Formal Analysis of a Space Craft Controller using SPIN

Klaus Havelund[1], Mike Lowry[1] and John Penix[2*]

[1] NASA Ames Research Center, Recom Technologies, Moffett Field, California, USA.
Email: {havelund,lowry}@ptolemy.arc.nasa.gov
[2] Knowledge-Based Software Engineering Lab, University of Cincinnati, USA.
Email: jpenix@ececs.uc.edu

Abstract. This report 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 December 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 not yet resolved at the time of writing.

* John Penix spent 4 months during spring 1997 working at NASA Ames Research Center.

Table of Contents

1 Introduction

SPIN [2] 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 report 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) [6], an artificial intelligence based spacecraft control system architecture that is scheduled to launch in December 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 sub-component 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.

The report is attempted made self–contained in the sense that the reader is not assumed to be familiar with SPIN, nor with ESL and the RA Executive. Section 2 contains a condensed SPIN tutorial. Section 3 contains an informal description of the RA Executive, while section 4 describes its formalization in PROMELA. Section 5 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, chapters 6 and 7 contain the RA programming team's evaluation of the project, and our own conclusions respectively. Our own conclusions concern issues such as tool support for model building; and

PROMELA's capabilities seen as a specification notation. Appendix A contains the full PROMELA model.

Acknowledgements 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 6. 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.

2 A Condensed Spin Tutorial

In this section, we shall give a short presentation of the Spin system. Spin is a tool for analyzing the correctness of finite state concurrent systems with respect to formally stated properties. A particular concurrent system is formalized in the C-inspired Promela programming language, and properties to be verified are formalized in the temporal logic Ltl[3] (*Linear Temporal Logic*). The Spin tool provides a so-called *model checker*, which automatically can decide whether a Promela program satisfies an Ltl property. The Spin tool also provides a simulator, with which Promela programs may be executed in a step-by-step manner. This can in particular be used to re-run error traces generated by the model checker for properties that are *not* satisfied.

Although Promela is a kind of programming language, it can be used to formalize any concurrent system involving software, hardware and physical objects. As an example, the physical world that surrounds a space craft, and which can influence its behaviour, can be formulated as a Promela process, which spontaneously can execute and thereby change the system state. A Promela program can be said to denote a set of *reachable* states – the *state space*: those that can be reached from the initial state by executing the program. In order to allow for automatic verification, this state space has to be finite and small, and the Promela programmer must make sure this is the case by abstracting from real world complexity.

This presentation is supposed to be minimal in the sense that it should just make a novice able to read and understand the formalization of the RA Executive. We focus on the Promela language, since the Ltl logic is quite simple, and since the facilities of the Spin tool are less important for understanding the RA Executive formalization. The description is a highly condensed version of [2], with those language elements omitted that have not been used. Due to certain inconveniences in the Promela language – for example lack of procedural abstraction – we have extended the syntax slightly for presentation purposes. These extensions are also described. Hence, the presentation will be divided into three subsections, corresponding to the Promela language; the syntactic extensions; and the Ltl logic.

2.1 The Promela Programming Language

An executing Promela program consists of a collection of processes that communicate via buffered channels and shared variables. Figure 1 illustrates a program with two processes `P` and `Q`, which communicate over a channel `c` and via the shared variable `x`.

[3] In fact, Ltl formulae are translated into *test automata*, a more general specification formalism, which we shall, however, not use.

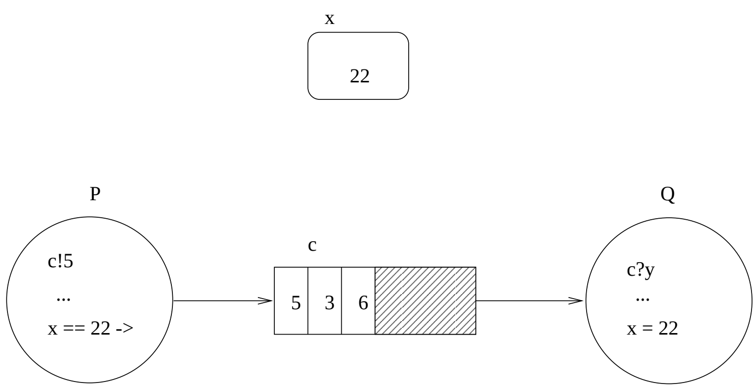

Fig. 1. Example of a PROMELA program

Process `P` sends values to `c` (`c!5`) and process `Q` reads values from `c` into the variable `y` (`c?y`). In addition, process `Q` assigns a value to `x` ("`=`" means assignment in PROMELA) and process `P` reads the value of `x` ("`==`" means equality test in PROMELA). The figure illustrates the 3 major components of a PROMELA program: variables, channels and processes. These will be described in turn below.

2.1.1 Datatypes and Variables The basic types over which variables can range are `int` (integers of 32 bits), `byte` (integers of 8 bits) and `bool` (one bit flags). The following declarations declare a byte variable `x`, two integer variables `i` and `j`, and a boolean variable `b`.

```
byte x;
int i,j;
bool b;
```

Variables are by default initialized to 0, and are updated by assignment statements like:

```
x = x + 1;
x++;
b = (x == 5)
```

Note that "`=`" means assignment in PROMELA, while equality is represented by "`==`". The first two statements have the same effect. Processes are basically made up of statements that are executed sequentially. Statements themselves can contain expressions, as `x + 1` and `x == 5` above. We shall return to statements and expressions below.

In addition to basic type variables, arrays of a fixed size can also be declared, as demonstrated by the following declaration, which declares an integer array named `state` of size 5.

```
int state[5];
```

the individual elements of this array are accessed as `state[0]`, `state[1]`, ..., `state[4]`. For example, the following statement updates the `state` variable in its first field by assigning to this the value of its last field incremented once.

```
state[0] = state[4] + 1
```

In addition to basic types and array types, PROMELA also supports record types. These have to be introduced in special `typedef` declarations like for example the following declaration of a record type, the elements of which are 2-field records, the first field named `status` and holding an integer, and the second field named `b` and holding a boolean array of size 5:

```
typedef Info{
  int status;
  bool b[5]}
```

Variables can now be declared of this type. The following two declarations declare a single variable and an array of this type, respectively.

```
Info i;
Info many[4]
```

The individual fields of a record variable are then accessed with dot-notation and can occur in expressions as well as on left-hand sides of assignment statements:

```
i.status
i.b[4]
many[3].status
many[3].b[4]
```

2.1.2 Channels A channel is a "first in first out" (FIFO) buffer capable of containing a specified maximal number of messages of a given type. Processes can communicate with each other by writing messages to, respectively reading messages from such buffers. The following declaration introduces a channel named `c`, capable of holding up to 10 messages, each of type `int`.

```
chan c = [10] of {int};
```

In the scope of this declaration, a process P can for example send a value to this channel by executing:

```
c!5
```

and another process can then read the buffer and store the result in the variable `y` as follows:

```
c?y
```

If a channel is full (contains 10 messages in the above example), then a send-statement (like `c!5`) will block. Similarly, if the channel is empty, a read-statement (like `c?y`) will block. A collection of operations are provided with which the status of a channel can be examined: `empty(c)` returns *true* if the channel is empty; `nempty(c)` returns *true* is the channel is non-empty, and `len(c)` returns the number of messages in the channel. Noting that non–equality in PROMELA is expressed as "`!=`", the operations `empty` and `nempty` can be expressed in terms of `len`:

```
empty(c)  == (len(c) == 0)
nempty(c) == (len(c) != 0)
```

As mentioned, `c?y` extracts the first element written to a channel (FIFO principle). The statement `c??y` extracts an arbitrary element from the channel, non-deterministically chosen. Furthermore, in case the channel `c` contains the message k (which must be a constant and not a variable), then `c??k` will extract that message, no matter where it is placed in the buffer (in case of several such messages, one will be extracted). Two test-operations exist to check the contents of a channel without retrieving its elements. Assume `k` is some constant (and not a variable), then `c?[k]` will return *true* if the first element inserted equals `k`, and `c??[k]` will return *true* if *some* element equals `k`.

