Formal Analysis of a Space Craft Controller using SPIN

Klaus Havelund[*], Mike Lowry and John Penix

NASA Ames Research Center
Moffett Field, California, USA

Email: {havelund,lowry,jpenix}@ptolemy.arc.nasa.gov
URL: http://ase.arc.nasa.gov

Abstract

This paper documents an application of the finite state model checker SPIN *to formally analyze a multi-threaded plan execution module. The plan execution module is one component of NASA's New Millennium Remote Agent, an artificial intelligence based spacecraft control system architecture which launched in October of 1998 as part of the* DEEP SPACE 1 *mission. The bottom layer of the plan execution module architecture is a domain specific language, named* ESL *(Executive Support Language), implemented as an extension to multi-threaded* COMMON LISP. ESL *supports the construction of reactive control mechanisms for autonomous robots and space crafts. For this case study, we translated the* ESL *services for managing interacting parallel goal-and-event driven processes into the* PROMELA *input language of* SPIN. *A total of* 5 *previously undiscovered concurrency errors were identified within the implementation of* ESL. *According to the Remote Agent programming team the effort has had a major impact, locating errors that would not have been located otherwise and, in one case, identifying a major design flaw. In fact, in a different part of the system, a concurrency bug identical to one discovered by this study escaped testing and caused a deadlock during an in-flight experiment 96 million kilometers from earth. The work additionally motivated the introduction of procedural abstraction in terms of inline procedures into* SPIN.

Index Terms – Program verification, concurrent programs, model checking, temporal logic, program abstraction, model extraction, spacecraft software.

[*] Recom Technologies

1 Introduction

SPIN [9] is a verification system that supports the design and verification of finite state asynchronous process systems. Programs are formulated in the PROMELA 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* automatically determines whether a program satisfies a property and, if the property does not hold, generates an error trace.

This paper documents an application of SPIN to formally analyze a software based multi–threaded *plan execution module* programmed in LISP, one component of NASA's Remote Agent (RA) [11], an artificial intelligence based spacecraft control system architecture. The Remote Agent also contains a *planning module*, which generates plans based on goals received from earth, and sends these plans to the plan execution module. A third module, the *mode identification and recovery module*, constantly monitors the state of the space craft and tries to recover in case of anomalies. The Remote Agent was one of 12 technologies tested on the DEEP-SPACE 1 space craft, launched October 1998. The Remote Agent itself was tested during May 1999, demonstrating the complete control of a space craft by artificial intelligence based software for the first time in NASA's history.

The bottom layer of the plan execution module is a domain specific language, named ESL (Executive Support Language), implemented as an extension to multi-threaded COMMON LISP. ESL supports 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. Our main focus was to analyze the ESL primitives used in the plan execution module. Hence, we created a small example plan execution model – with a fixed number of tasks all using constructs of the ESL, and then observed whether this *model* satisfied various desired properties.

The effort consisted of hand translating parts of the LISP code for ESL into a model in the PROMELA language of SPIN, and then verifying two properties formulated by the Remote Agent programmers. These properties were formulated after the PROMELA model had been created, hence they had no influence on the model design.

Both properties turned out to be broken, and a total of 5 flawed code fragments were identified, one breaking the first property, three breaking the second property, and one being a minor efficiency issue. This is regarded as a very successful result. According to the RA programming team, the effort has had a major impact, locating errors that they believe would not have been located otherwise and identifying a major design flaw. As an interresting aftermath, one of the five error patterns identified was mistakenly reintroduced in a different part of the plan execution module (not examined using SPIN), and caused a deadlock during flight in space. Because of the deadlock, thrusting did not turn off when required, and the space craft was unable to recover by itself. The craft was put in stand-by mode by the ground crew until a repair was made. The source of the error was identified as one of the flawed code patterns discovered by SPIN, and described in this paper. The repair consisted of introducing a critical section around the flawed code, the same solution we suggested based on the error trace generated by SPIN.

Section 2 contains a short introduction to SPIN and its modeling language PROMELA. Section 3 contains an informal description of the Remote Agent plan execution module, while section 4 describes its formalization in PROMELA. Section 5 presents the verification results, first stating the properties to be verified, and then 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, section 6 contains a discussion, including the RA programming team's evaluation of the project.

2 Introduction to SPIN

This section gives a short presentation of the SPIN system [9] with special focus on the PROMELA language constructs used in this paper. 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 as assertions in the program or as formulae in the temporal logic LTL (*Linear Temporal Logic*). The SPIN tool provides a so-called *model checker*, which automatically examines all program behaviors in order to decide whether a PROMELA program satisfies the stated properties. The SPIN tool also provides a simulator, with which PROMELA programs may be executed in a step-by-step manner. This tool can be used to re-run error traces generated by the model checker for properties that are *not* satisfied.

PROMELA 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 behavior, can be formulated as a PROMELA process, which can spontaneously execute and thereby change the system state. A PROMELA program can be said to denote a set of *reachable* states, called the *state space*, consisting of those states 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 of tractable size. The PROMELA programmer must make sure this is the case by abstracting from real world complexity, if necessary. Note, however, that even in the case where SPIN is not able to prove a property correct for all behaviors, it may still be able to locate errors if they show up early during the state exploration. Typically SPIN locates errors much faster than it proves correctness.

Promela Programs An executing PROMELA program consists of a collection of processes that communicate via buffered channels and shared global variables. At the top level, a program consists of a sequence of declarations of typed global variables and channels together with process declarations. There is also an initialization section constituting the main program in which all processes are started. A process is declared with the syntax:

`proctype` *name* `(` *arguments*`)` { *body*}

where the body is a sequence of local variable and channel declarations, and statements. Processes can be parameterized with values such as integers as well as with channels. The initialization section is regarded as a special unparameterized process declaration of the form:

`init`{ *body*}

A process `P` is started with the statement: `run P(...)`. Processes can be started dynamically at any point where a statement is allowed, within process bodies as well as in the `init` section.

Promela relies on macro definitions to introduce names for chunks of Promela text. One can either use the macro concept directly, for example when defining constants:

```
#define MAX 5
```

or one can define "procedures" using *inline* definitions:

```
inline f(x){c = c + x;x = x + 1}
```

The body of an inline is pasted into the body of a proctype at each point of invocation.

