing program, as was the case here. When model checking is instead applied during the early design phases, before a program is written, modeling becomes much more like traditional programming activity.

The modeling effort took 2 people about 6–8 weeks. The verification effort was in contrast small, about a week. Once the model was formulated, it was easy to formulate the properties to be verified, either in terms of assertions or in terms of LTL formulae. The model checker found the 5 errors right away.

6.2 Language Considerations

PROMELA was chosen as the modeling language due to its support of dynamic process creation. RA tasks are created and deleted dynamically over time, and we initially considered this as being important. As it turns out however, our verifications only involve a static number of processes.

The PROMELA language seen as a notation represents very much the state of the art in model checking languages, and is acceptable for the problem. However, a few highly recommended improvements for the language came out as a result of our efforts, as documented in [4]. Some of these recommendations have been adopted in the latest version of SPIN, inspired by our work during several email communications with Gerard Holzmann. First of all, the lack of procedural abstraction was felt as a clear drawback. Macros could be used, but they don't very well support local variables nor parameter type checking (not to mention typing "\" at the end of each macro definition line). Furthermore, the SPIN tool set does not support macros very well, since the type checker as well as the simulator cannot refer to lines within macros. This means that when for example simulating the result of a verification, one cannot really follow what goes on, and one has to examine instead the error trace in an ad hoc way (loading it into emacs for example). The advantage of macros is that there is no overhead in using them: macro calls are simply expanded out before the model checker is applied. These observations lead Gerard Holzmann to incorporate the "inline" procedures into PROMELA as announced in the SPIN newsletter 22 (April 1998). Also nested atomic constructs were regarded useful, and consequently incorporated into PROMELA. Still on the wish list are local variables, enumerated types, type equations and constant definitions. Generally, a complete avoidance of macro definitions would be preferable.

In [7] it *is* described how procedures can be modeled in terms of processes that are spawned, and which communicate their result back on a channel. That is, a procedure is modeled as a process, and each time the procedure is called, such a process is spawned. We tried this solution, but it turned out to cause two problems. First of all, SPIN had a limit on the number of processes allowed to be created, and this limit (256) was quickly reached in a program using a lot of procedural abstraction. The problem was, that in SPIN processes were not killed when they terminated. Due to an email conversation with Gerard Holzmann, SPIN was changed such that processes were killed and removed from the memory upon termination. However, this did not remove the second problem, that modeling procedure calls as process spawning is expensive, and slows down verification considerably. When we went over to using macros, verifications terminated an order of magnitude faster. A third solution is to model each procedure by a process, which is spawned only once, and where each procedure call then is modeled solely by a communication to that process. Hence, there is only one (1) spawning for each procedure *declaration*, in contrast to each procedure *call* as suggested in [7]. We have not experimented with this solution.

6.3 Tool Considerations

Even though manual translation was regarded harder than manual abstraction, we believe that translation can be mostly fully automated, at least for traditional programming languages such as JAVA (in contrast to LISP) whereas abstraction requires some human guided interactive tool support. Hence, the above experiences suggest that the translation activity should be automated as much as possible; perhaps a model checker could even be "hardwired" for the programming language (thereby avoiding the indirect translation into a modelchecking language). Abstraction, however, is not likely to be easily automated, and we therefore suggest an interactive tool, an *abstraction–workbench*, for supporting such abstractions. With such a tool, one could for example annotate a complete program with *abstraction information*, such as: *Putting a maximal bound on number of iterations in a loop*, *Limiting an infinite (or big) type to a finite (and small) subtype*, *Changing the type of a variable, and changing all related operations*, or *Omitting, replacing, adding code*. Also more automated capabilities could be considered such as for example program slicing.

We imagine that the tool will allow the user to make arbitrary (sound as well as unsound) modifications to his program, and not just sound modifications that are "correct" in some sense. In other words, it is important to note, that we have not proved the abstracted PROMELA program to be "correct" wrt. to the LISP program. That is, we have *not* shown that if a property holds in the PROMELA program it also holds in the LISP program. Such abstraction proofs are of course of big interest, and computer aided support for such correct abstractions is obviously desirable. Some abstractions can be done fully automatically, such as for example program slicing. More sophisticated approaches to abstraction have been attempted based on theorem proving, where a theorem prover is used to formulate abstractions and prove them correct, see for example [5] [6]. Some work tries to automate these more sophisticated abstractions [1] [2] [3]. The PVS interactive theorem prover [8] has a general higher order logic, allowing specification and verification of general infinite state transition systems. Particularly interesting is the current effort to effectively integrate model checking into PVS (as described in [8]).

In general, such proofs are, however, very hard to create, and we believe, that just the above mentioned abstraction–workbench could be *extremely* useful, although simpler in purpose. Interestingly enough this simpler approach is not even yet state of the art. We believe that a decent purpose of applying model checking is to find errors rather than to prove correctness, and for this purpose such a simpler tool is useful. Such a tool should in addition support strong version control, since such annotations may be changed quite often in the early phases of the verification activity.

6.4 Closing Remarks

We regard the exercise as highly successful in the sense that we found five errors quite easily, once the model was constructed. The errors were all classical concurrency related errors, where unforeseen inter-leavings between processes caused undesired events to happen. According to the RA programming team, the effort has had a major impact, locating errors that would probably not have been located otherwise, and identifying a major design flaw.

The major effort consisted in building the model, but we claim that this activity can be made much more efficient by providing translation and abstraction tools. Furthermore, the better the modeling language, the easier the modeling. Especially if one considers using a model checker in the early stages of systems design, before programming is begun, a nice notation is absolutely a *must*. These considerations have defined the research agenda within the AUTOMATED SOFTWARE ENGINEERING group at NASA Ames. We believe that verification techniques should be applied to the languages in use, and hence our current efforts have been directed towards JAVA and UML. As a more long term goal we have interest in applying verification techniques to higher level languages as well.

References

[1] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Computer-Aided Verification, CAV'98*, number

1427 in Lecture Notes in Computer Science, pages 319–331. Springer-Verlag, 1998.

[2] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A Tool for the Verification of Invariants. In *Computer-Aided Verification, CAV'98*, number 1427 in Lecture Notes in Computer Science, pages 505–510. Springer-Verlag, 1998.

[3] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Computer-Aided Verification, CAV'97*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[4] K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. Technical report, NASA Ames Research Center, California, 1997.

[5] K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer–Verlag, 1996.

[6] K. Havelund and N. Shankar. A Mechanized Refinement Proof for a Garbage Collector. *Formal Aspects of Computing*, 1998. Submitted for review.

[7] G. Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[8] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[9] B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith. Plan Execution for Autonomous Spacecrafts. In *Proceedings of the 1997 International Joint Conference on Artificial Intelligence*, 1997.