Towards the Hierarchical State Machine Oriented Proteus Systems Programming Language

Daniel Tellier*, Meyer Millman†, Brian McClelland‡, Kate Beatrix Go§, Alice Balayan¶, Michael J Munje‖, Kyle Dewey**, and Nhut Ho††
California State University, Northridge, Northridge, CA, 91330

Klaus Havelund‡‡ and Michel Ingham§§
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

In space applications, hierarchical state machines (HSMs) are often used for writing simulation and control software, including that of the Curiosity rover. At the Jet Propulsion Lab (JPL), multiple programming languages have been developed specifically for writing HSM-based software, and these have been used in practice. However, we observe that the existing languages developed have significant issues with one or more of usability, performance, and safety, making them problematic for HSM-based development. To address these concerns, we are taking lessons learned from these languages and developing a new programming language named *Proteus*. Proteus builds HSM support directly into the language, and permits complex HSMs to be defined which communicate with each other. Proteus is designed with a look and feel similar to C/C++, making it usable and overall approachable for use by JPL systems engineers. Proteus itself compiles to C++, allowing it to fit easily into existing development toolchains, and making it amenable to embedded real-time systems. To ensure that Proteus will be of use to its target audience, it is being developed iteratively through a series of prototypes which are regularly evaluated by key stakeholders at JPL, allowing us to refine its design if we go off course. While Proteus is still very young in its development, we demonstrate its basic viability on an example utilizing multiple independent HSMs communicating with each other, and relevant execution traces. In the future, we plan to apply Proteus to larger HSMs taken from real space applications, and many additional relevant features are planned.

I. Introduction

Hierarchical State Machines (HSMs) [1] are commonly used to design, implement, and reason about complex software systems to be deployed in space, including the Curiosity rover's flight software [2]. In particular, HSMs are used for the simulation of software models to be implemented, as well as the actual implementation of some software, such as control systems. While HSMs are a popular development model, in order to practically scale to large systems, there is a need for special tooling and programming language support. In this paper, we introduce *Proteus*: a programming language under design at the Jet Propulsion Lab (JPL) for HSM-based software development.

While there are multiple preexisting HSM-based language design efforts, we observe that these all have significant weaknesses. For example, a typical development approach involves the use of graphical modeling tools to draw HSMs and then automatically generate code from their internal representation. While visualization is recognized as essential, this approach often results in opaque code which bears little resemblance to the visualization, making the output code difficult to inspect and reason about. Closer to Proteus' design, other approaches have involved the use of purpose-built textual Domain-Specific Languages (DSLs) [3]. At least two textual HSM DSLs have already been developed at JPL,

*Undergraduate Student, Department of Computer Science
†Undergraduate Student, Department of Electrical and Computer Engineering
‡Masters Student, Department of Computer Science
§Undergraduate Student, Department of Computer Science
¶Undergraduate Student, Department of Computer Science
‖Undergraduate Student, Department of Computer Science
**Assistant Professor, Department of Computer Science
††Professor, Department of Mechanical Engineering
‡‡Senior Research Scientist, Laboratory for Reliable Software
§§Chief Technologist, Systems Engineering Division

including one embedded within the Scala general purpose programming language [4], and another wherein programmers mix fragments of DSL and C code. However, we argue that neither of these DSLs are appropriate for their purpose. Notably, the Scala-based DSL is only suitable for simulations, as Scala's reliance on garbage collection precludes it from the real-time embedded software commonly seen in space applications. While the C-based DSL is suitable for both simulations and actual implementation, it offers essentially no safety guarantees, making it ill-suited for the development of mission-critical software. Overall, we argue that existing HSM-based languages are not appropriate for the very domains they target. In contrast, with Proteus, we are developing a language which is suitable for both simulations and actual implementation, without compromising on safety.

Like existing HSM-based DSLs, Proteus offers built-in support for representing state machines, states, external events, and state transitions. Proteus also has integrated actor support, and allows for the definition of large systems composed of multiple HSMs which asynchronously communicate with each other. However, unlike existing HSM-based DSLs, Proteus is designed from the ground-up with both safety and performance in mind. Unlike C/C++, Proteus programs are memory-safe by construction, and are devoid of undefined behavior. As a result, many common program bugs endemic of C/C++ are simply impossible to have in Proteus. This memory safety is granted by Proteus' fundamental design, in contrast to expensive runtime features like garbage collection which preclude real-time embedded environments.