Variable Declarations Variables are declared using C-like syntax: a type followed by a name and an optional initial value, as for example in:

```
int x = 1;
```

Among the basic types are: `bool` (1 bit, a different name is `bit`), `byte` (8 bits), and `int` (32 bits). The basic datatypes include the standard literals and operators including: == (equal), `!=` (not equal), `!` (negation), `&` (and), `|` (or), and `&&` and `||` for the conditional versions of the latter two.

Promela has two kinds of composite types: arrays and records. Arrays must be given a size at declaration time, as in:

```
int a[5];
```

where an array `a` of integers of size 5 is declared. The elements of the array are accessed in the expected way, as in the assignment statement: `a[3] = a[3] + 1`. Record types are defined using the `typedef` keyword, as in the following definition of a record type `R` containing a boolean flag and an integer array of size 4:

```
typedef R{bool flag;int elements[4]}
```

A variable `r` can now be defined using the C-like notation: `R r`. The elements of a record are accessed using dot-notation, as in `r.elements[2] = 1`.

We shall use some notation for type abbreviations and enumerated types that is not supported directly by Promela, but which makes the presentation easier to follow. This notation is short for certain macro definitions as indicated here:

Notation:	Is short for:
`type Num = int`	`#define Num int`

`type Ev = {A,B}`	`#define Ev bit`
	`#define A 0`
	`#define B 1`

Channel Declarations 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 and 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, one process can for example send the value 5 to this channel by executing the statement `c!5`, while another process can receive this value in the variable `x` by executing `c?x`. If a channel is full (contains 10 messages in the above example), then a send-statement will block. Similarly, if the channel is empty, a read-statement will block. If the size of the channel is defined as 0, communication is by *rendez-vous*; the sending process blocks until a receiving process reads the value, and vice versa.

In addition to the channel reading statement `c?x` we shall mention a collection of more esoteric channel reading statements that we use. The statement `c?<x>` has the same effect as `c?x` except that the value stored in `x` is not removed from the channel. The statement `c??k` removes the oldest occurrence of the constant `k` from the channel, and blocks if there is no such constant. The statement `c??[k]` evaluates to *true* if `k` is in the channel, and $false$ otherwise. It does not block. Instead of `k` one can write `eval(exp)` for some expression `exp` to test whether the value of the expression occurs in the channel. Two operations are provided with which the emptiness of a channel can be examined: `empty(c)` returns *true* if the channel is empty; `nempty(c)` returns *true* if the channel is non-empty.

Statements Basic statements include assignment statements, such as `x = x + 1` or `x++`, and channel communication statements. The `skip` statement is a no-operation statement.

A statement can either be *executable* or *blocked* in a particular state. Two kinds of statements can block: channel communications as described above, and boolean expressions occurring as statements. Since a boolean expression is side-effect free and cannot communicate, its effect as a statement is equivalent to `skip`, except that it blocks if it evaluates to 0 ($false$). Hence, boolean expressions can be used to block a process if a certain condition on the state variables is not satisfied.

Statements can be composed sequentially, as in `s1;s2`, and can be grouped together using curly brackets: `{...}`. Note that curly brackets do not introduce a local scope for variables declared within them. A composed statement is executable if its first statement is executable.

The general form of a PROMELA if-statement is a sequence of statements, called *options*, surrounded by the keywords `if ... fi`, and each preceded by a double-colon:

```
if :: s1 :: s2 ... :: sn fi
```

Only one of the statements is executed, and only one where the first sub-statement – called the *guard* – is executable. When several statements have executable guards, the choice of the statement is non-deterministic. When no guard is executable, the if–statement blocks. The special `else` statement can be used at most once as the first sub-statement of an option, and it will become executable if all other options are non-executable. As an example, the following statement represents a traditional if-statement, using an arrow (->) as another way of writing sequential composition:

```
if
:: x == 0 -> x = 10
:: else -> x--
fi
```

Similar in syntax to an if-statement, a do–statement:

```
do :: s1 :: s2 ... :: sn od
```

behaves like the if-statement, except that it is executed repeatedly until the special `break` statement is encountered.

Interrupts are provided by the unless-statement, which has the form:

```
s1 unless s2
```

The statement `s1` is executed to its end unless statement `s2` becomes executable, in which case `s1` is aborted and `s2` continues executing. If `s2` does not become executable before the last sub-statement in `s1`, it will not be executed.

When two processes execute in parallel, their statements may execute interleaved in arbitrary order. In order to reduce this number of interleavings, statements within a process may be grouped together to execute in one indivisible step without interleaved execution from other processes as follows:

```
atomic{ s1; s2; ...; sn}
```

The purpose may be to enforce a correct behavior of the program, or alternatively to reduce the complexity of the system to obtain a smaller state space. The statements execute to the end unless one of them blocks, in which case control is transferred to another process. Control may continue from that point within the atomic construct if the statement becomes executable at a later point. The body of a process activated by a run-statement within an atomic construct is considered to be outside the scope of the atomic statement.

Specifying Properties SPIN provides two ways of defining properties, either by assertions placed as statements in the code, or by linear temporal logic (LTL) formulae defined separately from the code. The assert statement has the form:

```
assert( bool_expr)
```

It is always executable, and behaves as `skip` if the boolean valued expression is $true$. If the expression evaluates to $false$, SPIN will produce an error trace illustrating the execution path from the initial state to the state that violates the assertion.

An LTL formula states properties about the *execution traces* of the PROMELA program, where a trace is a sequence of *states*, each being an assignment of values to variables and channels. In general, a program denotes a set of execution traces, one for each possible interleaving of the processes in the program. Given a trace and some state in that trace, we shall define the *suffix trace* from that state to be the remaining part of the trace starting in that state.

Two kinds of temporal formulae are used in this paper: `[]`*P* , or *always P*, and <>*P*, or *eventually P*, where *P* is itself an LTL formula, the atomic case being a state predicate. A given execution trace satisfies the formula `[]`*P*, if *P* is $true$ in *every* state of that trace – strictly speaking, if *all suffix traces* of the trace satisfy *P*. Likewise, a given execution trace satisfies the formula <>*P*, if in *some* state of the trace, *P* is $true$ for the suffix trace starting in that state. These formulae can be nested. At the top–level, a program satisfies a formula if *all* the program's execution traces satisfy the formula. As an example, the formula:

```
[](request -> <>response)
```

states that whenever a request occurs then eventually a response occurs. Here `request` and `response` must be macro names, each representing a predicate on the state variables or a program control location.

