Spot: A Programming Language for Verified Flight Software

Robert L. Bocchino Jr.
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
bocchino@jpl.nasa.gov

Edward Gamble
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
ed_gamble@me.com

Kim P. Gostelow
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
kimpgostelow@gmail.com

Raphael R. Some
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
Raphael.R.Some@jpl.nasa.gov

ABSTRACT

The C programming language is widely used for programming space flight software and other safety-critical real time systems. C, however, is far from ideal for this purpose: as is well known, it is both low-level and unsafe. This paper describes Spot, a language derived from C for programming space flight systems. Spot aims to maintain compatibility with existing C code while improving the language and supporting verification with the SPIN model checker. The major features of Spot include actor-based concurrency, distributed state with message passing and transactional updates, and annotations for testing and verification. Spot also supports domain-specific annotations for managing spacecraft state, e.g., communicating telemetry information to the ground. We describe the motivation and design rationale for Spot, give an overview of the design, provide examples of Spot's capabilities, and discuss the current status of the implementation.

Categories and Subject Descriptors

D SOFTWARE [**D.3 PROGRAMMING LANGUAGES**]: D.3.2 Language Classifications—*Concurrent, distributed, and parallel languages*; D SOFTWARE [**D.1 PROGRAMMING TECHNIQUES**]: D.1.3 Concurrent Programming; D SOFTWARE [**D.2 SOFTWARE ENGINEERING**]: D.2.4 Software/Program Verification—*Model checking*; D SOFTWARE [**D.2 SOFTWARE ENGINEERING**]: D.2.5 Testing and Debugging

General Terms

Verification; Validation; Safety; Reliability

Keywords

Flight Systems; Avionics; Actors; Message Passing; Domain-specific Annotations

HILT 2014, October 18–21, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2663171.2663185.

1. INTRODUCTION

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 [2].[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, though C++ is used as well), 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

[1]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.

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++ has improved capabilities for abstraction-building, but it is very complex; and still it encourages low-level, hard-to-analyze code. The Ada programming language solves many of the expressivity and safety problems of C and C++, but rewriting of C flight code in Ada is simply not practical. Further, Ada's concurrency model is not ideal for flight code.

As to the second point, we claim that flight software can be improved by adding a handful of 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. As a concrete example, the SPIN model checker works by systematically exploring different reachable states of a program. When it reaches a state it has seen before, it stops and backtracks to a different state; there is no use continuing, because any state reachable from that point has already been seen. A basic assumption of SPIN, therefore, is that *every single bit of global state in the program is known and accounted for*. If this assumption is violated, then SPIN cannot work properly; usually it will fail with strange errors or crashes. Moreover, accounting for state in this way is extremely difficult when the language is as unstructured as C. Additional benefits, such as generation of telemetry communication code, are discussed later in this paper.

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. SPOT

We are developing 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 Design Rationale

At the heart of Spot's design is the very practical consideration that, for better or for worse, we are stuck with the C programming language for flight code at JPL. Therefore, our main goal is to find a minimal set of extensions and restrictions to C that can provide the expressivity and safety guarantees we seek. In particular the language must do two things:

1. Describe the program state sufficiently that the location and size of all state variables can be communicated to SPIN.
2. Provide a built-in concurrency model, so that a SPIN model can be automatically constructed.

We discussed point 1 in the introduction. As to point 2, modeling a concurrent system is extremely difficult if, as is currently the case in JPL flight code, task creation and message passing are scattered throughout a C program in the form of unstructured library calls.[3] Model construction is much easier (indeed, we believe it can be done mostly automatically) if the language itself provides a structured concurrency model.

We do not aim to design and implement an entirely new language from scratch, for several practical reasons. First, designing a new language (especially a good one!) is extremely difficult; it is much easier to extend an existing language. Second, achieving adoption seems much more realistic for a variant of C than for an entirely new language. Finally, the design and implementation of an entirely new language is beyond our cost budget. We do believe, however, that if Spot is successful, its ideas could form the basis for a new language in the future.