Key to Proteus' design is that it is intended to look and behave similarly to C++, and it even compiles to C++. Much existing development is already performed in C++ by systems engineers, so going for C++'s look and feel helps ensure it can be adopted. By compiling to C++, Proteus is amenable to real-time systems, as can also easily integrate with existing C++ development toolchains. We have special emphasis on generating C++ which is easily understandable, so that systems engineers unfamiliar with the Proteus code but familiar with C++ can nonetheless understand what compiled Proteus code does. However, unlike with writing C++ by hand, Proteus code is guaranteed memory safe, and has a plethora of HSM-related features not easily implemented directly in C++ itself.

The design and development of a novel programming language is a massive undertaking, especially one with such an atypical feature set. While individual features may be understood in isolation, the combination of features can result in unexpected interactions and unintended, emergent behavior from these interactions. On the one hand, the language is very experimental, and acts as a proving ground for new ideas. On the other hand, we need to have high confidence that the language is intuitive, or else it is unlikely to be used. A very real concern is that the language will deviate from user expectations. Complicating matters, user expectations may be vague or even intangible; users may not know what they want (or do not want) until they see it in front of them.

Towards ensuring that language development is moving in the right direction, our strategy is to gather a series of moderately-sized HSM implementations which were previously developed at JPL. Proteus is being developed as a series of prototypes, and we will re-implement these HSMs in Proteus for each Proteus prototype. If this re-implementation proves difficult (e.g., cannot be naturally expressed, time-consuming to write, hard to reason about), we will iterate on Proteus' design to simplify re-implementation. Once this re-implementation process is satisfactory, we can show the Proteus prototype to key stakeholders at JPL and solicit feedback from them early on, via interviews, focus groups, and surveys. By showing potential users a prototype, we expect this will enable very specific, actionable feedback. We plan to repeat this process for every major feature we plan to incorporate into Proteus.

Overall, the contributions of this paper are as follows:

1) A discussion of Proteus' core features which enable HSM-based development (Section III), as well as how these features are compiled to C++ (Section IV).
2) A discussion of the iterative process through which we are designing Proteus (Section V)
3) The application of Proteus to small HSMs, demonstrating basic viability (Section VI)

II. Background and Related Work

In this section, we discuss background information necessary to understand Proteus' design and feature set, as well as related work in HSM-based programming language design and development. We start with a discussion of actors.

A. Actors

Actors [5] are a parallel programming model wherein computation is broken up into different independent components called *actors*. Each actor executes its code sequentially, but multiple actors can execute in parallel with respect to each other. In addition to maintain its own executable code, each actor also maintains internal state upon which this code acts. Notably, this state is directly accessible only within the same actor; actors cannot directly access each other's state. The only way for actors to communicate with each other is by sending *messages* to each other. Messages can contain arbitrary data, allowing for very specific messages to be sent and received. An actor may send a message to any other actor, and similarly an actor may receive a message from any other actor. In practice, actors generally wait for an incoming message, do whatever computation is necessary to respond to the message (which may entail sending out other messages to other actors), and then repeat the process indefinitely.

While actors themselves are an independent concept from HSMs, we argue that actor support is practically necessary in an HSM-based language. For example, the HSM-based Curiosity rover control software requires well over 100 threads operating in parallel [2], and each of these threads can be viewed as a separate actor.

B. HSM Background

HSMs have been in use for over 30 years [1], and are commonly used in space applications [2]. HSMs are a variation of finite state machines, wherein states, and even entire state machines, can be embedded within other states. Any variables introduced in a parent state are accessible to nested states, and closely related behavior can be grouped inside of larger states. This helps modularize software design, and also minimizes redundancy in specifications. Notably, behavior from parent states (e.g., state transitions under certain input events) is inherited by child states, eliminating the need to wholly duplicate behavior in multiple states.

HSMs transition between states in response to input *events*. From the standpoint of the actor model, events are indistinguishable from messages, and Proteus exclusively uses the word "event" to refer to messages and events. With this in mind, when a Proteus actor receives an event, it can trigger a state transition, along with the execution of user-defined code.