2.1.3 Processes

Processes are declared using the `proctype` keyword. For example, assume that two integer valued channels `in` and `out` have been declared, then the following process – named `Add`, and parameterized with an integer `k` – will read a value from `in` into its local variable `x` and then output on `out` the result of adding `k`:

```
proctype Add(int k)
{ int x;
  in?x;
  out!(x + k)
}
```

A process of this type can then be spawned in a `run` statement as follows:

```
run Add(7);
```

Several processes can be spawned of the same type, even dynamically during program execution. Hence, process creation is dynamic in the sense that the number of processes that a program generates cannot generally be statically determined. Of course, if several processes are spawned of the above type, they will all communicate on the same channels, which may cause confusion. For that reason processes may be parameterized with channels, a technique we have however not used.

In general the body of a `proctype` declaration consists of a declaration part (introducing local variables) and a statement part describing the process's behaviour. In what follows, we shall explain what are possible statements.

2.1.4 Expressions and Statements

Expressions Statements are build from expressions. Expressions are either binary or unary. Binary expressions have the form $expr_1 \oplus expr_2$ where $\oplus$ is one of the operators: $+$ (*plus*), $-$ (*minus*), $>$ (*greater than*), $\geq$ (*greater than or equal*), $<$ (*less than*), $\leq$ (*less than or equal*), & and && (both representing *logical and*), and finally | and || (both representing *logical or*). In addition, the unary expression $!expr$ means "*not expr*".

In general, expressions may occur as statements, even though they have no effect on the state. In case an expression occurs as a statement, it is only executed if it evaluates to a value different from 0. Note that 0 represents $false$, and hence, an expression is *executable* as a statement only of it evaluates to $true$ (a value different from 0). This is typically used as a synchronization mechanism: a process can wait for an event to happen by waiting for some statement to become executable. For example if a process Q is supposed to assign the value 22 to a variable x, then another process P can wait for this to happen by attempting to execute the statement (expression):

```
x == 22
```

The statement `skip` in fact stands for the expression 1 and hence represents the always executable statement that has no effect.

Sequential Composition Statements are sequentially composed with semicolon "`;`". The arrow "`->`" has exactly the same meaning as "`;`", and is often used in connection with statements that may block, for example channel input statements, in order to indicate the blocking nature. For example, the following two statements have the same meaning:

```
c?x -> v = x + 1

c?x ;  v = x + 1
```

Conditional Statements The PROMELA equivalent to if-statements in traditional programming languages is illustrated by the following statement, which executes one of two statements `S1` or `S2` depending on the value of a variable `x` – `S1` is executed if `x` equals 0, and `S2` is executed if not:

```
if
:: x == 0 -> S1
:: x != 0 -> S2
fi
```

The general form of a PROMELA if-statement is a sequence of statements, each preceded by a double-colon:

```
if
:: stmt_1
:: stmt_2
...
:: stmt_n
fi
```

Only one of the statements is executed, and only such a one where the first sub-statement – called the guard (e.g. `x == 0` in the above example) – is executable. Which statement to be executed in case several have executable guards is non-deterministic. In case no guard is executable, the if–statement blocks (is not executable).

The above example can be written using instead the special `else` guard, as follows:

```
if
:: x == 0 -> S1
:: else -> S2
fi
```

The `else` branch will only be chosen in case none of the other branches can be chosen (none of the other guards are executable).

There is no restriction on the kind of statements that can be used as guards. As an example the following statement non-deterministically adds either 1 or 2 to the variable `x`:

```
if
:: x = x + 1
:: x = x + 2
fi
```

The following statement reads from channel `c` in case it is non-empty, continuing with `S` – otherwise it just terminates.

```
if
:: c?x -> S
:: empty(c)
fi
```

Iteration Quite similar to the if–statement, there is a do–statement, which basically works as the if–statement, except that its contents is repeatedly executed until the special `break` statement is executed. For example, the above if–statement that reads from a channel once can be turned into a loop that continues reading until the channel becomes empty:

```
do
:: c?x -> S
:: empty(c) -> break
od
```

The general form of the do–construct is:

```
do
:: stmt_1
:: stmt_2
...
:: stmt_n
od
```

Interrupts A special construct is provided for modeling interrupts, namely the unless–construct. It has the form:

```
S1 unless S2
```

with the semantics that statement `S1` is executed, step-by-step, to its end, *unless* the statement `S2` becomes executable, in which case the rest of `S1` is *immediately* ignored and `S2` is executed to its end. This can be used to monitor `S1`'s own operation, or it can be used to monitor and react to events coming from the surrounding environment. Note that the `unless` construct only works *within* a single process. That is, in the following piece of code `S` will never get executed, not even if it gets executable during the execution of `P`:

```
{run P()} unless S
```

Atomic Statements Normally, when given a sequence of statements `S1; S2; ...; Sn` within a process, their execution may be interleaved by the execution of statements within other processes, sometimes leading to undesired results. To avoid such undesired interleavings one can group the statements together with the `atomic` construct, indicating that they must be executed in one atomic transition, without the interleaving from other processes:

```
atomic{S1; S2; ...; Sn}
```

This construct can be used to model what is often referred to as *critical sections/regions*. Note that an outer `unless` construct may interrupt an `atomic` construct in the middle of its execution. Also, if an `atomic` construct blocks, then other processes are allowed to execute, and if the blocking condition eventually becomes *true* at some later point, the `atomic` construct may resume its execution, and continue atomically to its end, or until it gets blocked again. Note, however, it is not forced to resume immediately when the blocking condition becomes *true*.

Assertions PROMELA provides a construct – the assert–statement – for provoking the abortion of a program in case a certain property does not hold. The assert–statement has the form:

```
assert(boolean-expr)
```

It is always executable, and has no effect if the boolean valued expression is *true*. If on the other hand the expression evaluates to *false*, SPIN will produce an error report. Hence, this construct can be used to formulate certain correctness properties. For example, verifying a program with the following process will tell us whether `Q` will always receive positive numbers or not.

```
proctype Q()
{ do
  :: c?x -> assert(x > 0); S
  :: empty(c) -> break
  od
}
```

In case not, an error trace will be produced which explains the sequence of states that leads from the initial state to a state that breaks the assertion. Note that the SPIN model checker will examine all possible execution traces.

Initialization A PROMELA program must contain an initialization section, corresponding to the main section of a C–program. Typically, the `init`-statement will contain initialization of variables, and spawning of processes. For example:

```
init{
  x = 100;
  run P(x);
  run Q()
}
```

2.2 Macros and a Way to Avoid Them in Our Presentation

The source text of a PROMELA program is processed by the C preprocessor [3] for macro–expansion. Hence, C's general macro-definition language is available. This means that one for example can define a constant as follows:

```
#define MAX 10
```

and a procedure as follows:

```
#define swap(x,y) \
   int t; \
   t = x; \
   x = y; \
   y = t
```

Note that there is absolutely no typechecking performed on macro parameters. Also, local variables to macros are not really local, since the call of a macro will just expand the body. This results in a variable name clash when a macro is called twice in the same scope. Furthermore, the SPIN tool is not able to identify line numbers within macros, which makes finding syntax errors and bugs virtually impossible. Finally, it appears annoying to write "\" at the end of macro definition lines.

