Spot: A Programming Language for Verified Flight Software

Rob Bocchino
Ed Gamble
Kim Gostelow
Rafi Som

Jet Propulsion Laboratory
California Institute of Technology

High-Integrity Language Technology (HILT)
October 21, 2014

© 2014 California Institute of Technology
Government sponsorship acknowledged
Motivation

• Most flight software (FSW) today is written in C

• Pros
  ✓ Familiar
  ✓ Simple
  ✓ Low overhead
  ✓ Easy to reason about resource use (speed, memory, power)

• Cons
  X Lacks important abstractions for FSW
  X **Requires** unsafe, low-level code
  X Verification and validation (V&V) is very expensive
Example

Mars Science Laboratory (MSL) FSW coverage using the Spin model checker
Experience

• Spin is under-utilized for FSW
  – Extracting a Spin model is hard work
  – Three man-months per module

• Reason: C is very unstructured

<table>
<thead>
<tr>
<th>Program Property</th>
<th>Expressed in C as</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spacecraft state</td>
<td><code>malloc</code>, <code>pointers</code></td>
</tr>
<tr>
<td>Concurrency</td>
<td><code>C library calls</code></td>
</tr>
<tr>
<td>FSW abstractions</td>
<td><code>C library calls</code></td>
</tr>
</tbody>
</table>
Our Solution: Spot

• A new domain-specific language (DSL) or FSW

• Based on C
  − Retains the benefits of C or FSW programming
  − Linkage compatible with C in both directions
  − Supports incremental adoption

• Key features
  − FSW abstractions: modules and messages
  − Improved memory management and precise accounting of state
  − Annotations for automatic testing and verification
  − Improved arrays, no pointer arithmetic
  − Value type system supporting safe parallelization
Outline

• The Spot language
• Benefit
• Implementation status
• Future plans
Module Code

```java
module Counter {
    priority P qsize 100
    constructor create () {}
    state int count = 0
    message void increment () priority P {
        next count = count + 1;
    }
    message int read() priority P {
        return count;
    }
}
```

Client Code

```java
val Counter c = Counter.create ();
var int count;
send c.increment ();
send c.read () receive count;
printf ("count is %d\n", count);
```
Modules and Messages

Module Code

module Counter {
    priority P qsize 100
    constructor create () {} 
    state int count = 0 
    message void increment () priority P {
        next count = count + 1;
    }
    message int read() priority P {
        return count;
    }
}

Messages

Modules have state

Counter

Client Code

val Counter c = Counter.create ();
var int count;
send c.increment ();
send c.read () receive count;
printf ("count is %d\n", count);
Module Code

module Counter {
    priority qsize 100
    constructor create () {}
    state int count = 0
    message void increment () priority P {
        next count = count + 1;
    }
    message int read() priority P {
        return count;
    }
}

Messages

Modules have state

Messages operate on state

Client Code

val Counter c = Counter.create();
var int count;
send c.increment();
send c.read() receive count;
printf ("count is %d\n", count);
Modules and Messages

Module Code

```java
module Counter {
    priority P qsize 100
    constructor create () {}
    state int count = 0
    message void increment () priority P {
        next count = count + 1;
    }
    message int read() priority P {
        return count;
    }
}
```

Messages

- Counter

Client Code

```java
val Counter c = Counter.create ();
var int count;
send c.increment ();
send c.read ()
receive count;
printf ("count is %d\n", count);
```
Modules and Messages

Module Code

```java
module Counter {
    priority P qsize 100
    constructor create () {}  
    state int count = 0  
    message void increment () priority P {
        next count = count + 1;
    }
    message int read() priority P {
        return count;
    }
}
```

Messages

- **Counter**
  - `increment`
  - `read`

Client Code

```java
val Counter c = Counter.create ();
var int count;
send c.increment ();
send c.read () receive count;
printf ("count is %d\n", count);
```
Memory Management

1. Stack variables: As in C

2. Message-local heap variables
   - Are created during a message invocation
   - Do not persist across messages
   - Are automatically reclaimed at the end of a message

3. State variables
   - Must be declared
     • With state keyword
     • Inside a module definition
   - Are associated with a module instance \( m \)
   - Persist across all messages received by \( m \)

There are no global variables in Spot
Typing Guarantees

1. No two module instances share memory
   - Modules communicate by passing values
   - Easy to move modules between cores

2. State memory stores no pointers
   - State memory never points to non-state memory
   - Deallocate of message-local memory is safe
Updating State

• State update
  - Is called out with the `next` keyword
  - Occurs all at once at the end of message processing

• Purpose
  - Buffer current state for possible undo
  - Separate current state from next state in assertions

```plaintext
module Counter {
    state int count = 0
    ...
    message int read_and_increment () priority P {
        next count = count + 1;
        return count;
    }
}
```
Updating State

• State update
  - Is called out with the **next** keyword
  - Occurs all at once at the end of message processing

• Purpose
  - Buffer current state for possible undo
  - Separate current state from next state in assertions

```plaintext
module Counter {
  state int count = 0
  ...
  message int read_and_increment () priority P {
    next count = count + 1;
    return count;
  }
}
```
Updating State

- State update
  - Is called out with the `next` keyword
  - Occurs all at once at the end of message processing