LTL formulae are translated into so-called *never-claims* in Promela, which are equivalent to Büchi Automata. Never-claims can be written directly, providing a more general specification language, although less user friendly.

3 The Remote Agent Executive

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

3.1 Overview

The RA Executive, Fig. 1, is designed to support execution of software controlled *tasks* on board the space craft. A task may be, for example, to run and survey a camera. A task often requires that specific *properties* hold during its execution. 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 example camera–surveying task will try to turn on the camera before running the camera. Properties may, however, be unexpectedly broken (e.g., the camera may turn off) and tasks depending on such broken properties must then be informed about this, after which they can request to get the properties restored. In order to simplify the later modeling in PROMELA we shall, however, assume that the tasks in this case are *aborted*, hence completely terminated.

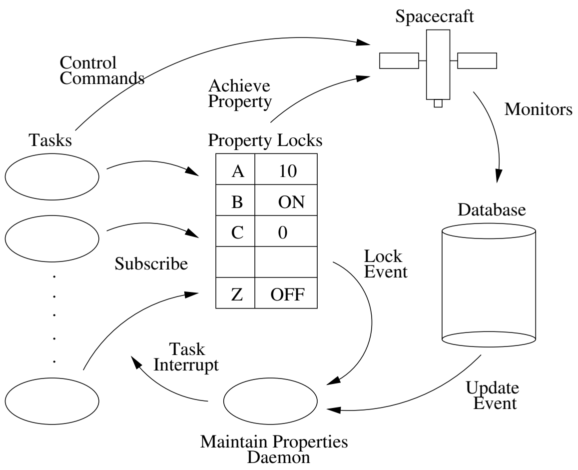

Figure 1. Remote Agent Executive

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* property requests from executing simultaneously. Two properties are incompatible if they require different observed values of the same equipment sensor. Executing concurrently with the tasks is a *property maintenance daemon* that monitors the database representing the status 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 remains inactive unless certain *events* occur, such as a change of the database or lock table.

The Executive permits various *achieve methods* to be associated with a property. When a task makes a request for a property to be achieved, the Executive calls the achieve method that is appropriate for the current situation. This aspect was not subjected to verification; the tasks were regarded as being able to achieve properties directly themselves.

3.2 Data Types

Property Database

The state of the spacecraft at any particular point can be considered as an assignment of values to a fixed set of variables, each corresponding to a component sensor on board the space craft. As an example, the variable CAMERA may have one of the values ON or OFF. A particular assignment of a value to a variable is called a *property*, where the variable is called the *property name* and the value is called the *property value*. Hence, a property p is a pairing of a property name pn and a property value pv: $p = (pn, pv)$. The fact that the camera is on can be written as the property: (CAMERA,ON).

The actual state of the space craft is constantly monitored, and stored in a *database*. We shall assume that, at any moment, the database will contain a set of monitored properties that are consistent with the actual state of the spacecraft.

The Property Lock Table

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

An inconsistency occurs between the database and the locks 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. If such an inconsistency occurs, the daemon suspends all tasks subscribed to the property.

Events

Whenever the lock table or the database is changed, the daemon must be awakened so that it can examine the new system state. In general, application tasks may also wait for such events to happen. For this purpose, event objects are introduced for each kind of event in the system: LOCK_EVENT (representing a change of the lock table) and MEMORY_EVENT (representing a change of the database). Any process (task or daemon) wanting to wait for an event calls a *wait* procedure, which adds the process to a corresponding list of waiting tasks. Whenever a change happens to the lock table or the database, the corresponding event is signaled, via a *signal* procedure, and the waiting processes are removed from the list and restarted.

3.3 Processes

Tasks

Before a task executes its main job, it will try to achieve the properties that the execution depends on. The first step is to lock the properties in the lock table. Locking a property will only succeed if it is compatible with the existing locks; otherwise, the task is aborted. If there are no conflicting locks and the lock does not already exist, the task will create it. Note that some other task may have already locked the same property, which is not defined as a conflict. If it succeeds, the task also puts itself into the subscribers list of the lock, indicating that the 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 indicates the property was successfully achieved. Hence, the *wait* procedure takes the property to be waited for as argument in addition to the event waited for.

Once a task has first locked 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 if the list 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 system contains a daemon to guarantee that achieved properties are maintained while subscribing tasks are executing. An achieved property in the property lock table is said to be *maintained* as long as it is also contained in the database (and hence is a property of the space craft).

The daemon is normally in "sleep mode", waiting for an event that modifies the database (MEMORY_EVENT) or the property lock table (LOCK_EVENT). Sleep mode is implemented by letting the daemon wait in the corresponding event lists. Once started, the daemon examines all locks in the property lock table. 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 aborted and a recovery procedure is initiated to re-achieve the property. After examining all locks, the daemon goes into sleep mode again by waiting for another MEMORY_EVENT or LOCK_EVENT.

4 Formalization in PROMELA

Our discussion of the the PROMELA model of the RA Executive focuses on operations of the lock table. The basic data type of LISP is the list, so we begin our exposition by outlining how we modeled lists in PROMELA. The presentation is then divided into the following topics: the state space (constants, types and global variables), operations on events, the tasks, the daemon, the environment that may introduce violations, and initialization of the system state.

The LISP program that was modeled in PROMELA is highly structured using procedural abstraction, and hence is divided into a collection of relatively small-sized procedures and functions. We tried to maintain the same level of structuring in the PROMELA code using the inline and macro concepts. All communication between processes takes place via *shared variables*, since this is how the LISP implementation works. PROMELA channels are only used to represent lists, as described in the next section.

As stated above, the PROMELA model focuses on operations on the lock table. Hence, it is an abstraction of the LISP program, omitting details not regarded as important for the lock table operations. The LISP program is approximately 3000 lines of code while the PROMELA model is 500 lines of code. Furthermore, the model only deals with a limited number of tasks and properties in order to limit the search space the SPIN model checker has to explore. Most abstractions are made in an informal manner without any formal proofs showing that bugs are maintained. Hence, in the abstraction phase we may have left out errors in the LISP code. However, the errors in the model that we are about to describe were also errors in the LISP code.