We also believe that any viable solution must maintain linkage compatibility with existing C code, so that wholesale rewriting in Spot is not required from the start. The linkage compatibility must go in both directions, i.e., it must be easy to link both Spot files into C programs and C files into Spot programs. That way, Spot can gradually replace C — for example, on a component-by-component basis — in existing flight code. We believe that strict adherence to C syntax, especially in its more ungainly aspects, is not required, so long as this linkage compatibility exists.[4]

2.2 Features

We now briefly discuss the major features of Spot.

Actor concurrency. Spot follows the *actor model* [1]: 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.

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`

[2]Other examples include a control mode or position estimate. 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).

[3]Our experience has been that constructing a SPIN model for a single JPL flight module takes roughly three man-months, and that most of this time is spent identifying the concurrent tasks and their interactions.

[4]One major design mistake of C++, in our opinion, is that it is both too rigid in its adherence to C syntax and not flexible enough in its linkage idiom. While linking C files to C++ is straightforward, the opposite direction is not.

```
1  module Counter {
2    constructor create () {}
3    state int count = 0
4    priority P qsize 100
5    message void increment (val int i)
6      priority P
7    {
8      next count = count + i;
9    }
10   message int read ()
11     priority P
12   {
13     return count;
14   }
15 }
```

Figure 1: Example Spot code.

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 constant local variable. As in Scala, the keyword `val` denotes a variable that is immutable after initialization (similar to `const` in C), while the keyword `var` denotes a mutable 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 definitions 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, as shown in Figure 2. 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.

```
1 var int x;
2 val Counter c = Counter.create ();
3 send c.read () receive x;
4 send c.increment ();
```

Figure 2: Blocking receive.

Line 8 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 of the module that will be seen the next time one of its messages 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 Figure 2, 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. They also extend the `next` semantics to remote memory access: effectively, every remote update acts upon the next state of the updated object, while every remote read sees the current state.

Annotations for correctness and testing. Spot provides a flexible way to write various kinds of program annotations, including the following:

- Standard C-style assertions.
- Design-by-contract-style preconditions and postconditions, i.e., `assumes` clauses stating what is assumed to be true at the start of a function or message and `guarantees` clauses stating what is guaranteed to be true on exit from a function or message, assuming that the `assumes` clauses are satisfied.
- Temporal logic specifications — for example, assertions that a certain value of a state variable must eventually be achieved — for guiding the SPIN model checking.
- Annotations that specify tests: for example, test inputs and expected results for checking the behavior of functions, messages, and modules.

The test input sets can be specified either directly, e.g., as a range of values, or indirectly as one or more conditions. As an example of a conditional input specification, an annotation might say "check all inputs such that the condition b holds," where b is a Boolean function of the inputs.

The syntactic form of a Spot annotation is very simple: it is just the symbol `@` followed by an identifier and an optional expression which, if it appears, must be enclosed in parentheses. The expression may be arbitrarily complex and may be (or include) a function call. Very general conditions, assertions, input ranges, and expected test outputs may be expressed with this simple syntax.

As in languages such as Java and Scala, the Spot compiler just provides the syntax of annotations; their meaning (generating executable assertions, for example, or generating tests) is provided by plugin compiler passes or separate analyzers. This approach keeps the annotation language very flexible and lets it be adapted to new purposes.

```
1 message void increment(val int i)
2   priority P
3   @assumes (i > 0)
4   private @guarantees (
5       next count == count + i
6   )
7 {
8   next count = count + i;
9 }
```

Figure 3: Example Spot assertions.

Figure 3 shows a simple of example of contract-style assertions in Spot. Here we have annotated the `increment` message from Figure 1 with assertions stating that (1) the value bound to the message parameter `i` must be greater than zero and (2) the result of the message is to increase the value of the state variable `count` by `i`. The second annotation is marked `private` (meaning it is visible only in this

translation unit) because it refers to the private state variable `count` of module `Counter`. (All state variables in Spot are implicitly private.) These annotations could generate executable assertions and/or be checked by SPIN.

