1. The Problem

Test and verification are the most costly parts of flight software development. Automation, such as model checking, can significantly improve software test coverage over hand-coded tests; but it too can be expensive. To illustrate, time and cost constraints allowed only four of the 150 software modules on board the Curiosity rover to be checked by the SPIN model checker.\(^1\) The question is: how can we improve the reliability of flight software while reducing the cost?

We have concluded that a significant source of cost lies in the programming languages used to write flight software (primarily C and C++), and in particular two aspects of those languages:

1. Low-level, unsafe programming language constructs.
2. An inability to cleanly express key flight software concepts.

As to the first point, low-level constructs like pointers, semaphores, and callbacks are both powerful and necessary: they are the elementary building blocks from which any algorithm or data structure can be constructed. However, these constructs are also much too primitive for general use; instead they should be hidden wherever possible behind suitable abstractions. Otherwise the code is tedious to write, unsafe and error-prone, hard to reuse, and impossible to analyze. Unfortunately, C — which was originally designed for programming operating systems — often requires the use of primitive mechanisms like pointer manipulation, with no satisfactory alternative. C++ provides more capabilities for building abstractions, but it is very complex; and still it encourages low-level, hard-to-analyze code.

As to the second point, we claim that flight software can be improved by adding certain key concepts to the programming model. In our view the most important concept is an exact accounting of state. By state we mean a variable whose value persists across messages or function calls, such as the number of bytes in a telemetry buffer.\(^2\) By exact, we mean the compiler can determine the minimum number of bytes required to represent the state, as well as its location and actual size.

Knowing this information provides several benefits. For example, testing is largely an issue of state space size. That is, the number of combinations of state and input values is so large that there is not enough time to test them all. This was the principal motivation for SPIN: its job is to create and run tests, and do so in a way that efficiently traverses the state space by avoiding execution paths already traveled on some previous test.

---

\(^1\) Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. Model checking is one of a number of formal methods that can help ensure program correctness; it works by exploring all executions of a simplified version of a program, called a model. Other formal methods include abstract interpretation and theorem proving. SPIN is a widely-used model checker.

\(^2\) Other examples include a control mode or estimation state vector. We do not mean the state of the bits in a memory word (though that may well be state in some lower-level model of a hardware system).
Finally, we believe the two points are really two sides of a single design philosophy. While it is unrealistic to eliminate all use of low-level constructs in flight code, if programmers have good abstractions for the job at hand, then use of these lower-level techniques will be a last resort, rather than a first choice. The result should be an improvement in the three “R’s” of software quality: readability, reliability, and reusability.

2. Our Solution

We have defined a state-aware programming language called Spot, derived from C. On the one hand, Spot disallows unsafe uses of low-level constructs like pointers; on the other, it extends C to provide key abstractions for flight software, particularly in regard to state and real-time processing.

2.1. Spot Basics

Spot follows the actor model\(^3\) of concurrent computation: a Spot program consists of several modules that interact concurrently by sending each other messages. A Spot module is a unit of computation that corresponds to a module in traditional flight code; it is instantiated at runtime into one or more module instances that encapsulate some state and some related operations on that state.

```
module Counter {
    constructor create () {} 
    state int count = 0;
    priority P qsize 100;
    message void increment (int i) priority P {
        next count = count + i;
    }
    message int read () priority P { return count; }
}
```

Figure 1: Example Spot code

The code snippet shown in Figure 1 illustrates these concepts. Line 1 defines a module Counter, representing a counter variable together with increment and read operations on the variable. Line 2 defines a constructor create for creating instances of the module. Line 3 defines the variable count and specifies several facts about it: its type is int, its initial value is 0, and it is part of the state of the Counter module. In Spot, state consists of mutable variables defined inside module definitions, and no other variables. In line 5, for example, variable i is not state, because it is a local variable. Global mutable variables are not allowed in Spot.

The rest of the code defines the messages that increment and read the state of the counter. Message definition are given priorities that govern the order in which they are handled. Otherwise a message definition looks much like a C function definition. A message may be sent by invoking it on a target module inside a send statement, for example:

```
var int x;
var Counter c = Counter.create ();
send c.read () receive x;
send c.increment ();
```

In line 3, the return value of the read message is transmitted to the caller via an implicit return message and stored in the variable x. This operation causes the caller to block and wait for the return value; if more concurrency is needed, one may use a non-blocking receive or an explicit callback.