4.1 Modeling Lists

The fundamental data type in LISP is the list. Lists are used heavily in the program, and hence we tried to find a convenient way to represent them in PROMELA. One solution is to define an abstract data type implementing lists as arrays and defining the classical operations like *add an element*, *remove an element*, etc. as macros (or inlines in the newest version of SPIN). We did not take this approach, due to an early attempt to avoid macros since they are not well integrated into SPIN; for example, they do not support local variables 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 briefly describe the technique. First, with the macro definition:

```
#define list chan
```

we define a new symbol `list` to stand for the symbol `chan`, the PROMELA keyword for declaring channels. With this definition we can 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. The signatures for a number of operations defined upon lists are shown in Fig. 2.

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

Figure 2. Signatures for list operations

Informally, the procedures and functions do the following: 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 and stores it in the result variable `x`, assuming the list is not empty[1]. Suppose we have the following declarations:

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

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

```
                         x   numbers    temp

                         0      []       []
append(1,numbers);             [1]
append(2,numbers);            [2,1]
append(3,numbers);           [3,2,1]

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

Figure 3. Examples of list operations

4.2 The State Space

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

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

Figure 4. The constants

The constant `NO_PROPS` defines the number of property names. This constant determines the size of both the property lock table and the database, because they each have an entry for each property name. We chose to work with two property names: 0 and 1. The constant `NO_EVENTS` defines the number of events, in our case 2: `MEMORY_EVENT` and `LOCK_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.

The constants `NO_PROPS` and `NO_TASKS` are chosen small in order to obtain a small state space for the model checker to explore. This is an example of an abstraction that is not formally justified. Hence, some errors in the LISP program may be left out. On the other hand, any errors

[1] 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, and hence the implementations of these procedures are sometimes more elaborate.

found in this reduced system will also be errors in the real system, and since our goal is to identify errors rather than to prove correctness, the abstraction is regarded as acceptable.

A number of types were defined to model the data structures in the system, Fig. 5. 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 ($\mathtt{NO\_TASKS} = 3$): the daemon, which is given identity 0 and two application tasks, given identity 1 and 2 respectively.

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

type
  Property_Name = byte;
  Property_Value = byte;

typedef Property{
  Property_Name name;
  Property_Value value};

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

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

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

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

Figure 5. Types

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

The property lock table is modeled as a mapping from property names to locks in the type `Lock`. Hence each property name is mapped to a record containing the following three fields: the property value it is supposed to have; the list of tasks subscribing to the lock; and a flag indicating whether it has been achieved or not.

Each event (`MEMORY_EVENT` and `LOCK_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 and a list of pending tasks waiting for the event. 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 the events it is waiting for, in case the state is `SUSPENDED`; and a property called `prop`. This property represents a condition that must be satisfied before the task can be re-started by an event signal. It is relevant when the task is not the owner of a lock and some other task is supposed to achieve the property. In this case, the task must wait for this property to be achieved, and so the property becomes the `prop` condition.

The state space of the model is declared in Fig. 6. The database is represented by the variable `db`, which is an array mapping property names into property values. The property lock table is represented by the variable `locks`, which is an array mapping property names into locks. In the LISP code, the property lock table is represented as a list of (property name, lock) pairs, and the existence of a lock l on a property name pn is represented by the fact that the pair (pn, l) is in the list. An array is used to model this list, instead of a channel as described in section 4.1, because the elements are composite record values.

Since we model the property lock table as a mapping from property names to locks, the property name pn will *always* have an entry, and we must model the non-existence of a lock differently. We have reserved the property value 0 for those locks that are "non-existent". That is, if a property name maps to a lock with property value 0, it means the property is not locked (corresponding to not being in the LISP list). The constant:

