Virtual Research Presentation Conference

A Modeling Language for Next Generation Flight Software

Principal Investigator: Klaus Havelund (348)
Co-Is: Robert Bocchino (348)
Program: Topic

Assigned Presentation RPC-128

Jet Propulsion Laboratory
California Institute of Technology

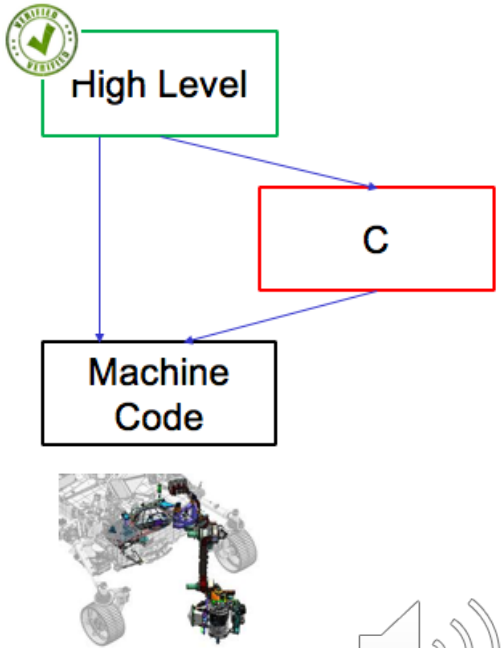
Introduction

JPL flight software (FSW) is written in C, sometimes in C++

- C is low-level and primitive
- C++ is complex and awkward
- Both are unsafe, lack modern language features, and hinder productivity

In this project we investigated

- What modern programming languages we can use instead of C and C++ for FSW
  - We studied Rust and Scala
  - We developed extensions to Scala
- How to add frameworks and domain-specific languages (DSLs)
- How to test and verify FSW written in the language



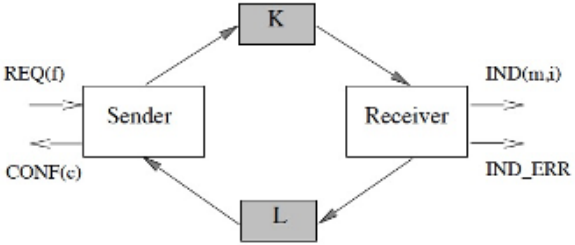
Relevance to NASA and JPL

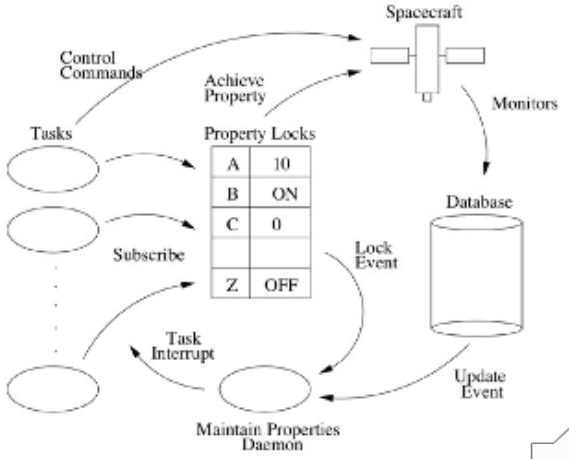
- Hardware power and complexity are increasing
- Software complexity is increasing
  - More mission activities
  - More advanced activities, e.g., on-board planning
  - Multicore programming
- Programming in C will become more and more challenging
  - At some moment a breaking point will be reached
  - This project is an attempt to be at the front of this path
- The project also addresses modeling of software
  - Modeling is essential for programming complex applications
  - Modeling must be integrated with the implementation language

Image taken from:
https://www.jpl.nasa.gov/spaceimages/details.php?id=PIA11431

Programming Exercise

- **Bounded Retransmission Protocol (BRP)**
  - File transfer with unreliable channels
  - Sender sends file records over channel K
  - Receiver sends acknowledgements over channel L
  - Channels can drop messages
  - Lost file records are retransmitted
- **Remote Agent (RA)**
  - AI planning and scheduling
  - *Planner* generates plans
  - *Plan execution engine* executes plans
  - *Monitor* monitors spacecraft state
  - We focused on plan execution engine (shown in figure)





Programming Languages

- Rust
  - First released by Mozilla in 2010
  - Provides high-level abstractions with high efficiency
  - Allows low-level programming
- Scala
  - Designed by the academic institution EPFL, Switzerland; first released in 2004
  - Powerful combination of object-oriented and functional programming
  - Runs on the Java Virtual Machine (JVM)

Results

Language	Safety	Efficiency	Ease of Use
Rust	High	High	Low
Scala	High	Medium	High

FSW Scala

- Scala presents significant challenges as a FSW implementation language
  - Runs on the Java Virtual Machine, not natively
  - Uses garbage collection
  - Provides insufficient control over memory layout
- We have begun to develop a language called *FSW Scala*
  - Full Scala language plus new features that support FSW programming
    - Native code generation
    - Allocation of class instances in static storage, on the stack, and on the heap
    - FSW compilation mode in which garbage collection is disabled
    - Mutable and immutable references to class objects
  - We have described a formal language called System E (for "embedded")
    - We have proved soundness results for this formal language.