- Purpose
  - Buffer current state for possible undo
  - Separate current state from next state in assertions

```haskell
module Counter {
    state int count = 0

    count is n
    message int read_and_increment () priority P {
        next count = count + 1;
        return count;
    } Set count to n + 1 and return n
}
```
Annotation Language

- Spot has a simple but powerful annotation language built in
- Syntax: `@ identifie ( expression )`
- Semantics: define by pluggable checker
  - Spin code generation
  - Design-by-contract-style runtime checks

```plaintext
module Counter {
    state int count = 0
    ...
    message void increment () priority P
    private @assumes (count >= 0)
    private @guarantees (next count == count + 1)
    {
        next count = count + 1;
    }
}
```
Other Features of Spot

• Improved arrays
  – Arrays store their length and are bounds-checkable
  – Fortran-style loops and array slices
  – Multidimensional arrays with variable dimension sizes
  – No pointer indexing! (Arrays ≠ pointers in Spot)

• Value types
  – You can atomically create and initialize immutable struct values
  – Essential for safe parallelization
  – In C
    • You can define a struct with `const` members
    • But atomic object creation is limited, even with C99 extensions
    • As a result, mutable structures are effectively required
Outline

• The Spot language
• Benefit
• Implementation status
• Future plans
Benefits of Spo

- Improved programmability vs. C
  - Module and message abstractions
  - Memory management and state partitioning
  - Improved arrays and value types
- Atomic update of state
- Auto-generation of
  - Verification
  - Telemetry
- Multicore support
- C compatibility
Atomic Update

- Message handlers function as atomic transactions
  - Modules $M_1, \ldots, M_n$ run concurrently
  - Within module $M_i$, handlers run sequentially
- Message $m$ can be safely aborted and restarted
  - If $m$ sends no message that updates remote state
    - All state accessed by $m$ is known and buffered
    - Just throw away next state and start over
  - If $m$ sends a message $m'$ that updates remote state
    - $m'$ must have return type `void`
    - Defer sending of $m'$ until $m$’s computation is done
- Big step towards controlled software reset
  - Avoid sledgehammer of system reboot
Verificatio

- Easy to translate annotations into runtime checks
  - Test cases
    - Input ranges
    - All cases satisfying condition $B$
  - `@assumes`, `@guarantees`, `@assert`

- Spin code generation is also straightforward
  - Concurrency is explicit
  - Typing guarantees reduce the state space

- Should vastly reduce the cost of V&V for FSW
Telemetry

- Telemetry causes lots of code generation
  - A pain to manage using current techniques
  - Duplicates information already in FSW code
- Spot can do much of this with simple annotations

```plaintext
module GnC {
  @periodic (q, planet)
  @onchange (planet)
  state GncVector x
  ...
  @param
  state GncParms z
  ...
}

type GncVector = struct {
  var double[4] q
  var Planet planet
  var GncMode mode
  var int a
}
```
Telemetry

- Telemetry causes lots of code generation
  - A pain to manage using current techniques
  - Duplicates information already in FSW code

- Spot can do much of this with simple annotations

```plaintext
module GnC {
  @periodic (q, planet)
  @onchange (planet)
  state GncVector x ...
  @param
  state GncParms z ...
}
```

Send q, planet periodically to the ground

```plaintext
type GncVector = struct {
  var double[4] q
  var Planet planet
  var GncMode mode
  var int a
}
```
Telemetry

• Telemetry causes lots of code generation
  - A pain to manage using current techniques
  - Duplicates information already in FSW code

• Spot can do much of this with simple annotations

```go
module Gnc {
  @periodic (q, planet)
  @onchange (planet)
  state GncVector x ...
  @param
  state GncParms z ...
}

Send q, planet periodically to the ground
Send planet to the ground when it changes

type GncVector = struct {
  var double[4] q
  var Planet planet
  var GncMode mode
  var int a
}
Telemetry

- Telemetry causes lots of code generation
  - A pain to manage using current techniques
  - Duplicates information already in FSW code
- Spot can do much of this with simple annotations

```
module GnC {
  @periodic (q, planet)
  @onchange (planet)
  state GncVector x
  ...
  @param
  state GncParms z
  ...
}
```

Send q, planet periodically to the ground
Send planet to the ground when it changes

```
type GncVector = struct {
  var double[4] q
  var Planet planet
  var GncMode mode
  var int a
}
```

z is a parameter variable
Multicore Support

• Each module
  – Is logically a thread
  – Can go on its own core

• Message bodies can be parallelized
  – Value types minimize access to shared mutable data
    • Write helpers as pure functions
    • Enables auto-parallelization
  – Where mutable data is required (e.g., arrays)
    • Encapsulate parallel data structures behind library APIs
    • Update state at top-level only

• Concurrent message handling is future work
Outline

• The Spot language
• Benefit
• Implementation status
• Future plans
Implementaton Status

• Draft language specification is done
  – Formal syntax
  – Informal semantics

• Compiler implementation is in process
  – Complete parser
  – Mostly-complete C code generator
  – Prototype Spin code generator

• Case studies
  – We have compiled, run, and verified several simple examples
  – Working on more extensive examples drawn from MSL code

© 2014 California Institute of Technology
Government sponsorship acknowledged
• The Spot language
• Benefit
• Implementation status
• Future plans
Future Plans

• Further evaluation to answer research questions
  − What are the gains vs. plain C
    • In safety and verification
    • In productivity?
  − What is the performance cost?

• Evaluate for deployment

• Several flight projects have expressed interest