```
#define undef_value 0
```

is introduced to denote this undefined property value.

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

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

Figure 6. Variables

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`, Fig. 7, and `signal_event`, Fig. 8.

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

Figure 7. wait_for_event

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

Figure 8. signal_event

The procedure `wait_for_event` takes three parameters. The parameter `this` (type `TaskId`) identifies the task that calls the procedure, that is, the task that wants to wait for an event to happen; the parameter `a` (type `EventId`) identifies the event to be waited for; and 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)`, it calls the procedure as follows: `wait_for_event(this,MEMORY_EVENT,CAMERA_ON)`. This property is referred to as the *restart condition*.

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

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

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

A procedure `wait_for_events` also exists which is very similar to `wait_for_event`, except that it takes several event parameters, and waits for any one of these to happen.

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

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

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

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

Figure 9. db_query

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

In the case where 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 the LISP code, the body of the signal_event procedure is embedded within a critical section. A direct modeling of this in PROMELA results in an atomic construct around the body.

4.4 The Tasks

Tasks are modeled as PROMELA processes. A collection of support procedures were used to define tasks. The procedure fail_if_incompatible_property, Fig. 10, is called by a task just before it tries to lock a property, to check if the lock conflicts with existing locks. The procedure takes as parameter the property p (type Property) to be locked, and returns $true$ if some other task has already locked the property name, but with a different, and therefore incompatible, property value. The result of this test is stored in the return variable err, which is used to direct control in the calling context.

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

Figure 10. fail_if_incompatible_property

The procedure lock_property, Fig. 11, is called by a task to lock a property. The procedure takes as parameters the identity, this (type TaskId), of the calling task; and the property, p (type Property), to be locked. The outcome of the operation is written into the result variable err.

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

Figure 11. lock_property

The procedure first checks whether the request is compatible with the already existing locks. That is, there must not be a lock with the same property name, but with a different property value. The result of this check is written into the err variable. In the calling context, Fig. 17, we shall later see the effect of this result variable becoming $true$: an abort will occur and terminate the task. The task is then appended to the list of subscribers to the property. Then, if the property is not already in the lock table, it is "inserted": the property name of p is set to denote the property value of p, and the achieved field is set to reflect whether the property already holds in the database via db_query. Finally, the LOCK_EVENT is signaled to restart the daemon in case it is waiting.

After locking the property, if the task is the owner, it must achieve the property. A task is the owner of a property if it was the first to subscribe to the property, and hence is the first element in the property's subscriber list in the lock table. The procedure find_owner, Fig. 12, determines the owner of the property p. The brackets <...> are special PROMELA notation which prevents the owner from being removed from the list.

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

Figure 12. find_owner

When a task wants to achieve a property, it calls the procedure achieve_lock_property, Fig. 13. The task can only achieve the property if it is the owner. Hence, it first determines which task is the owner of the property p. In case the owner equals the calling task, the property is achieved by a call of the procedure achieve (defined in Fig. 14) and the achieved field is set to $true$. On the other hand, if the task is not owner, it must wait for the owner to achieve the property. This wait is initiated by a call to `wait_for_event` with the property `p` as the restart condition. That is, the calling task will only be restarted on a memory event if the property `p` is satisfied in the database, and hence has been achieved.

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

Figure 13. achieve_lock_property

The procedure `achieve`, Fig. 14, actually achieves the property by updating the database. If the property `p` is already satisfied in the database, the procedure returns successfully. Otherwise, a non-deterministic choice is made between success, modelled by updating the database to achieve the property, and failure, modelled by setting `err` to $true$. This non-determinism reflects the fact that achievement can fail, and abstracts away the details about the possible causes of failure.

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

Figure 14. achieve

Once the task has achieved the property, it is ready to execute its real job while assuming that the property is maintained; the daemon must intervene and stop the task if this is not the case. The procedure `closure`, Fig. 15, represents the job of the task. Its body is simply a non-deterministic choice between `skip` and `false`. In case `skip` is executed the procedure returns immediately. In case the second option is executed, the `false` statement will make the calling task block. This blocking simulates a time consuming computation, and is used later to formulate a correctness condition: *in case the property is broken (i.e.: is no longer in the database), the task will be terminated.* If `closure` always terminated, this property would be trivially satisfied; hence the blocking alternative, allows us to verify that the daemon actively aborts the task.

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

Figure 15. closure

After a task finishes or has been aborted, it no longer needs to lock the property. Hence, our task must release the property, meaning that the task must be removed from the property lock table. This will allow other tasks to lock the same property name but with different property values. The releasing is done by a call to the procedure `release_lock`, Fig. 16. The body of `release_lock` is embedded within an `atomic` to model a critical section in the LISP code. It removes the task from the property name's subscriber list in the lock table. If this action causes the subscriber list to become empty, the lock must be removed completely from the lock table. Removal of the lock is modelled by assigning the `undef_value` as property value to the property name in the table.

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

Figure 16. release_lock

The top-level procedure `execute_task`, Fig. 17, can now be defined. When called by a task, this procedure locks the property `p` to be maintained, achieves `p`, executes the job, and finally releases `p` again.

The variable `err` is passed as a result parameter to most of the procedures. This variable is a local variable of `execute_task`, and is passed as a parameter to the procedures `lock_property` and `achieve_lock_property`. The calls of these two procedures are embedded within an `unless` construct of the form

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

  active_tasks[this].state = TERMINATED;

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

Figure 17. execute_task

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

where the *condition* is that either `err` is $true$, or the task is aborted: `active_tasks[this].state == ABORTED`. As we shall see in the next section, the daemon aborts a task by assigning `ABORTED` to the `state` field in the tasks status record. Thus, locking, achieving, and job are performed to the end, unless the condition becomes $true$, in which case the whole statement terminates.

Once locking, achieving and job have terminated, either normally or abnormally, the statement:

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

executes as part of modeling the LISP `unwind-protect` construct [12]. The assignment restores the value of the `state` field in case the task was aborted by the daemon. The last statement in `execute_task` releases the property from the lock table, but is abruptly terminated if the `state` field gets assigned the value `ABORTED`, which occurs if the daemon discovers a property violation at this point.

The process type `Achieving_Task` used to instantiate tasks is defined in Fig. 18. It assigns the property to be acheived to the local variable `p`. In order to reduce the state space, we focus on property name 0 and we arbitrarily let the task achieve a property *value* which is identical to `this`, which is the task's identity. Finally, the main procedure is called. Note that all tasks in this model perform the same job (`closure`). This is an example of an abstraction from the LISP code, where details regarded as unimportant for the verification have been omitted.