Syntax of System E

System

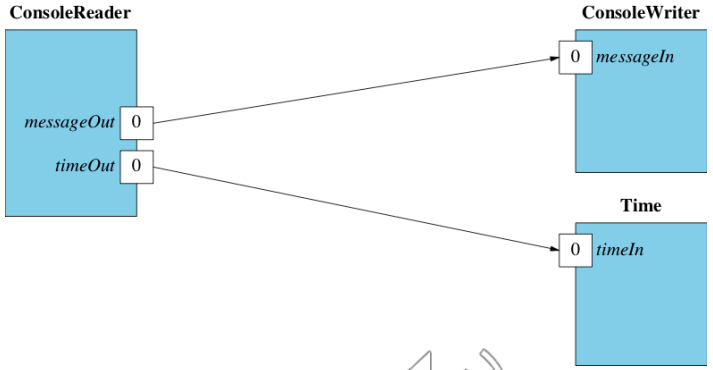
E

Natural Numbers	n	$0 \mid 1 \mid 2 \mid \ldots$
Size Parameters	s	
Binary Operators	B	$+ \mid *$
Size Expressions	S	$n \mid s \mid (S\ B\ S)$
Variable Names	x	
Lifetime Qualifiers	L	$\epsilon \mid \textbf{local}$
Mutability Qualifiers	M	$\textbf{val} \mid \textbf{var}$
Expressions	e	$S \mid \textbf{let}\ L\ M\ x = e\ \textbf{in}\ e \mid L\ \lambda(L\ M\ x : T)e \mid \pi s.e \mid$ $x \mid (e\ e) \mid \&M\ e \mid *e \mid (e = e) \mid (e\ B\ e) \mid \textbf{obj}(S)$
Permanent Types	P	$\textbf{nat} \mid T \to T \mid \Pi s.T \mid \textbf{ref}\ M\ T \mid \textbf{obj}(S)$
Types	T	$P \mid \textbf{local}\ T \to T \mid \textbf{ref local}\ M\ T$

Frameworks

F Prime

- Free, open-source FSW framework developed at JPL
- Based on components that communicate over ports

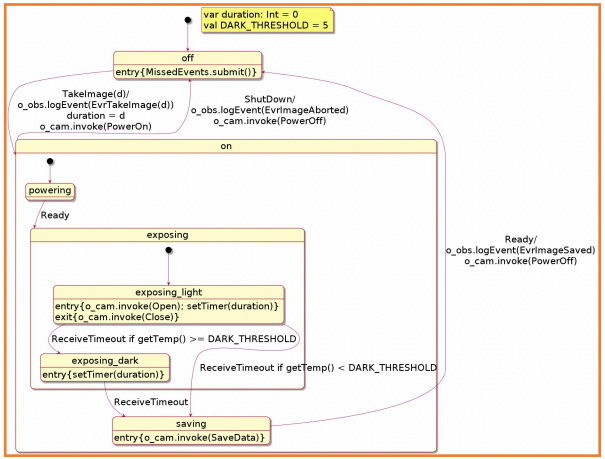


For each of Rust and Scala we did the following:

- Ported F Prime to the language
- Wrote several components
- Connected the components into an application

Domain-Specific Languages (DSLs)

- DSLs can be *internal* or *external*
  - Internal means the DSL is expressed directly in a host language
  - External means the DSL has its own parser and internal representation
- We wrote four internal DSLs in Scala
  - F Prime
  - HSM for programming with *Hierarchical State Machines*
  - Daut for monitoring with *Data automata*
  - Rules for rule-based testing
- In Scala, we implemented an external DSL for hierarchical state machines
  - Parsing, type checking
  - Visualization