Line 6 of Figure 1 shows how state update occurs in Spot. The keyword next specifies that (1) state is being updated; and (2) the update is to the next state, that is, the state that will be seen the next time the message is invoked. During the current execution, the message body sees the old value of count; updates done via next are buffered and applied at the end of message execution.

A message with void return type may update remote state, so the Spot runtime does not actually send any such message until the end of the enclosing message. For example, in the code fragment above, if lines 3 and 4 were swapped, then the increment message would still be sent last. A message with non-void return type (e.g.,

read in Figure 1) may not update remote state. These rules ensure that no message updates any state (either locally or remotely) until it is finished executing.

Other features of Spot include the following:

- **Improved arrays**: In Spot, arrays store their dimensions, enabling bounds checking and making array-traversal code more compact and easier to understand. Spot also cleanly separates pointer and array types: for example, array indexing is allowed, but pointer indexing and arithmetic are not.

- **Value types**: Data structures may be created and initialized together, then treated as immutable values. In C, such structures require assignments and pointer manipulation.

- **Annotations for testing and model checking**: Spot lets the programmer write annotations specifying (1) how to generate tests for messages and functions and (2) how to check messages using the SPIN model checker.

### 2.2. Benefit of Spot

The Spot programming model provides several benefits over writing flight code in plain C. First, by providing higher-level abstractions such as modules, improved arrays, and value types, Spot increases productivity and code quality as explained in Section 1.

Second, Spot’s type system ensures that state is always passed by value, never by reference, between modules. This ensures strict partitioning of the memory representing the state of each module. By eliminating global variables and separating state from non-state memory, Spot also enables a simple form of memory management: any memory allocated within a message handler can be automatically deallocated after the handler finishes running.

Third, modules update their state **atomically**: no other module may see an inconsistent state (for example, halfway through an update). This fact enables Spot to generate all telemetry code for the spacecraft, and much of the ground code, automatically. It also simplifies (1) aborting and restarting messages in response to a fault; and (2) migrating messages from one core to another, for example due to a change in power allocation.

Fourth, Spot enables flight code developers who are not expert model checkers to use SPIN, as it solves two of the thorniest issues: what is the state, and what is the model to be checked? Identification of state works as explained above. The model is the program itself, and the Spot compiler produces all the code that SPIN needs to do its work.

Fifth, Spot naturally supports automatic parallelism on multicore architectures. If the programmer follows simple rules (such as using value types instead of pointers and mutable structures), then the compiler will produce code that runs a single module in parallel on multiple cores.

Finally, Spot compiles to C. Therefore the standard tool chains (compilers, analyzers) and libraries are compatible with Spot.

### 3. Implementation Status

We have written a specification for Spot consisting of a formal syntax and an informal semantic description. We are implementing a compiler and runtime based on the specification. Our current implementation contains a complete lexer and parser, a mostly-complete Spot-to-C code generator, and enough semantic analysis to support code generation. We plan to implement a full type checker. The current runtime runs on Unix-like systems and uses pthreads as the concurrency mechanism; porting to other systems such as VxWorks should not be difficult.

We have used the compiler and runtime to compile and run a number of small Spot modules. We have also prototyped the verification methodology (annotations plus SPIN code generation) for these modules. We have integrated the annotations into our Spot compiler and are now integrating the SPIN code generation.

Once we have finished the compiler and runtime, we plan to translate several modules from the Curiosity flight software into Spot. We will then evaluate the efficacy of the approach. In particular, we plan to investigate two questions:

1. What are the gains in productivity and safety versus plain C?
2. What is the performance cost?

Several flight projects (Mars Science Laboratory, Asteroid Retrieval, and Comet Rendezvous) have expressed interest in Spot, particularly for developing Guidance, Navigation, and Control (GNC) systems.
4. Conclusion

We have briefly described Spot, a new programming language based on C for programming flight software systems. Spot offers enhanced programmability over plain C, and it interoperates with legacy C code. By carefully managing program state, Spot also supports semi-automatic verification, automatic memory management, fault tolerance, and multicore parallelism. We believe that Spot is useful not just for flight systems, but for any system in which safety, fault tolerance, or security are essential.