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

Figure 18. Achieving_Task

4.5 The Daemon

The daemon is responsible for detecting whether violations of locks occur in the database. It must react when a property name pn in the lock table is bound to a property value pv_1 and the corresponding `achieved` field is $true$, indicating that an executing task relies on the property, but in the database pn denotes a value pv_2 and $pv_2 \neq pv_1$. In this case the daemon must abort the tasks relying on the property (pn, pv_1) and repair the violation by updating the database so that pn denotes pv_1 again. The procedure `abort_task`, Fig. 19, aborts a task `t` by assigning the value `ABORTED` to the `state` field of its status record, which causes the `unless` construct in `execute_task` to terminate the task process (Fig. 17).

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

Figure 19. abort_task

The procedure `property_violated`, Fig. 20, determines whether locks have been violated, assigning $true$ to the result variable if the property name is locked and has been achieved, but the database binds the property name to a different value.

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

Figure 20. property_violated

The procedure `property_violated` is called from the procedure check_locks, Fig. 21, which checks the whole property lock table for violations. The first loop in check_locks aborts the subscribers to property pn if pn is violated. In the LISP program, the subscribers to the property are not automatically aborted, instead they are informed about the violation, where after they decide whether or not to abort. If all the subscribing tasks abort, the property does not need to be recovered. If, on the other hand, some of the tasks decide not to abort, the broken property has to be recovered. The second loop in check_locks determines if there are any violations left. A break statement terminates the loop if a violation is found. The result returned in the variable lock_violation will be used in the calling context to decide whether the database should be recovered. Since the PROMELA model aborts all subscribing tasks in the first loop, the second loop should not be needed. However, an error was found in the model which reflected an error in the LISP code, so we maintain this feature of the model in this presentation.

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

Figure 21. check_locks

The daemon process is an instance of the process type Daemon, Fig. 22, which declares three local variables: lock_violation holds the result of check_locks, event_count keeps track of new events, and first_time indicates whether the daemon has just started. The body consists of an infinite loop. On each iteration, the daemon calls check_locks to determine if there are any violations. If there are violations, it calls do_automatic_recovery to repair the database by making it consistent with the lock table. Next, the daemon decides whether to stop and wait for a new memory event or lock event, or whether to start a new iteration. Another iteration is needed if a memory event or a lock event has occurred since the last time the daemon was restarted. If first_time is $true$ (initial iteration), the daemon simply waits for either a MEMORY_EVENT or a LOCK_EVENT to occur. The procedure wait_for_events is similar to wait_for_event, Fig. 7, except that it waits for one of *two* events and it sets a boolean variable daemon_ready to $true$ just prior to waiting. If first_time is false, and the expression

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

Figure 22. Daemon

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

is $true$, then there has been an event since the last time event_count was updated.

4.6 The Environment

A PROMELA model is generally closed, and hence the environment has to become part of the model. In our case, violations are introduced by an instance of the process type `Environment`, Fig. 23, which runs in parallel with the tasks and daemon, and may cause a database change at any moment in time. The environment assigns the value 0 to property name 0, introducing a violation if a lock has been created for property name 0 with a value different from 0. It then signals `MEMORY_EVENT` to wake up the daemon, in case it's not already running.

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

Figure 23. Environment

4.7 Initialization

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

Figure 24. spawn

All processes are instantiated with the procedure `spawn`, which takes as parameters the parameterized `task` (a `proctype`) to be spawned and the identity `t` (type `TaskId`) of the task to be spawned. The system is initialized by spawning the daemon, the two tasks, and the environment, Fig. 25. Before the tasks are spawned, the daemon must terminate its own local initialization, modelled by waiting for the variable `daemon_ready` to become *true*. In an early model, the tasks were spawned without waiting for the daemon, but that led to an error which was discovered by the model checker, as explained in Section 5.7.

5 Analysis wrt. Selected Properties

5.1 Identifying Properties to be Verified

The model has been analyzed with respect to the following two properties:

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

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

Figure 25. initialization

RELEASE Property: *A task releases all of 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 the following we demonstrate how we formulated these properties in terms of PROMELA assertions and LTL formulae, and we show the results of applying the SPIN model checker to verify these properties. It turned out that none of the properties were satisfied in the presented model, and that the model was accurate in the sense that all of these errors also existed in the LISP code. Their discovery led the RA programmers to make appropriate code modifications, hence improving the code quality. When we discuss fixes to some of these errors, they apply to the PROMELA model, but in most cases similar fixes were also made to the LISP code .

The attempted verification of the two properties led to the direct discovery of four programming errors – one breaking the RELEASE property, and three breaking the ABORT property. All of these errors are classical concurrency errors in the sense that they arise due to processes interleaving in unexpected ways. For example, two errors can be corrected by introducing additional critical sections. Furthermore, a less serious efficiency error (code executing twice instead of once) was discovered by examining traces generated during model checking. Hence, a total of five errors were identified in the LISP code, four of which were considered important. Additionally, the analysis highlighted the need for a mechanism to insure that the daemon reaches "steady state" before proceeding.

5.2 Error 1 – The RELEASE Property

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

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

Figure 26. RELEASE **predicate**

To state the RELEASE property, we modify the definition of the process Achieving_Task, Fig. 18, adding an assert–statement after the call of execute_task, as shown in Fig. 27. When a task terminates, it must no longer subscribe to the property name that it locked.

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

Figure 27. Formalization of RELEASE **property**

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

1. A task starts, running process Achieving_Task in Fig. 27. This implies a call of the procedure execute_task, defined in Fig. 17.

2. The procedure execute_task does the locking, the achieving, and the closure call. It then changes its state to TERMINATED and is ready to release its lock (call the release_lock procedure).

3. At this point, the Environment, Fig. 23, introduces an inconsistency in the database.

4. The Daemon, Fig. 22, detects this inconsistency and aborts the task in the check_locks procedure, Fig. 21, by calling the procedure abort_task defined in Fig. 19. That is, the status of the task becomes ABORTED.

The execute_task procedure (Fig. 17) exits when the state becomes ABORTED, skipping release_lock. Hence, even though the locking, achieving, and closure are protected against aborts, the lock releasing itself is not. This model reflects how the corresponding LISP construct in the real system: "(unwind-protect E1 E2)" executes E1 and then E2 (the lock releasing), with the addition that if an abort occurs during the execution of E1, the remainder of E1 is skipped, and E2 gets executed. The unexpected situation is that an abort can occur during the execution of E2, with the result that the rest of E2 will not be executed.

Normally this semantics is sufficient for sequential programs where an abort typically is thrown only within E1, something that can be verified by code inspection. However, in a concurrent setting as here, such an abort can be thrown from a parallel process at any time, hence also during the execution of E2. Admittedly this error could have been detected by code inspection when the PROMELA model was created. However, the properties to be verified had not yet been formulated at that point in time, so the problem was therefore not in our focus.