In practice, JPL makes heavy use of HSMs in space and flight software, for good reason. While HSMs are less expressive than arbitrary code, they are theoretically much simpler to both manually and automatically reason about. This makes them amenable to automated verification and validation (V&V) techniques like model checking [6] and theorem proving [7].

C. Existing HSM Languages

We are aware of two programming languages which were specifically designed for HSM-based development, herein called ScalaHSM [4] and TextHSM. Both were developed internally at JPL and have seen ongoing use there. ScalaHSM is built as an *internal* DSL embedded into the Scala programming language itself; it is viewable as a Scala programming library. While ScalaHSM has been useful for simulations, it suffers from two major issues. For one, since Scala itself is a garbage-collected language, ScalaHSM is not appropriate for the actual implementation of any real-time code. More specifically, the garbage collector can impart sizable delays at unpredictable times, which could occur at critical moments. Additionally, since ScalaHSM is really just a Scala library, its users must be familiar with Scala. This is unfortunately not the case for most systems engineers, who tend to have a background in C/C++. Scala is a significantly different language from C/C++, making it a difficult and time-consuming task for a typical system engineer to pick up and use ScalaHSM. As a result, ScalaHSM is not very approachable for its target audience.

In contrast to ScalaHSM, TextHSM is built as an *external* DSL, meaning that its syntax is separate from any other programming language. Like traditional programming languages (and Proteus itself), TextHSM is compiled to another language, specifically C. This use of C makes it appropriate for actual implementations, not just simulations. Like Proteus, TextHSM presents HSM-specific features to the user. However, TextHSM is merely a very thin wrapper on top of C. TextHSM effectively has "holes" where users directly write C code, and TextHSM itself simply passed the contents of these holes along into the compiled product. There is no checking that this C code is correct, or even syntactically valid. For this reason, we consider TextHSM to be inherently unsafe and thus risky for any mission-critical development. TextHSM makes no attempt to ensure the generated code is correct, or even meaningful.

III. Proteus Core Features: A User's Perspective

This section discusses the core features of Proteus from a user's standpoint, with special emphasis on those features which enable HSM-based development. We introduce these features through the use of example HSMs which are to be implemented in Proteus. We first discuss this example.

A. Example

Our example is based on a simplified scenario involving separate, but nonetheless related, power and camera controls for a space vehicle. The corresponding HSMs are shown in Figure 1. The `Power` HSM operates with two events