Since we find macros to be unpleasant as a programming notation, we have decided to "extend" PROMELA with procedures and functions, and some extra forms of constant and type definitions. This will allow us to present the RA formalization without the use of macros, in a notation closer to traditional programming notation. Note that these extensions are *not* supported by the SPIN tool, and are only introduced here as for presentation purposes. Appendix A contains the full model using the original macro-notation. The extensions can be regarded as a suggestion for extending PROMELA. Figure 2 shows the 5 extensions and their mapping into macro definitions.

Extension 1 is a simple constant definition. Extension 2 is a simple type equation, where a new name, here `Num` is introduced for an already existing

Number	New syntax	Corresponding macro definition
1	`const MAX = 10`	`#define MAX 10`
2	`type Num = byte`	`#define Num byte`

3	`type Ev = {A,B,C}`	`#define Ev byte`
		`#define A 0`
		`#define B 1`
		`#define C 2`

4	`procedure p(int x;wr bool b)` `{stmt}`	`#define p(x,b) stmt`
5	`function f(int x):bool` `{expr}`	`#define f(x) expr`

Fig. 2. Syntactic extensions to PROMELA

name, here `byte`. Such type equations are used to give more meaningful names to types. Extension 3 is an enumerated type definition, where the type `Ev` is defined to contain exactly the three elements `A`, `B` and `C`. The corresponding set of macro definitions introduce the type and the three constants. Finally, extensions 4 and 5 define procedures, respectively functions using C-like syntax. Note that parameter types are not translated. Also, note that we use the convention to prefix a parameter type with the keyword `wr` to indicate that it is modified by a procedure.

2.3 The LTL Property Language

A program may exhibit many execution sequences, depending on the non-determinism appearing in the program. Non-determinism may either be caused by non-deterministic constructs within a single process, such as an if–construct with several overlapping guards (that may all be executable), or it may be caused by the fact that several processes run in parallel and thereby may interleave in different ways. Consider for example the program:

```
int x;

proctype P()
{do
 :: x == 10 -> x = 0
 :: x < 10 -> x = x + 1
 od
}

proctype Q()
{do
```

```
  :: x == 1 -> x = x - 1
  od
}

init{run P(); run Q()}
```

The two processes may interleave in various ways, and two examples of (initial prefixes of) execution traces are shown in Figure 3, where states are shown as black circles, and the transitions between states have labels, each showing the statement executed, and which process that executes it.

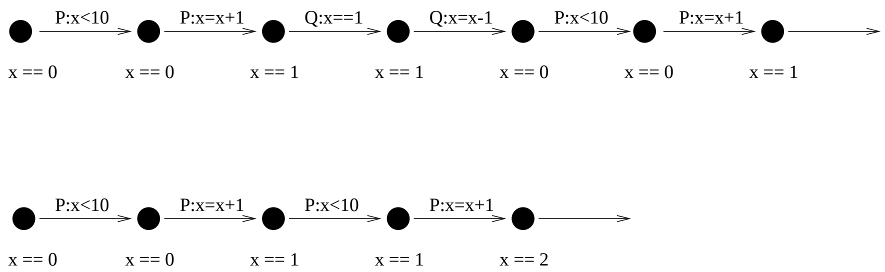

Fig. 3. Possible execution traces

SPIN provides a logic for stating properties about such execution traces. There are basically two kinds of formulae: "`[]P`" – meaning *always* `P`, and "`<>P`" – meaning *eventually* `P`. That is, a given execution trace satisfies the formula `[]P`, if `P` is *true* in *every* state of that trace[4]. Likewise, a given execution trace satisfies the formula `<>P`, if in *some* state of the trace, `P` is *true*[5]. These formulae can be nested. At the top–level, a program satisfies a formula, if *all* the program's execution traces satisfy the formula. The model checker of SPIN will in fact examine all possible traces.

Suppose for example that we now want to verify two properties: *that* `x` *is always non-negative*, and that *once* `x` *becomes strictly greater than* 1, *then eventually it will become* 10. These properties can be stated as follows[6]:

```
[] x >= 0

[] x > 1 -> <> x == 10
```

[4] Strictly speaking a trace satisfies `[]P` if *all suffix-traces* satisfy `P`.
[5] Strictly speaking, if *some suffix-trace* satisfies `P`.
[6] SPIN does in fact only allow macro calls to occur as arguments to the temporal operators, hence the predicates `x >= 0`, `x > 1` and `x == 10` must be named with the `#define` construct.

In fact, when applying the model checker to these properties, the first one is rejected, perhaps surprisingly to the reader, while the second is verified as being correct. The reason the first is rejected is, that process `Q` at some point may decide to decrement `x` since `x == 1`, but it waits with the decrement statement `x = x - 1` until after process `P` has increased `x` to 10 and there after reset it to 0. At that point `Q` executes `x = x - 1`, and `x` becomes -1. The SPIN tool discovers this, and yields an error trace, which can then be executed in the simulator in a step-by-step manner.

3 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 datatypes and the processes of the system.

3.1 Overview

The RA Executive, Figure 4, 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*.

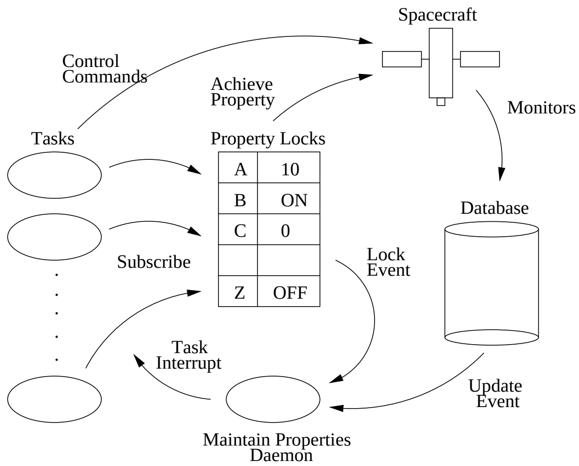

Fig. 4. Remote Agent Executive Resource Manager

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 uses *locks* to prevent tasks with *incompatible* 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.

3.2 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 *memory property*, has some value, called the *memory value*. For example, the following is a property:

$$\textsc{Camera}\ \textit{is}\ \textsc{On}$$

It states that the memory property CAMERA has the memory value ON. Hence, a property p is a pairing of a memory property mp and a memory value mv: $p = (mp, mv)$. 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 $p_1 = (mp, mv_1)$ and $p_2 = (mp, mv_2)$ are incompatible, if they have the same memory property (mp) but different memory values ($mv_1 \neq mv_2$). 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: $(p, subscribers, achieved)$[7].

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.

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

The Events Whenever the lock table or the database is changed, this is signalled 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 datastructures, the corresponding event lists are signaled, via the *signal* procedure, resulting in the waiting processes being restarted - for example the daemon.

3.3 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.

4 Formalization in PROMELA

In this section we present the PROMELA model of the RA Executive. The basic datatype 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.

Note that 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 SPIN's macro concept, here disguised as procedures and functions as explained in section 2.2. This choice has a drawback, namely that the SPIN simulator does not really support macros in a satisfactory way. More specifically, it cannot distinguish line numbers within a macro[8]. We have, however, chosen to stay as close as possible to the procedural structuring in the LISP program.

Note 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.