The programmer's response was: *"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".*

The error can be corrected by protecting the lock releasing from being aborted. This was done in a modified version of the PROMELA model by removing the unless construct attached to the call of release_lock. Thereafter, the model was verified to satisfy the RELEASE property using SPIN. However, since our correction was not easily implementable in LISP, we continued with the original model. The remaining verification results are based on a model still containing this error, but this does not effect their validity.

5.3 Error 2 – The ABORT Property

As already mentioned, we attempted to verify this property three times, each time demonstrating an error in the model. We present the first attempt in this section.

Focusing on task 1, we stated that, if task 1 has locked and achieved property name 0 to denote property value 1 in the database (as in Fig. 18) and the environment breaks this property, then task 1 will be terminated. The two predicates in Fig. 28 formally define what it means for task 1's property to be broken and for task 1 to be terminated.

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

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

Figure 28. ABORT **predicates**

These definitions allow the ABORT property to be stated as the LTL formula in Fig. 29. The formula asserts that *"in all states, if* `task1_property_broken` *holds, then eventually* `task1_terminated` *will hold"*.

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

Figure 29. Formalization of ABORT **property**

This property is only interesting if task 1 may fail to terminate if it is not aborted. The closure in Fig. 15 is defined to provide this capability. It can arbitrarily choose the `true -> false` branch causing it to hang on the `false` expression. Of course, in the real LISP program a task will generally terminate. We are therefore really interested in the task being terminated within a certain time frame. However, since PROMELA cannot deal explicitly with time, we focus only on the distinction between termination and non-termination.

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, Fig. 22, starts and eventually calls `wait_for_events`.
2. A task, Fig. 18, starts, locks and achieves successfully. It then signals `LOCK_EVENT` from `lock_property` (Fig. 11) and begins executing its closure. This closure chooses the `true -> false` branch.
3. The daemon is awakened, but discovers no inconsistencies, so it decides to wait again. The decision to wait occurs at the last `else` branch in Fig. 22. However, the call to `wait_for_events` does not happen immediately.
4. The environment, Fig. 23, introduces an inconsistency, and signals the `MEMORY_EVENT`. However, this signal does not affect the daemon since it has *already* checked the counter and decided to call `wait_for_events`.
5. The daemon now calls `wait_for_events`. Hence, the task is not aborted, but continues its "big" computation.

The programmer's comment about this error was: *"This [bug] 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!"* In fact, this code pattern, which reoccurred in a different part of the code, **did** cause a deadlock in flight, as explained at the end of the paper.

A solution to the problem illustrated above is to embed the decision to wait and the actual waiting into a critical section that cannot be interrupted by other processes. Fig. 30 shows the `Daemon`, extended with such a critical section.

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

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

Figure 30. New `Daemon`

5.4 Error 3 – The ABORT Property

In the attempt to verify the ABORT property in the corrected model, SPIN yields an error trace demonstrating that the property is still not satisfied. The trace illustrates the following sequence of events:

1. The daemon, Fig. 30, starts and eventually calls `wait_for_events`.
2. A task, Fig. 18, starts, locks and achieves successfully. It then signals `LOCK_EVENT` from `lock_property` (Fig. 11) and starts executing its closure. This closure chooses the `true -> false` branch.

3. The daemon is awakened and calls `check_locks`, Fig. 21. The daemon executes the first loop of `check_locks`, finds no violation, and is *ready* to execute the second loop.

4. The environment, Fig. 23, introduces an inconsistency, and signals the `MEMORY_EVENT`. However, the daemon is already running, so the only effect is that the `MEMORY_EVENT` counter is incremented.

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.

6. Finally, because the counter for `MEMORY_EVENT` was incremented in Step 4, the daemon executes `check_locks` again, but does not find anything wrong. It therefore calls `wait_for_events` without aborting the task.

The programmer's comment about this error was as follows: *"Ah, good point! I had neglected to consider the case where a new assertion that violates a lock happens in the middle of check-locks. It's hard to get out of a single-threaded mind set! Thanks for pointing this out ... the intent was that tasks would be notified whenever a locked property was violated after initial achievement. In some cases this can be important ... even if the constraint is automatically restored"*.

At the time this error trace was discovered, we believed that it was an intended behavior due to an initial hesitation by the programmer to classify it as an error. Only later was it confirmed to be unexpected and undesired. Hence, we did not correct it.

5.5 Error 4 – The ABORT Property

Since we originally did not regard the above situation as an error, we continued the verification as if it was a correct behavior. We reformulated the ABORT property to assert that when a property is broken, either it is repaired, or the task is terminated. This correctness criteria allows properties to oscillate, thereby "repairing" themselves without the daemon discovering the temporary deviation. This behavior is possible in the real system and is regarded as acceptable.

Hence, we introduce the predicate in Fig. 31, which returns $true$ if the database and the lock table match with respect to property name 0.

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

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

Figure 31. ABORT **predicate**

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

Figure 32. Re-formalization of ABORT **property**

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, Fig. 18, starts, and eventually calls `achieve_lock_property`, Fig. 13. It executes the call to `achieve`, and is delayed prior to executing the assignment to the `achieved` field.

2. At this point, the `Environment`, Fig. 23, introduces an inconsistency in the database.

3. The daemon, Fig. 30, is awakened by the environment change, but does not discover the inconsistency since the task has not yet set the `achieve` field. Hence, the daemon goes back to sleep.

4. The task now assigns $true$ to the `achieved` field, and continues as if everything was consistent.

The programmer's comment about this error was as follows: *"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."*

A solution to the problem is to place 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, as shown in Fig. 33. The SPIN model checker certified that the ABORT property in Fig. 32 was satisfied in this new model.

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

Figure 33. New `achieve_lock_property`

5.6 Error 5 – An Efficiency Problem

A design flaw was discovered in the LISP code during inspection of the error traces generated during the verifications above. The procedure `check_locks` is called twice whenever the daemon returns from `wait_for_events`. That is, when an event is signaled by a call to `signal_event`, Fig. 8, the event counter for that event is incremented and the waiting tasks are restarted. Thus, after the first call to `check_locks` (and perhaps `do_automatic_recovery`), the test:

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

evaluates to $true$, and hence another iteration of the loop is begun, re-executing `check_locks`. The programmer's comment about this error was as follows: *"It's a bug, but since it's just an efficiency problem it's pretty low priority"*.

5.7 "Daemon–Ready" Flag Required

In an early model, the tasks were spawned without waiting for the daemon to initialize itself. That led to the discovery of an error by the model checker. A lock violation could occur before the daemon got to its initial waiting point, and hence the daemon would ignore the violation and call `wait_for_events`. This situation was not considered an error in the LISP program since the daemon should always start before everything else. However, the programmer emphasized that this highlighted the need for a mechanism to insure all daemons have reached "steady state" before proceeding.

5.8 Statistical Information

The model was verified exhaustively using SPIN's partial order reduction algorithm and state compression. Fig. 34 shows the data for the different verifications. For each of the first four errors, we indicate what kind of property was verified, assertion (A) or LTL formula; the number of states explored by SPIN; the memory consumption in Mb; and finally the time to locate the error.