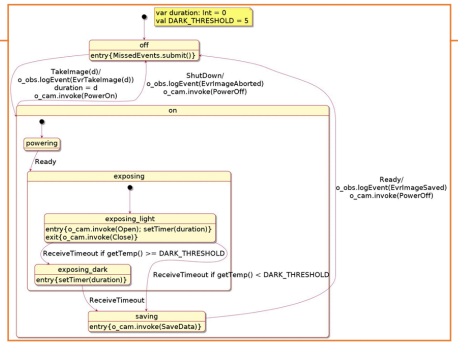


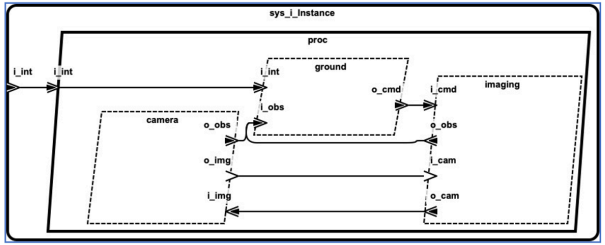
Scala makes it easy to construct DSLs

Four Internal DSLs

HSM

```
object on extends state() {
  when {
    case ShutDown => off exec {
      o_obs.logEvent(EvrImageAborted)
      o_cam.invoke(PowerOff)
    }
  }
}

object powering extends state(on, true) {
  when {
    case Ready => exposing
  }
}

object exposing extends state(on)

object exposing_light extends state(exposing, true) {
  entry {
    o_cam.invoke(Open)
    setTimer(duration)
  }
  exit {
    o_cam.invoke(Close)
  }
  when {
    case ReceiveTimeout => {
      if (getTemp() >= DARK_THRESHOLD) exposing_dark
      else saving
    }
  }
}
```





F'

```
val imaging = new Imaging
val camera = new Camera
val ground = new Ground

imaging.o_cam.connect(camera.i_img)
imaging.o_obs.connect(ground.i_obs)
camera.o_img.connect(imaging.i_cam)
camera.o_obs.connect(ground.i_obs)
ground.o_cmd.connect(imaging.i_cmd)
```

Daut

```
object SaveOrAbort extends Monitor[Observation] {
  always {
    case EvrTakeImage(_) => hot {
      case EvrImageSaved | EvrImageAborted =>
        ok
      case EvrTakeImage(_) =>
        error("Image was not saved or aborted")
    }
  }
}
```

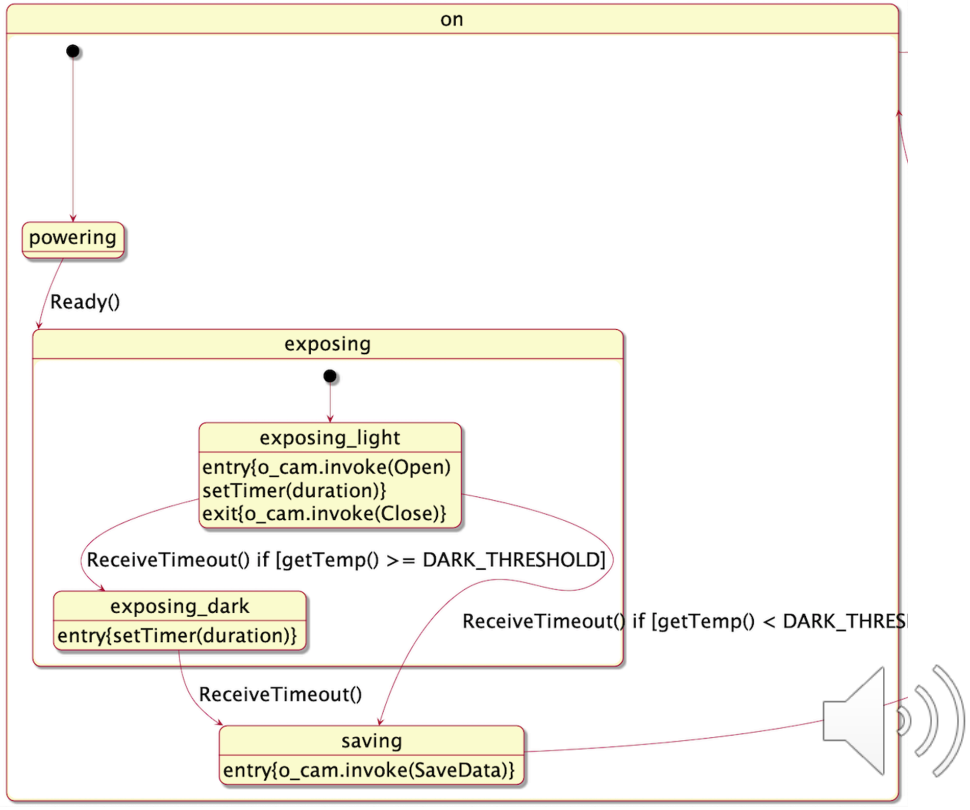
Rules

```
object TestRules extends Rules {
  val MAX_IMAGES: Int = 1000
  val MAX_SHUTDOWNS: Int = 1000
  val MAX_READY: Int = 1000

  var imageCount: Int = 0
  var shutdownCount: Int = 0
  var readyCount: Int = 0

  rule("TakeImage") (imageCount < MAX_IMAGES) -> {
    o_cmd.invoke((TakeImage(imageCount)))
    imageCount += 1
  }

  rule("ShutDown") (shutdownCount < MAX_SHUTDOWNS) -> {
    o_cmd.invoke(ShutDown)
    shutdownCount += 1
  }

  rule("Ready") (readyCount < MAX_READY) -> {
    o_cam.invoke(Ready)
    readyCount += 1
  }

  strategy(Pick())
}
```

```
state on {
  when {
    ShutDown =r> off {
      o_obs.logEvent(EvrImageAborted)
      o_cam.invoke(PowerOff)
    }
  }

  initial state powering {
    when {
      Ready =d> exposing
    }
  }

  state exposing {
    initial_r state exposing_light {
      entry {
        o_cam.invoke(Open)
        setTimer(duration)
      }
      exit {
        o_cam.invoke(Close)
      }
      when {
        ReceiveTimeout if [getTemp() >= DARK_THRESHOLD] =u> exposing_dark
        ReceiveTimeout if [getTemp() < DARK_THRESHOLD] =l> saving
      }
    }

    state exposing_dark {
      entry {
        setTimer(duration)
      }
      when {
        ReceiveTimeout =r> saving
      }
    }
```