4.1 Modeling Lists

The fundamental datatype 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 datatype, implementing lists as arrays and defining the classical operations like *add an element*, *remove an element*, etc. as macros. 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

[8] In particular, when activating the simulator's "Time Sequence Panel" with the "One Window per Process" option, normally the currently executed code–line will be highlighted at each step. In case of a macro call, however, only the calling line will be highlighted throughout the macro's execution, and not its contents.

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 5. The type `list[Elem]` used as formal parameter type is supposed to represent all lists of elements of type `Elem`, the type `Elem` here supposed to be some polymorphic type.

```
procedure append(Elem e; wr list[Elem] l);

procedure remove(Elem e; wr list[Elem] l);

procedure copy(list[Elem] l1; wr list[Elem] l2);

procedure next(wr list[Elem] l, wr Elem x)
```

Fig. 5. Signatures for list operations

Informally, the procedures and functions do the following[9]. 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 6 illustrates the use of the list operations, and their effect on the variables `x`, `numbers` and `temp`. All statements execute, hence boolean valued expressions evaluate to *true*.

[9] 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.

```
                         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]     [3,2]
remove(3,temp);                         [2]
next(temp,x);            2              []
x == 2
```

Fig. 6. Examples of list operations

4.2 The State Space

Three constants define the bounds of the system, Figure 7. That is, they define the size of the state space, an important factor for obtaining efficient model checking.

```
const
  NO_PROPS = 2;
  NO_EVENTS = 2;
  NO_TASKS = 3;
```

Fig. 7. The constants

The constant `NO_PROPS` defines the number of memory properties, and hence the size of the property lock table and database, which each have an entry for each memory property. We shall work with two memory properties: 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 8. 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.

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

type
  Memory_Property = byte;
  Memory_Value = byte;

typedef Property{
  Memory_Property memory_property;
  Memory_Value memory_value};

typedef Lock{
  Memory_Value memory_value;
  list subscribers = [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 event_arg_test};

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

Fig. 8. Types

The type `Memory_Property` contains the memory properties, of which there are two (`NO_PROPS` = 2): 0 and 1. Correspondingly, the type `Memory_Value` contains the memory values. There is no constant defining the maximal number of memory 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 memory property and a memory value.

Now, as we shall see, the property lock table will be modeled as a mapping from memory properties to locks in the type `Lock`[10]. Hence each memory property is mapped to a record containing the following three fields: the memory 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.

[10] 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.

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 event_arg_test. 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 9. The database is represented by the variable db, which is an array mapping memory properties into memory values. The property lock table is represented by the variable property_locks, which is an array mapping memory properties into locks. In the LISP code, the property lock table is represented as a list of (memory property, lock) pairs. Hence, in the LISP program, the existence of a lock l on a memory property p is represented by the fact that the pair (p, l) is in the list. Since we model the property lock table as a mapping from memory properties to locks, the memory property p will *always* have an entry, and we therefore have to model the non-existence of a lock differently. We have reserved the memory value 0 for those locks that are "non-existent". That is, if a memory property maps to a lock with memory value 0, it means it is not locked (corresponding to not being in the list in the LISP program). The constant:

```
const
  undef_value = 0
```

is introduced to denote this undefined memory 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.

4.3 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_until[11], Figure 10, and signal_event, Figure 11.

The procedure wait_for_events_until takes three parameters: the parameter this (type TaskId) identifies the task that calls the procedure, and

[11] A procedure wait_for_events also exists, but it is very similar to wait_for_events_until.

```
Memory_Value
  db[NO_PROPS];
Lock
  property_locks[NO_PROPS];
Event
  Ev[NO_EVENTS];
Task
  active_tasks[NO_TASKS];
```

Fig. 9. Variables

```
procedure wait_for_event_until(TaskId this; EventId a; Property p)
{
atomic {
  append(this,Ev[a].pending_tasks);
  append(a,active_tasks[this].waiting_for);
  active_tasks[this].event_arg_test.memory_property =
    p.memory_property;
  active_tasks[this].event_arg_test.memory_value =
    p.memory_value;
  active_tasks[this].state = SUSPENDED;
  active_tasks[this].state == RUNNING }
}
```

Fig. 10. wait_for_event_until

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`[12], then it calls this procedure as follows: `wait_for_events_until(this,MEMORY_EVENT,CAMERA_ON)`. We shall refer to this property as the *restart condition*.

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 `event_arg_test` 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:

[12] That is, the memory property `CAMERA` must have the value `ON`.

```
procedure signal_event(EventId a)
{
TaskId t;
EventId e;
list pending = [NO_EVENTS] of EventId;
Ev[a].count = Ev[a].count + 1;
copy(Ev[a].pending_tasks,pending);
do
:: next(pending,t) ->
   if
   :: (active_tasks[t].event_arg_test.memory_value == undef_value ||
        db_query(active_tasks[t].event_arg_test) ) ->
      do
      :: next(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
}
```

Fig. 11. signal_event

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

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 25 page 39. 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 `event_arg_test` is satisfied it is restarted. The restart condition is satisfied, if either its memory 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].event_arg_test)
```

evaluates to *true*. The function db_query, Figure 12, takes as parameter a property p, and returns *true* if the database satisfies it (the property's memory property denotes the property's memory value).

```
function db_query(Property p):bool
{
db[p.memory_property] == p.memory_value
}
```

Fig. 12. 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 would result in an atomic construct around the body. This has not been done, however, since PROMELA at the time of writing does not allow nested atomic constructs, and since all signal_event calls occur within atomic constructs.

4.4 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 13, 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 memory property, but with a different, and therefore incompatible, memory value. Recall that if the memory property 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.

The procedure snarf_property_lock, Figure 14, is called by a task to snarf a property. The procedure takes as parameter the identity, this (type TaskId),

```
procedure fail_if_incompatible_property(Property p; wr bool err)
{
if
:: (property_locks[p.memory_property].memory_value != undef_value &
    property_locks[p.memory_property].memory_value != p.memory_value)
      -> err = 1
:: else
fi
}
```

Fig. 13. fail_if_incompatible_property

```
procedure snarf_property_lock(TaskId this; Property p; wr bool err)
{
atomic{
  fail_if_incompatible_property(p,err);
  append(this,property_locks[p.memory_property].subscribers);
  if
  :: property_locks[p.memory_property].memory_value == undef_value ->
      property_locks[p.memory_property].memory_value = p.memory_value;
      property_locks[p.memory_property].achieved = db_query(p)
  :: else
  fi;
  signal_event(SNARF_EVENT)}
}
```

Fig. 14. snarf_property_lock

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`.

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 memory property, but with a different memory value. Note that the result of this check is written into the `err` variable. In the calling context, Figure 20, 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 memory property of `p` is set to denote the memory 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 15, 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.

```
procedure find_owner(Property p; wr TaskId owner)
{
if
:: property_locks[p.memory_property].subscribers?[1] ->
     owner = 1
:: property_locks[p.memory_property].subscribers?[2] ->
     owner = 2
:: property_locks[p.memory_property].subscribers?[3] ->
     owner = 3
:: property_locks[p.memory_property].subscribers?[4] ->
     owner = 4
fi
}
```

Fig. 15. find_owner

When a task finally wants to achieve a property, it calls the procedure `achieve_lock_property`, Figure 16. 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 17 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_until` 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 17, 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[13]. Otherwise (`else`), in case the property is not already satisfied, a non-deterministic choice is made

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

```
procedure achieve_lock_property(TaskId this; Property p; wr bool err)
{
TaskId owner;
find_owner(p,owner);
if
:: owner == this ->
     achieve(p,err);
     property_locks[p.memory_property].achieved = true
:: else ->
     wait_for_event_until(this,MEMORY_EVENT,p);
fi
}
```

Fig. 16. achieve_lock_property

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.