The last two rows show data for the corrected models. After error 1 was corrected, the model was verified to be correct with respect to the assertions. After error 4 was corrected, the model was verified correct with respect to the LTL formulae. The LTL formulae were verified in a model where assertion error 1 had not been corrected.

Error	Kind	States	Memory (Mb)	Time (sec)
1	A	2963	2.582	0.3
2	LTL	49038	3.708	5.3
3	LTL	45705	3.606	4.9
4	LTL	48858	3.708	5.4
1√	A	222840	7.088	21.2
4√	LTL	107479	5.040	11.6

Figure 34. Verification data

6 Discussion

In general we regard the effort as being a very successful application of SPIN. Five errors were identified in the PROMELA model within a week of its creation; the programmer confirmed that each represented an error in the LISP code. In other words, the errors found were *real*, and not only errors in the model. The errors were all classical concurrency 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.

6.1 Analysis of the Effort

The modeling effort, obtaining a PROMELA program from the LISP program, took about 12 man weeks during 6 calendar weeks. The verification effort was small in contrast, taking about one week. Once the model was formulated the properties to be verified were easily formulated in terms of assertions or LTL formulae. The model checker found the 5 errors right away.

The modeling effort consisted of two sub-activities: *abstraction* and *translation*. While these two activities can be separated, in practice, we performed abstraction as we were translating the system. By *abstraction* we mean the activity of reducing the original LISP program to a finite state system, which is small enough for efficient verification, but accurate enough to contain potentially important errors. In this case, our focus was on the tasks, the daemon, and their operations on the lock table, and whether these operations were safe. Hence, though we did not know the properties to be verified, we had some gross guidelines of what to eliminate: everything not crucial for the operations on the lock table. We therefore performed a form of slicing with respect to the lock table data structure. As an example, in the model we have completely ignored the goal specific contents of the individual tasks, such as details of operating the camera. Also, we consider only two tasks and two properties, while in the real system, several tasks and properties are allowed.

An ideal abstraction is *sound* and *complete* with respect to the properties being verified. By a *sound abstraction* we mean that if an error is present in the abstract program it is also present in the concrete program. By a *complete abstraction* we mean the dual: if an error is present in the program then it is present in the abstraction. In our case, the abstraction was sound with respect to the two properties being verified since all 5 errors found in the abstraction were also present in the concrete program. However, the abstraction may not be sound with respect to properties that we have not analyzed. Furthermore, the abstraction is most likely not complete. There may still be errors in the program that we have not caught. No formal attempt was made to show soundness and completeness. Abstractions were based more on intuition than on formal reasoning. Interestingly enough, this informal *abstraction* activity was regarded as easy. Once a piece of code was understood, deciding what to keep and what to remove was often quite clear.

By *translation* we mean the activity of writing the actual PROMELA code, based on the now abstracted LISP program. For example an if-statement in LISP was mapped into and if-statement in PROMELA and LISP lists were mapped into arrays or channel buffers. Generally, for each LISP construct, we identified a set of PROMELA constructs which, when put together, would have an equivalent semantics and would keep the state space as small as possible.

The translation phase was non-trivial. It was the most time consuming activity due to the strength of the LISP language compared to the weaker PROMELA language. This work motivated the introduction of inline procedures and nested atomic statements in SPIN.

6.2 Some Citations from the Programmer

We asked the programmer three questions regarding the impact of our work on the development of the system. His answers are cited below. We found the responses interesting due to the difficulty that is often encountered when integrating formal methods into software practice. This information is primarily anecdotal, but it gives some idea of the programmer's perception of our work.

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

6.3 Follow Up Work

As described above, the modeling activity can be regarded as consisting of *abstraction* and *translation*. The translation phase could have been fully automated. Moreover, in our opinion, the abstraction phase could have been semi-automated to an extent where a programmer could have abstracted the program to a form suitable for model checking in less than one day.

These findings have led to two corresponding research activities in the Automated Software Engineering group at NASA Ames: *an abstraction workbench* and a *translation workbench*. The general focus for our work is to provide tools that can find complex interleaving errors in programs rather than prove them totally correct. Our focus is on programs written in mainstream programming languages, such as JAVA. We hope that this focus will provide a useful synergy between technology providers and technology users.

Some abstractions can be done fully automatically, such as program slicing. More sophisticated approaches to abstraction are based on theorem proving: a theorem prover is used to formulate abstractions and prove them correct (e.g., [7]). Some work tries to automate these more sophisticated abstractions [1, 2, 5]. Also interesting is the effort to integrate model checking into the PVS theorem prover [10]. In general we imagine an interactive abstraction environment which would allow unsound and incomplete abstractions, as well as sound and complete ones.

We have developed a translator from a subset of the JAVA programming language to PROMELA [6, 8]. Similar work, described in [4], also translates JAVA programs into PROMELA, but does not handle exceptions or polymorphism. Finally, [3] describes a translation of JAVA to a transition model, using static pointer analysis to aid *virtual coarsening* and reduce the size of the model.

6.4 Aftermath - The Remote Agent Anomaly

On May 18, 1999, The Remote Agent was running in space on board the DEEP-SPACE 1 space craft, when an anomaly occurred: thrusting did not turn off as requested. The Remote Agent experiment was immediately terminated from ground, and the space craft put in stand-by mode. The experiment was continued only after the error was understood. The anomaly was caused by a piece of code with the exact same pattern that caused error number 2 from section 5.3: a missing critical section around a conditional wait statement allowed an event to arrive between the evlauation of the condition and the wait. Thus, the thread executing the wait missed the event and the system was caught in a deadlock. The flawed code was part of a module not analyzed using SPIN.

Acknowledgments

We would like to thank Erann Gat, the lead developer of ESL, for his useful responses to our error reports, and for the comments above. We also thank Ron Keesing and Barney Pell for explaining parts of the Executive and suggesting properties to be verified. Finally, but certainly not least, we want to thank SPIN's designer, Gerard Holzmann, for his always reliable support during the work.

References

[1] S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *CAV'98: Computer-Aided Verification*, number 1427 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

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

[3] J. C. Corbett. Constructing Compact Models of Concurrent Java Programs. In *Proceedings of the ACM Sigsoft Symposium on Software Testing and Analysis*, March 1998. Clearwater Beach, Florida.

[4] C. Demartini, R. Iosif, and R. Sisto. Modeling and Validation of Java Multithreading Applications using SPIN. In *Proceedings of the 4th SPIN Workshop*, November 1998. Paris, France.

[5] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *CAV'97: Computer-Aided Verification*, number 1254 in Lecture Notes in Computer Science. Springer-Verlag, 1997.

[6] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.

[7] 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*, number 1051 in Lecture Notes in Computer Science. Springer-Verlag, 1996. Oxford, England.

[8] K. Havelund and J. Skakkebæk. Applying Model Checking in Java Verification. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking – 5th and 6th International SPIN Workshops*, number 1680 in Lecture Notes in Computer Science. Springer-Verlag, July and September 1999. Trento, Italy – Toulouse, France (presented at the 6th Workshop).

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

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

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

[12] G. L. Steele. *Common LISP – The Language*. Digital Press, 1990. Second edition. The `unwind-protect` construct is described on page 188.