Spot: A Programming Language for Verified Flight Software

Rob Bocchino (bocchino@jpl.nasa.gov)
Ed Gamble (ed.gamble@jpl.nasa.gov)
Kim Gostelow

Jet Propulsion Laboratory
California Institute of Technology

The Workshops on Spacecraft Flight Software (FSW 2013)
December 12, 2013
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*
1. Use C++
   - ✓ Classes and templates support abstractions
   - ✓ Performs well
   - ✗ Complex, opaque
   - ✗ Still low-level, hard to verify
   - ✗ Requires wholesale rewriting of existing FSW

2. Use a modern high-level language (Scala, Python, ...)
   - ✓ Abstraction support
   - ✓ Extensible syntax
   - ✗ May not be feasible for embedded, low power
   - ✗ Still hard to verify, requires rewriting
Our Solution: Spot

- New domain-specific language (DSL) for FSW
- Based on C
  - Retains the benefits of C for FSW programming
  - Interoperates with existing C code
- 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 auto-parallelization
• The Spot language
• Expected benefits of Spot
• Implementation status
• Future plans
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;
  }
}

Client Code

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

Client Code

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

© 2013 California Institute of Technology
Government sponsorship acknowledged
Module Code

```plaintext
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

Client Code

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

```cpp
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

```cpp
var Counter c = Counter.create();
var int count;
send c.increment();
send c.read() receive count;
printf("count is \%d\n", count);
```
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
Ty ping Guarantees

- Spot enforces two memory partitioning guarantees:
  1. No two module instances share any memory
  2. No state points to non-state, and vice versa

- Enforcement uses a combination of
  - Type consistency checks
  - Implicit deep copies of objects

```plaintext
module M {
  state T x;
  ...
  function void f(T *state y) { ... }
  message void m(T* y) priority P {
    f(&x);
    f(y);
  }
}
```
Typing Guarantees

• Spot enforces two memory partitioning guarantees:
  1. No two module instances share any memory
  2. No state points to non-state, and vice versa

• Enforcement uses a combination of
  – Type consistency checks
  – Implicit deep copies of objects

```java
module M {
    state T x;
    ...
    function void f(T *state y) { ... }
    message void m(T* y) priority P {
        f(&x); // state->state: normal pass-by-pointer
        f(y);
    }
}
```
Typing Guarantees

• Spot enforces two memory partitioning guarantees:
  1. No two module instances share any memory
  2. No state points to non-state, and vice versa

• Enforcement uses a combination of
  – Type consistency checks
  – Implicit deep copies of objects

```c
module M {
  state T x;
  ...
  function void f(T *state y) { ... }
  message void m(T* y) priority P {
    f(&x); // state->state: normal pass-by-pointer
    f(y); // non-state->state: implicit deep copy
  }
}
```
Updating State

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

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

```plaintext
module Counter {
    state 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

• Motivation
  – Buffer current state for possible undo
  – Separate current state from next state in assertions

```plaintext
module Counter {
    state count = 0;
    ...
    count is n
    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

• Motivation
  – Buffer current state for possible undo
  – Separate current state from next state in assertions

```plaintext
module Counter {
  state count = 0;
  ...
  message int read_and_increment() priority P {
    next count = count + 1;
    return count;
  }
  set count to n + 1 and return n
}
```
• Spot has a simple but powerful annotation language built in
• Syntax: `@ identifier ( expression )`
• Semantics: defined by pluggable checker
  – Spin code generation
  – Design-by-contract-style runtime checks

```java
module Counter {
    state 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 define and create immutable struct values
    
    ```
    declare value type tree_t;
    type tree_t = value struct { tree_t* left; tree_t* right; }
    ```
  
  - Essential for auto-parallelization
  - C **requires** pointers to mutable structs
Outline

• The Spot language
• Expected benefits of Spot
• Implementation status
• Future plans
Expected Benefits of Spot

- 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 code (Spin, runtime checks)
  - Telemetry code
- Multicore support
- C compatibility
Atomic Update

• Message handlers update state atomically
  – Modules \( M_1, \ldots, M_n \) run concurrently
  – Within a module \( M_i \), handlers run sequentially

• Message \( m \) can be safely aborted, restarted in many cases:
  – If it sends no message \( m' \)
    • All state accessed by \( m \) is known and buffered
    • Just throw away next state and start over
  – If it sends a message \( m' \) with return type \texttt{void}
    • Defer sending of \( m' \) until \( m \)'s computation is done

• Big step towards controlled software reset
  – Avoid sledgehammer of system reboot
• Easy to translate annotations into runtime checks
  – Test cases, e.g., input ranges
  – @assumes, @guarantees, @assert

• Spin code generation is also straightforward
  – C program is the Promela model
  – Spot semantics reduces the state space

• Should vastly reduce the cost of V&V for FSW
Telemetry Code

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

- Spot can do much of this automatically
  - Engineering, Housekeeping, and Accountability
    - Is state
    - Spot knows about it
  - Event Reporting (EVRs)
    - Are events associated with state change
    - Spot can check for specified state change and downlink as EVR
Multicore Support

• Each module is logically a thread, can go on its own core

• Message bodies can be parallelized
  – Use value types to 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
• Expected benefits of Spot
• Implementation status
• Future plans
Implementaton Status

• Language specification
  – Formal syntax
  – Informal semantics

• Compiler implementation
  – Alpha version (write in mix of Spot, C, and Scheme): done
    • Generates C and Promela (Spin) code
    • Links against runtime
  – Beta version: in progress
    • Parser is done
    • Spot-to-C and Spot-to-Promela code generators in progress

• Case studies
  – Compiled and ran simple examples
  – Working on more extensive examples drawn from MSL code
Outline

• The Spot language
• Expected benefits of Spot
• 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
  – Expect beta version of compiler by end of FY14
  – Several flight projects have expressed interest
Acknowledgements

• Rafi Some
  – Instigated the work
  – Suggested connection between state and reboot
• Jane Oh contributed to early prototype
• JPL flight software community
  – Miguel San Martin
  – Rob Manning
  – Bill Whitaker
• Programming languages research community
  – Object-oriented and actor-based programming
  – State encapsulation
  – Functional programming