```
procedure achieve(Property p; wr bool err)
{
if
:: db_query(p)
:: else ->
     if
     :: db[p.memory_property] = p.memory_value
     :: err = 1
     fi
fi
}
```

Fig. 17. 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 damon must intervene and stop the task if this is not the case. The procedure `closure`, Figure 18, 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 damon really explicitly and violently aborts the task.

```
procedure closure
{
if
:: true -> skip
:: true -> false
fi
}
```

Fig. 18. closure

Assume that the task now has called the `closure`, and 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 memory property but with different memory values. The releasing is done by a call of the procedure `release_lock`, Figure 19. It takes as parameter the identity, `this` (type `TaskId`), of the calling task; and the property, `p` (type `Property`), to be released.

```
procedure release_lock(TaskId this; Property p)
{
atomic{
  remove(this,property_locks[p.memory_property].subscribers);
  if
  :: empty(property_locks[p.memory_property].subscribers) ->
       property_locks[p.memory_property].memory_value = undef_value
  :: nempty(property_locks[p.memory_property].subscribers)
  fi }
}
```

Fig. 19. 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 memory property'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 memory value to the memory property in the table. Recall, that this is the way we model the absence of a lock (a memory property 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 `funcall_with_maintained_property`, Figure 20 – 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; the property, `p` (type `Property`), to be achieved and thereafter maintained to the end of the task; and finally the job, `c` (type `Closure`), to be executed.

```
procedure funcall_with_maintained_property(TaskId this;
  Property p; Closure c)
{
bool err = 0;

{ snarf_property_lock(this,p,err);
  achieve_lock_property(this,p,err);
  c
} unless {err || active_tasks[this].state == ABORTED};

active_tasks[this].state = TERMINATED;

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

Fig. 20. funcall_with_maintained_property

We have up until now seen the variable `err` occurring as result parameter to most of our procedures. 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 21, 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 28. 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 memory property 0 (`p.memory_property = 0`), and we arbitrarily let the task achieve a memory *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.

4.5 The Daemon

The daemon is responsible for detecting whether violations of locks occur in the database. That is, it must react in case a memory property mp in the lock table is locked to a memory value mv_1, and the corresponding `achieved` field is set to *true* (hence a task relies on it and is executing its job), but mp denotes a value $mv_2 \neq mv_1$ in the database. In that case the daemon must interrupt the tasks relying on the property (mp, mv_1) and repair the violation by updating the database, assigning mv_1 to mp again. The procedure `interrupt_task`, Figure

```
proctype Achieving_Task(TaskId this)
{ Property p;
  p.memory_property = 0;
  if
  :: this == 1 -> p.memory_value = 1;
  :: this == 2 -> p.memory_value = 2
  fi;
  funcall_with_maintained_property(this,p,closure)
}
```

Fig. 21. Achieving_Task

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

Fig. 22. interrupt_task

22, 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 20).

The procedure `lock_property_violated`, Figure 23, is used to determine whether locks have been violated. It is called for each memory property having an entry in the lock table (0 and 1 in our reduced case), and takes as parameter this memory property `mp` (type `Memory_Property`); 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 memory property is locked (memory value is defined), has been achieved, but has a memory value different from the one in the database.

The procedure `lock_property_violated` is called from the procedure `check_locks`, Figure 24, which checks the whole property lock table for violations. This is done in a loop that iterates over all the memory properties $\{0 \ldots \texttt{NO\_PROPS} - 1\}$. In fact, there are two such loops, each of the form:

```
mp = 0;
do
:: mp < NO_PROPS ->
     lock_property_violated(mp,lock_violation);
     if
     :: lock_violation ->
          HANDLE_VIOLATION
```

```
procedure lock_property_violated(Memory_Property mp;
  wr bool lock_violation)
{
  lock_violation =
    (property_locks[mp].memory_value != undef_value &
     property_locks[mp].achieved &
     db[mp] != property_locks[mp].memory_value)
}
```

Fig. 23. lock_property_violated

```
     :: else
     fi;
     mp++
:: else ->  break
od;
```

where the generic slot `HANDLE_VIOLATION` is executed for each violation. In a language with for–loops and traditional if–statements this could be formulated as:

```
For mp := 0 To NO_PROPS - 1 Do
  lock_property_violated(mp,lock_violation);
  If lock_violation Then
    HANDLE_VIOLATION
  End;
End;
```

In the first loop, in case of a memory property `mp` being violated (denoting something different in the database than in the lock table), all the subscribers to that memory property 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, `HANDLE_VIOLATION` is a `break`, meaning that the second loop terminates 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 `lock_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 behaviour found during verification, to be explained in section 5.4, we quote Erann Gat's explanation of the code:

> *The structure of this code is complicated by the design requirement that an external process may be responsible for restoring violated properties.*

```
procedure check_locks(wr bool lock_violation)
{
Memory_Property mp;
list sub = [NO_TASKS] of TaskId;
TaskId t;
mp = 0;
do
:: mp < NO_PROPS ->
     lock_property_violated(mp,lock_violation);
     if
     :: lock_violation ->
          atomic{copy(property_locks[mp].subscribers,sub)};
          do
          :: next(sub,t) -> interrupt_task(t);
          :: empty(sub) -> break
          od
     :: else
     fi;
     mp++
:: else ->  break
od;
mp = 0;
do
:: mp < NO_PROPS ->
     lock_property_violated(mp,lock_violation);
     if
     :: lock_violation -> break
     :: else
     fi;
     mp++
:: else -> break
od
}
```

Fig. 24. check_locks

(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 `Maintain_Properties_Daemon`, Figure 25, 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[mp] = mv` for each memory property `mp`, where the lock table maps `mp` to `mv`, but the database `db` does not.

```
proctype Maintain_Properties_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
}
```

Fig. 25. Maintain_Properties_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*, 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_until, Figure 10, 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 28, 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.

4.6 The Environment

Violations are introduced by the environment, here modeled by the process type Environment, Figure 26. 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 memory property 0 getting memory value 0. This will introduce a violation in case a lock has been created for memory property 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)
  }
};
```

Fig. 26. Environment

4.7 Initialization

All processes, the daemon and the tasks, are all instantiated with the procedure `spawn`, which takes as parameter the parameterized `task` (type `Process(TaskId)` represents[14] the type of processes parameterized with a task id) 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 `run` statement. Before that happens, the task's `state` field gets the value `RUNNING`.

```
procedure spawn(Process(TaskId) task; TaskId t)
{
atomic{
  active_tasks[t].state = RUNNING;
  run task(t)}
}
```

Fig. 27. spawn

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 28. 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.[15]

[14] Note that we use our own informal notation instead of macro definitions.

[15] 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. 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`.

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

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

Fig. 28. initialization

5 Analysis wrt. Selected Properties

5.1 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 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.

Each discovered error is illustrated by showing that fragment of the implemented code which causes the error to occur. Although the implementation is in Lisp, we shall for reasons of confidentiality present these program fragments in a pseudo-code notation invented for the purpose.

5.2 Error 1 – The Release Property

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

5.2.1 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 29 returns *true* if task `t` does not subscribe to memory property `mp`[16], hence has released it's lock on `mp`.

```
function not_subscriber(TaskId t; Memory_Property mp){
  (t == 1 -> !property_locks[mp].subscribers??[1]  :
     (t == 2 -> !property_locks[mp].subscribers??[2]  : true))}
```

Fig. 29. not_subscriber

To state the Release property, we modify the definition of the process `Achieving_Task`, Figure 21, adding an assert–statement after the call of `funcall_with_maintained_property`. This modification is shown in Figure 30.