Other features. Other features of Spot include the following:

- *Improved arrays.* In Spot, arrays store their dimensions, enabling arrays to be safely passed as message arguments. 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, structures require assignments and pointer manipulation.
- *Domain-specific annotations.* Spot supports flight-specific annotations, for example to guide the generation of telemetry information. These are discussed further in Section 3.

Of course genuine arrays and safe references are standard features of most modern languages; but they are also critical to making C suitable as a basis for Spot.

2.3 Benefits

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. The partitioning enables easy migration of modules from one core to another — for example due to a change in power allocation — even when the cores do not share the same physical memory. 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). Thus Spot messages behave like *transactions* in a database or other transaction processing system. As discussed in the next section, this fact enables Spot to generate all telemetry code for the spacecraft, and much of the ground code, automatically. It also simplifies aborting and restarting messages in response to a fault.

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 and is fully linkage-compatible with C, as discussed above.

3. DOMAIN-SPECIFIC ANNOTATIONS

As a concrete example of the benefits of Spot, we discuss domain-specific annotations for handling two kinds of spacecraft state: telemetry and parameters. Telemetry is data sent to the ground in real time, so that ground operations personnel can monitor the state of the system and, in the case of a mishap, determine what has gone wrong. Parameters are values that govern spacecraft computations, for example, an alignment correction for a gyro device. Parameters are essentially constants, but sometimes they must be updated from the ground. For example, one might discover after launch that a gyro has moved and requires an alignment adjustment.

```
1  module GnC {
2    @periodic (q, planet)
3    @onchange (planet)
4    state GncVector x
5    ...
6    @param
7    state GncParms z
8    ...
9  }
10
11 type GnCVector = struct {
12   var double[4] q
13   var Planet planet
14   var GncMode mode
15   var int a
16 }
```

Figure 4: Parameter and telemetry annotations.

Figure 4 illustrates how Spot expresses these concepts. Lines 1–9 define part of a module called `GnC`. (`GnC` stands for "guidance, navigation, and control.") In the part of the state that is shown, there are two variables: a variable `x` of type `GncVector` and a variable `z` of type `GncParms`. `GncVector` and `GncParms` are both structure types; the definition of the `GncVector` struct is shown in part in lines 11–16. The annotations appearing in lines 2–6 use the general annotation syntax described in Section 2.2. Here, though, the annotations describe telemetry communications and parameters, instead of providing test inputs or correctness conditions.

The annotations say the following:

- The contents of the structure members `q` and `planet` of `x` should be sent periodically in telemetry communications to the ground.
- The contents of the structure member `planet` of `x` should additionally be sent as a telemetry communication to the ground whenever it changes.
- `z` is a parameter variable.

Notice that the annotations are on the variables, and the variables carry their types. Therefore the compiler has all the information it needs to generate code for transmitting code to and from the spacecraft either in response to ground commands (in the case of parameters) or at specified times or under specified conditions (in the case of telemetry).

In contrast to these simple annotations, the code for telemetry and parameter communication in state-of-the-art

flight systems is a mess: it is complex, ad-hoc, and a burden to read, write, and maintain. Typically it consists of handwritten XML specifications. Not only is this needlessly painful, but it forces programmers to maintain essentially the same information (i.e., the types and sizes of the relevant state variables) in two different places. Primarily this is because the state information, while implicit in the C code, is not readily extracted, and so must be specified by hand all over again. By contrast, once the state is known and accounted for, the parameter and telemetry code can be readily generated with a few simple annotations.

We cannot overstate the potential savings from automatic parameter and telemetry code not only in programming time, but also in testing time. Furthermore, ground tools can utilize the information to generate tables needed to display the data on the ground control displays.

4. 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.

5. 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 potentially useful not just for flight systems, but for any system in which safety, fault tolerance, or security are essential.

6. ACKNOWLEDGEMENTS

We performed this research at the Jet Propulsion Laboratory (JPL), California Institute of Technology. Our funding came from the Game Changing Development (GCD) program at the Space Technology Mission Directorate, National Aeronautics and Space Administration; and from the Research and Technology Development (R&TD) program at JPL.

7. REFERENCES

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Publishing Company, 2003.