Automatically visualized

External HSM DSL



Test and Verification

- We studied several tools for theorem proving
  - Stainless for Scala
  - Logika for an embedded programming version of Scala
  - Viper for Rust
- Theorem proving is powerful but difficult to use
- A lighter-weight approach is automated testing
  - We studied the P programming language (Microsoft)
  - Supports automated testing of state machines
  - This kind of technology seems more suited for practical use
- We investigated runtime monitoring for use in flight
  - A key challenge is to make it memory efficient
  - We investigated the use of memory pools

Results

- FSW Scala
  - High potential
  - More work required
  - Potential for outside collaboration
- Rust
  - Pro: Clear path to implementing FSW code
  - Con: Programming difficulty
- F Prime
  - Opportunities for refactoring
  - Potential for new back ends
  - Potential for adding state machines
- DSLs
  - New: F Prime, rule-based
  - F Prime integration: HSM, Monitoring

Publications

[A] Robert Bocchino and Klaus Havelund, "RustSpot: A Little Rust, for Explanation", in preparation.

[B] Robert Bocchino and Klaus Havelund, “A Lambda Calculus for Embedded Programming”, in preparation.

[C] Manfred Broy, Klaus Havelund, Rahul Kumar, and Bernhard Steffen, “Towards a Unified View of Modeling and Programming”, Leveraging Applications of Formal Methods, Verification and Validation, ISoLA’18, Lecture Notes in Computer Science volume 11244, pp. 3-21, Springer, Limassol, Cyprus, Oct 30-Nov 1, 2018.

[D] Klaus Havelund and Robert Bocchino, “Component-based Programming in Scala - Can Embedded Programming be Made Easy?”, August, to be submitted, 2020.

[E] Klaus Havelund and Rajeev Joshi, “Modeling with Scala”, Leveraging Applications of Formal Methods, Verification and Validation. ISoLA’18, Lecture Notes in Computer Science volume 11244, pp. 184-205, Springer, Limassol, Cyprus, Oct 30-Nov 1, 2018.

[F] Daniel Tellier, Meyer Millman, Brian McClelland, Kate Beatrix Go, Alice Balayan, Michael J Munje, Kyle Dewey, and Nhut Ho, Klaus Havelund, and Michel Ingham, “Towards the Hierarchical State Machine Oriented Proteus Systems Programming Language”, to appear in AIAA ASCEND’20, 2020.

References

[1] Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki, "The Essence of Dependent Object Types", Lecture Notes in Computer Science, Volume 9600, Springer, 2016.

[2] Robert Bocchino, Timothy Canham, Garth Watney, Leonard Reder, and Jeff Levison, “F Prime: An Open-Source Framework for Small-Scale Flight Software Systems, In 32nd Annual AIAA/USU Conference on Small Satellites, Utah State University, 2018.

[3] Steve Klabnik and Carol Nichols, "The Rust Programming Language", No Starch Press, Inc., Mozilla Corporation and and the Rust project Developers, June 26, 2018.

[4] Martin Odersky, Lex Spoon, and Bill Venners, "Programming in Scala", Artima Incorporation, USA, 3rd edition, 2016.

[5] Richard Whaling, "Modern Systems Programming with Scala Native", Pragmatic Bookshelf, 2020.