```
proctype Achieving_Task(TaskId this)
{ Property p;
  p.memory_property = 0;
  if
  :: this == 1 -> p.memory_value = 1;
  :: this == 2 -> p.memory_value = 2
  fi;
  funcall_with_maintained_property(this,p,closure);
  assert(not_subscriber(this,p.memory_property))  -- assertion added
}
```

Fig. 30. Formalization of Release property

[16] Due to the semantics of Promela, it is necessary to write the body of the function as a conditional over `t`, since the arguments to the `??[]` operator have to be constants (and not names).

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

5.2.2 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 30. This implies a call of the procedure `funcall_with_maintained_property`, defined in Figure 20.
2. The procedure `funcall_with_maintained_property` 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 26, introduces an inconsistency in the database such that the memory value of memory property 0 becomes 0 in the database, while it is expected to be different from 0 by the running task.
4. The Daemon, Figure 25, detects this inconsistency and aborts the task in the `check_locks` procedure, Figure 24, by calling the procedure `interrupt_task` defined in Figure 22. That is, the status of the task becomes `ABORTED`.

The way the `funcall_with_maintained_property` 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 20):

```
{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.

The implemented version of `funcall_with_maintained_property` is sketched in Figure 31. It shows how the snarfing, achieving and closure all occur within a `Protect` construct, where the second argument after the `Exit` keyword is the statement `release_locks(locks)`.

A statement of the form "`Protect P Exit Q End`" executes `P` and then `Q`, 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. The snarfing is performed within a critical section, meaning that other tasks are blocked.

```
Procedure funcall_with_maintained_properties(props : Property list;
                                             closure : Task)
Begin
  Var locks : Lock list;
  Protect
    Critical
      locks := snarf_property_locks(props)
    End;
    achieve_lock_properties(locks);
    closure();
  Exit
    release_locks(locks)
  End
End
```

Fig. 31. Implemented version of `funcall_with_maintained_property`

5.2.3 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[17], 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 report.

5.3 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.*

5.3.1 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 memory property 0, assuming it to denote memory value 1 in the database (as stated in Figure 21) 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 32 shows two such predicates.

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

```
function task1_property_broken{
  (property_locks[0].memory_value == 1 &
   property_locks[0].achieved &
   db[0] == 0)}

function task1_terminated{
  (active_tasks[1].state == TERMINATED ||
   active_tasks[1].state == ABORTED)}
```

Fig. 32. Predicates used to formalize the ABORT property

The predicate `task1_property_broken` returns *true* in case of an inconsistency between `property_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 33. The property states that *"in all states (*`[]`*), if* `task1_property_broken` *holds, then eventually (*`<>`*), at some future point in time,* `task1_terminated` *will hold"*.

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

Fig. 33. 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 passed to `funcall_with_maintained_property` in Figure 21 is defined as in Figure 18. 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.

5.3.2 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 25, 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 21, starts; snarfs and achieves successfully, thereby signalling `SNARF_EVENT` from `snarf_property_lock`, Figure 14, 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 signalling 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 25 at the last `else` branch.
4. The environment, Figure 26, 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.

The implemented version of `Maintain_Properties_Daemon` is shown in Figure 34. We can point to the problem in the code: the decision to wait is taken in the second `If` construct, but some inconsistency may be introduced between this decision and the call of `wait_for_events`.

```
Procedure Maintain_Properties_Daemon();
Begin
  Loop
    If check_locks Then
      do_automatic_recovery
    End;
    If Not changed(event_count(memory_event) +
                   event_count(pl_snarf_event))
    Then
      wait_for_events(memory_event,pl_snarf_event)
    End
  End
End
```

Fig. 34. Implemented version of `Maintain_Properties_Daemon`

5.3.3 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 35 shows how the `Maintain_Properties_Daemon` has been extended with such a critical section around the code portion that decides whether to wait or not (the last if–statement).

```
proctype Maintain_Properties_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;
   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
   }
od
}
```

Fig. 35. Correction of `Maintain_Properties_Daemon`

Reapplying the SPIN model checker to verify the ABORT property formulated in Figure 33 for the modified model, however, shows that there is still an error in the system, as described in the next section.

5.4 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.

5.4.1 Formalizing The Property The property to be verified is as before, namely the one pictured in Figure 33.

5.4.2 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 35, 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 21, starts; snarfs and achieves successfully, thereby signalling `SNARF_EVENT` from `snarf_property_lock`, Figure 14, 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 35, has been awakened by the signalling of the `SNARF_EVENT`, and calls `check_locks`[18], Figure 24. 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.
4. The environment, Figure 26, 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 signalling 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.

The implemented version of `check_locks` is shown in Figure 36. One observes the two `ForEach` loops in between which the environment modifies the database. The first loop hence discovers nothing, while the second loop discovers the introduced inconsistency, and executes "`Return true`", with the result that `do_automatic_recovery` is called. Then `check_locks` is called again, but this time the inconsistency has been repaired and the first loop therefore discovers nothing, and no tasks are interrupted.

[18] In fact, `check_locks` is called twice, see section 5.6, and it's the second – and last – call which is referred to.

```
Function check_locks():bool
Begin
  ForEach lock In property_locks Do
    If lock_property_violated(lock) Then
      ForEach task In lock.subscribers Do
        interrupt_task(task)
      End;
    End;
  End;
  ForEach lock In property_locks Do
    If lock_property_violated(lock) Then
      Return true
    End;
  End;
  Return false
End;
```

Fig. 36. Lisp version of `check_locks`

5.4.3 Error Correction At the time when this error trace was generated, we believed that it was in fact an intended behaviour, and only later was it confirmed to be an unexpected and undesired behaviour – 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.

5.5 Error 4 – The Abort Property

5.5.1 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 behaviour. 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[19]. 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 37. This predicate returns *true* if the database and the lock table match wrt. to memory property 0 (recall that we have focused on task 1 that snarfs memory property 0).

The new correctness property using this new predicate is shown in Figure 38. The property states that *"in all states, if* `task1_property_broken` *holds, then eventually either* `task1_terminated` *or* `task1_property_repaired` *will hold"*.

[19] 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.

```
function task1_property_repaired{
  (property_locks[0].memory_value == db[0])}
```

Fig. 37. Additional predicate used to formalize the ABORT property

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

Fig. 38. Re-formalization of ABORT property

5.5.2 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 21, starts, and eventually calls `achieve_lock_property`, Figure 16. This procedure contains the two lines:

   ```
   achieve(p,err);
   property_locks[p.memory_property].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 26, introduces an inconsistency in the database such that the memory value of memory property 0 becomes 0 in the database, hence, destroys the just achieved property.
3. The daemon, Figure 35, 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 `lock_property_violated` Figure 23. 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.

The implemented version of `achieve_lock_property` is shown in Figure 39. It shows how the two lines marked are unprotected in the sense, that other code may execute in between them. To prevent this, they need to be embedded in a critical section, and this is what is demonstrated in the next section.

```
Procedure achieve_lock_properties(locks : Lock list)
Begin
  Var
    p : Property;
    owner : TaskId;
  ForEach lock In locks Do
    p := lock.property;
    owner := first_inserted(lock.subscribers);
    If owner = this_task Then
      achieve(p);            -- line 1
      lock.achieved := true  -- line 2
    Else
      memory-wait(p)
    End
  End
End
```

Fig. 39. LISP version of achieve_lock_property

5.5.3 Error Correction As stated, 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 40.

The SPIN model checker now certifies that the ABORT property in Figure 38 is satisfied in this new model.

5.6 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 Maintain_Properties_Daemon in Figure 25.

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 signalled by a call of signal_event, Figure 11, 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. In the implemented code, Figure 34, this corresponds to

```
procedure achieve_lock_property(TaskId this; Property p; wr bool err)
{
TaskId owner;
find_owner(p,owner);
if
:: owner == this ->
     atomic{  -- added
       achieve(p,err);
       property_locks[p.memory_property].achieved = true
     }
:: else ->
     wait_for_event_until(this,MEMORY_EVENT,p);
fi
}
```

Fig. 40. Correction of `achieve_lock_property`

the `changed`–expression to evaluate to *true*, hence causing an unintended extra call of `check_locks`. The RA programming team has confirmed this as an error, although one of low priority.

5.7 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.

6 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.

6.1 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 5, 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 5.

Error 1 – Release Property (section 5.2):

> *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 5.5):

> *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 5.3):

> *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 5.6):

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

Error 3 – ABORT Property (section 5.4):

> *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 mindset! Thanks for pointing this out.*

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

> *... internal joke ... 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.*

6.2 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.

7 Conclusion

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

7.1 Analysis of the Modeling and Verification 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 41. 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.

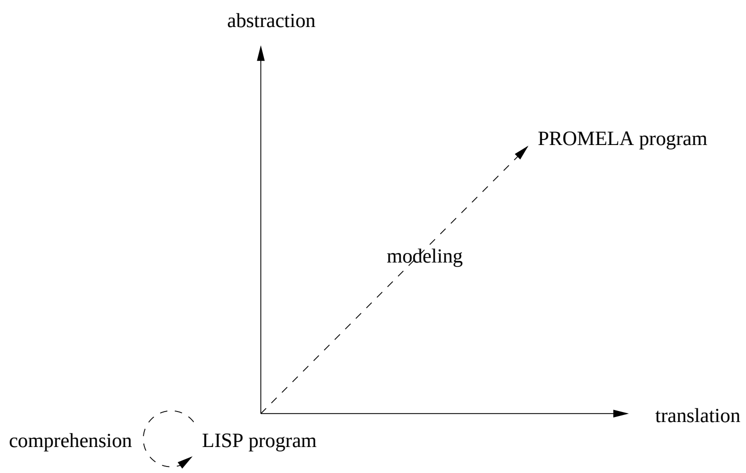

Fig. 41. Modeling = comprehension + abstraction + translation

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, as we shall return to, LISP is probably one of the most powerful languages around since it provides a combination of untyped functional programming and imperative object oriented programming, while the strongly typed PROMELA is probably weaker than most existing programming languages (for example, it does not have procedural abstraction). 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.

7.2 Tool Considerations

Even though manual translation was regarded harder than manual abstraction, we believe that translation can be mostly fully automated, 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 come with the programming language, as part of a debugging package. 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:

1. *Putting a maximal bound on number of iterations in a loop.*
2. *Limiting an infinite (or big) type to a finite (and small) subtype.*
3. *Changing the type of a variable, and changing all related operations.*
4. *Omitting, replacing, adding code.*

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. We imagine that the tool will allow the user to make arbitrary modifications to his program, and not just such 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.

Theorem proving can for example be used for this purpose, see [1]. Such proofs are very hard to create, however, 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.

7.3 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 (2). As an alternative to using SPIN, we considered the MURPHI model checker [4], which only allows for a static number of processes, although dynamic process creation can be modeled. At a higher level, an option is to use for example the PVS interactive theorem prover [5], which has a very nice and 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 [5]). A number of verification experiments combining theorem proving and model checking are described in [1].

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 can be suggested. First of all, the lack of procedural abstraction has been felt as a clear drawback. Macros can 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.

In [2] 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 has a limit on the number of processes allowed to be created, and this limit (256) is quickly reached in a program using a lot of procedural abstraction. The problem is, that in SPIN processes are not killed when they terminate. 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. That is, 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 [2]. We have not experimented with this solution.

Our general suggestion is an extension of PROMELA with procedures and functions, explaining them in terms of macros along the lines sketched in section 2.2. The hardest part of this work seems to be the treatment of local variables, parameter type checking, and adaption of the simulator (which for the moment cannot show the body of a macro). Local variables should not really be a problem since a unique static copy of each such variable can be made for each call, forbidding recursion that is.

Of further things one could want from PROMELA is: modules, nested atomic–constructs, enumerated types, type equations, and constant definitions. Generally, a complete avoidance of macro definitions would be preferable. Finally, on page 37 it is shown how a PROMELA loop of 12 lines can be written using 6 lines in a normal programming language having a for–loop and a traditional if–statement. Hence, why not have a modeling language supporting a general high level programming language? One could even consider an object oriented modeling language. One should mention MURPHI for its nice notation, supporting general datatypes, procedures and functions. MURPHI is rule oriented, however, a notation differing from traditional programming notation.

7.4 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 interleavings 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 not yet resolved at the time of writing. Furthermore, personnel (Erann Gat) in the RA team seems to have changed attitude drastically toward formal methods, from being *very skeptical* to being *believer*.

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*.

A PROMELA Model of the DEEP SPACE 1 RA Executive

This appendix contains the complete PROMELA model of the RA Executive. Backslashes "\" ending lines of macro definitions have, however, been removed for readability.

```
/***************************************/
/*                                     */
/*        PROMELA Model of the         */
/* DEEP-SPACE 1 Remote Agent Executive */
/*                                     */
/*          Klaus Havelund             */
/*            Mike Lowry               */
/*            John Penix               */
/*                                     */
/*     NASA Ames Research Center       */
/*                                     */
/*           August 5, 1997            */
/*                                     */
/***************************************/


/*********************/
/* System Parameters */
/*********************/

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

/*********************/
/* Boolean Constants */
/*********************/

#define false 0
#define true  1


/*********************/
/* EventId Constants */
/*********************/

#define MEMORY_EVENT 0
#define SNARF_EVENT  1
```

```
/*******************/
/* State Constants */
/*******************/

#define SUSPENDED  0
#define RUNNING    1
#define ABORTED    2
#define TERMINATED 3


/**************************/
/* Memory_Value Constants */
/**************************/

#define undef_value 0


/**********************/
/* Type Abbreviations */
/**********************/

#define TaskId byte
#define EventId byte
#define State byte
#define Memory_Property byte
#define Memory_Value byte
#define list chan


/********************/
/* Type Definitions */
/********************/

typedef Property{
  Memory_Property memory_property;
  Memory_Value memory_value};

typedef Lock{
  Memory_Value memory_value;
  list subscribers = [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 event_arg_test};


/********************************/
/* Global Variable Declarations */
/********************************/

Memory_Value db[NO_PROPS];
Lock property_locks[NO_PROPS];
Event Ev[NO_EVENTS];
Task active_tasks[NO_TASKS];
bool daemon_ready;


/*********************/
/* Lists as channels */
/*********************/

/* append(byte e; wr list[byte] x) */

#define append(e,x)  x!e


/* copy(list[byte] x; wr list[byte] y) */

#define copy(x,y)
  byte count,ce;
  count = len(x);
  do
  :: (count > 0) ->
     x?ce;
     x!ce;
     y!ce;
     count = count - 1
  :: (count == 0) -> break
  od


/* remove(byte e; wr list[byte] x) */
```

```
#define remove(e,x)
     assert(e <= 4);
     if
     :: e == 0 & x??[0] -> x??0
     :: e == 1 & x??[1] -> x??1
     :: e == 2 & x??[2] -> x??2
     :: e == 3 & x??[3] -> x??3
     :: e == 4 & x??[4] -> x??4
     :: else
     fi

/* next(list[byte] x; wr byte e) */

#define next(x,e) x?e

/* end lists as channels */


/*******************************************/
/* "Maintain_Properties_Daemon" Procedures */
/*******************************************/

/* wait_for_events(TaskId this; EventId a,b) */

#define wait_for_events(this,a,b)
  atomic {
     append(this,Ev[a].pending_tasks);
     append(this,Ev[b].pending_tasks);
     append(a,active_tasks[this].waiting_for);
     append(b,active_tasks[this].waiting_for);
     active_tasks[this].state = SUSPENDED;
     daemon_ready = 1;
     active_tasks[this].state == RUNNING
   }


/* wait_for_event_until(TaskId this; EventId a; Property p) */

#define wait_for_event_until(this,a,p)
  atomic {
     append(this,Ev[a].pending_tasks);
     append(a,active_tasks[this].waiting_for);
     active_tasks[this].event_arg_test.memory_property = p.memory_property;
     active_tasks[this].event_arg_test.memory_value = p.memory_value;
     active_tasks[this].state = SUSPENDED;
```

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


/* signal_event(EventId a) */

#define signal_event(a)
  TaskId t;
  EventId e;
  list pending = [NO_EVENTS] of {EventId};
  Ev[a].count = Ev[a].count + 1;
  copy(Ev[a].pending_tasks,pending);
  do
  :: next(pending,t) ->
     if
     :: (active_tasks[t].event_arg_test.memory_value == undef_value ||
          db_query(active_tasks[t].event_arg_test) ) ->
        do
        :: next(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


/* interrupt_task(TaskId t) */

#define interrupt_task(t)
  active_tasks[t].state = ABORTED


/* lock_property_violated(Memory_Property mp; result bool lock_violation) */

#define lock_property_violated(mp,lock_violation)
  atomic{
    lock_violation =
      (property_locks[mp].memory_value != undef_value &
       property_locks[mp].achieved &
       db[mp] != property_locks[mp].memory_value)
  }
```

```
/* check_locks(result bool lock_violation) */

#define check_locks(lock_violation)
  Memory_Property mp;
  list sub = [NO_TASKS] of {TaskId};
  TaskId t;
  mp = 0;
  do
  :: mp < NO_PROPS ->
       lock_property_violated(mp,lock_violation);
       if
       :: lock_violation ->
            atomic{copy(property_locks[mp].subscribers,sub)};
            do
            :: next(sub,t) -> interrupt_task(t);
            :: empty(sub) -> break
            od
       :: else
       fi;
       mp++
  :: else ->  break
  od;
  mp = 0;
  do
  :: mp < NO_PROPS ->
       lock_property_violated(mp,lock_violation);
       if
       :: lock_violation -> break
       :: else
       fi;
       mp++
  :: else -> break
  od


/* do_automatic_recovery() */

#define do_automatic_recovery
  bool locks_consistent;
  byte lock_counter;
  do
  :: lock_counter = 0;
     locks_consistent = true;
     do
```

```
    :: lock_counter < NO_PROPS ->
         if
         :: property_locks[lock_counter].achieved ->
              locks_consistent =
                locks_consistent &&
                (property_locks[lock_counter].memory_value == db[lock_counter])
         :: else
         fi;
         lock_counter++
    :: else -> break
    od;
    if
    :: locks_consistent -> break
    :: else ->
         if
         :: property_locks[0].achieved &&
            !(property_locks[0].memory_value == db[0]) ->
                db[0] = property_locks[0].memory_value;
         :: property_locks[1].achieved &&
            !(property_locks[1].memory_value == db[1]) ->
                db[1] = property_locks[1].memory_value;
         fi
    fi
  od


/*********************************************/
/* "with_maintained_properties" Procedures */
/*********************************************/


/* db_query(Property p) */

#define db_query(p)
  db[p.memory_property] == p.memory_value


/* fail_if_incompatible_property(Property p; result bool err) */

#define fail_if_incompatible_property(p,err)
  if
  :: (property_locks[p.memory_property].memory_value != undef_value &
      property_locks[p.memory_property].memory_value != p.memory_value) ->
        err = 1
  :: else
```

```
  fi


/* snarf_property_lock(TaskId this; Property p; result bool err) */

#define snarf_property_lock(this,p,err)
  atomic{
    fail_if_incompatible_property(p,err);
    append(this,property_locks[p.memory_property].subscribers);
    if
    :: property_locks[p.memory_property].memory_value == undef_value ->
         property_locks[p.memory_property].memory_value = p.memory_value;
         property_locks[p.memory_property].achieved = db_query(p)
    :: else
    fi;
    signal_event(SNARF_EVENT)
  }

/* achieve(Property p; result bool err) */

#define achieve(p,err)
  if
  :: db_query(p)
  :: else ->
       if
       :: db[p.memory_property] = p.memory_value
       :: err = 1
       fi
  fi


/* find_owner(Property p; result TaskId owner) */

#define find_owner(p,owner)
  if
  :: property_locks[p.memory_property].subscribers?[1] ->
       owner = 1
  :: property_locks[p.memory_property].subscribers?[2] ->
       owner = 2
  :: property_locks[p.memory_property].subscribers?[3] ->
       owner = 3
  :: property_locks[p.memory_property].subscribers?[4] ->
       owner = 4
  fi
```

```
/* achieve_lock_property(TaskId this; Property p; result bool err) */

#define achieve_lock_property(this,p,err)
  TaskId owner;
  find_owner(p,owner);
  if
  :: owner == this ->
       achieve(p,err);
       property_locks[p.memory_property].achieved = true
  :: else ->
       wait_for_event_until(this,MEMORY_EVENT,p);
  fi


/* release_lock(TaskId this; Property p) */

#define release_lock(this,p)
  atomic{
    remove(this,property_locks[p.memory_property].subscribers);
    if
    :: empty(property_locks[p.memory_property].subscribers) ->
         property_locks[p.memory_property].memory_value = undef_value
    :: nempty(property_locks[p.memory_property].subscribers)
    fi
  }


#define hang 0


/* closure() */

#define closure if :: true -> skip :: true -> hang fi


/* funcall_with_maintained_property(TaskId this; Property p; Closure c) */

#define funcall_with_maintained_property(this,p,c)
  bool err = 0;
  {
    snarf_property_lock(this,p,err);
    achieve_lock_property(this,p,err);
    c
  }
```

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


/*****************/
/* Task Spawning */
/*****************/

/* spawn(Process(TaskId) task; TaskId t) */

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

/*************/
/* Processes */
/*************/


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


proctype Maintain_Properties_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
};


proctype Achieving_Task(TaskId this)
{ Property p;
  p.memory_property = 0;
  if
  :: this == 1 -> p.memory_value = 1;
  :: this == 2 -> p.memory_value = 2
  fi;
  funcall_with_maintained_property(this,p,closure);
};


/******************/
/* Initialization */
/******************/

init
{
  spawn(Maintain_Properties_Daemon,0);
  daemon_ready == true;

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

References

1. 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.
2. G. Holzmann. *The Design and Validation of Computer Protocols.* Prentice Hall, 1991.
3. B. W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.
4. R. Melton, D.L. Dill, C. Norris Ip, and U. Stern. Murphi Annotated Reference Manual, Release 3.0. Technical report, Stanford University, Palo Alto, California, USA, July 1996.
5. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In Rajeev Alur and Thomas 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.
6. 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.