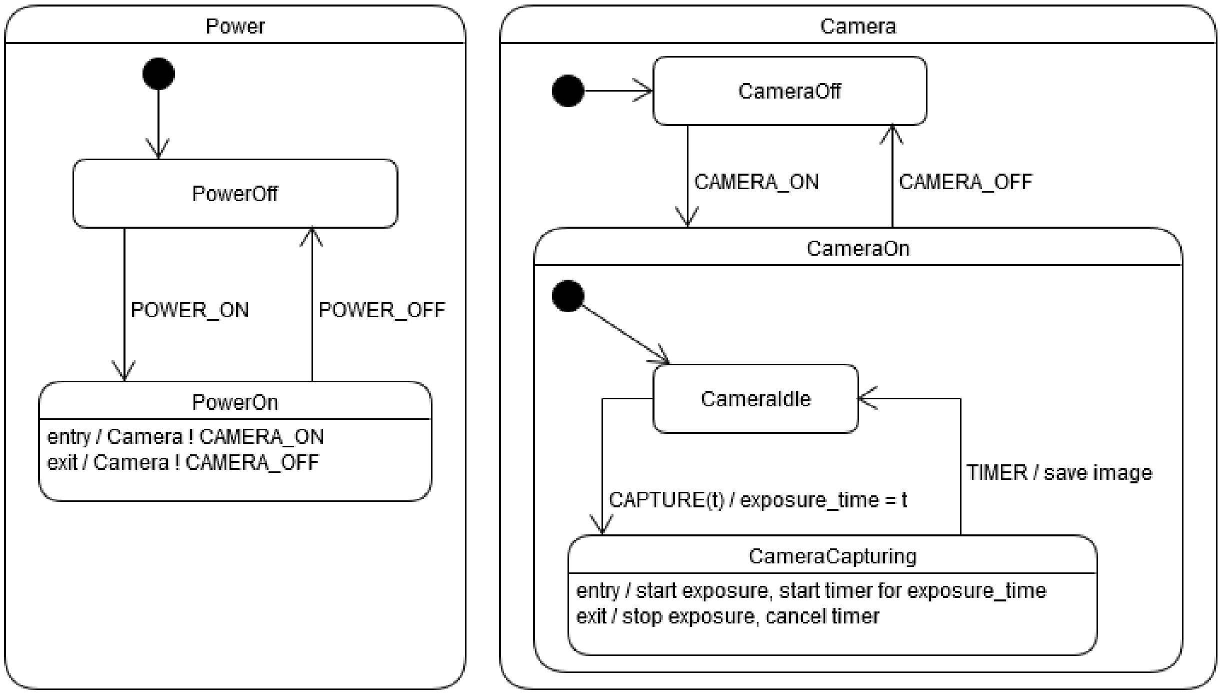

Fig. 1 Example HSMs for managing power and camera control.

POWER_ON and POWER_OFF, which are assumed to be provided from ground control or some other external source. When a POWER_ON event is received by the POWER HSM, POWER transitions to the PowerOn state and subsequently sends a CAMERA_ON event to the Camera HSM. Similarly, if Power receives a POWER_OFF event, it will transition back to the PowerOff state, and send a CAMERA_OFF event to the Camera HSM.

As for the Camera HSM, when it receives a CAMERA_ON event, it will transition to the CameraOn state, ultimately entering the CameraIdle state. From here, it will wait for a CAPTURE event which is bundled with an exposure time t, which is saved in a variable. Like POWER_ON and POWER_OFF events, CAPTURE events are assumed to come from ground control or some other external source. Upon receiving a CAPTURE event, the exposure time is extracted from the CAPTURE event into t, and saved into the variable exposure_time. From here, Camera transitions to the CameraCapturing state. In the CameraCapturing state, a timer is started for exposure_time, and a camera exposure is started. Once the exposure time is reached, the timer will send a TIMER event to Camera, causing the image captured to be saved, and transitioning execution back to the CameraIdle state. At any point, if the CAMERA_OFF event is received by Camera, a transition to CameraOff will occur. Additionally, if CAMERA_OFF is received while in the CameraCapturing state, the exit action of CameraCapturing will be executed, which will stop the camera's exposure and cancel any timers.

The Proteus code in Figure 2 implements the HSMs in Figure 1. This code is presented in two columns for space. We dedicate the rest of this section discussing the various Proteus features used in this example, subdivided by feature. This discussion is strictly from the user's perspective; how these features are compiled to C++ is later covered in Section IV.

B. Actors

Actors are at the heart of all Proteus execution, and all executable code must be within an actor. Actors are all named, and run for the duration of the program. To specify that code is to be executed within an actor, the user must enclose their code in an actor block. actor blocks are only valid at the top level of a program; they cannot be nested in any other construct, including other actors. Semantically, at runtime, the code contained in each actor block runs in a separate thread.

The Proteus example in Figure 2 uses two actors, one for the Camera HSM, and another for the Power HSM. Since each HSM is defined in a separate actor, each HSM can run in parallel with the other.

Variables can be declared inside of actors like so:

```
event POWER_ON{};
event POWER_OFF{};
event CAMERA_ON{};
event CAMERA_OFF{};
event CAPTURE{int};
event TIMER{};

actor Camera {
  statemachine {
    initial CameraOff;
    state CameraOff {
      on CAMERA_ON{} {
        go CameraOn {}
      }
    }
    state CameraOn {
      initial CameraIdle;
      int exposure_time = 0;
      on CAMERA_OFF{} {
        go CameraOff {}
      }
      state CameraIdle {
        on CAPTURE{t} {
          go CameraCapturing {
            exposure_time = t;
          }
        }
      }
      state CameraCapturing {
        int timer_id = 0;
        entry {
          timer_id =
            start_timer(exposure_time);
          start_exposure();
        }
        exit {
          stop_exposure();
          cancel_timer(timer_id);
        }
        on TIMER{} {
          go CameraIdle {
            save_image();
          }
        }
      }
    }
  }
}
```

(a) Proteus implementation of events and `Camera` HSM from Figure 1.

```
actor Power {
  statemachine {
    initial PowerOff;
    state PowerOff {
      on POWER_ON{} {
        go PowerOn {}
      }
    }
    state PowerOn {
      entry {
        Camera ! CAMERA_ON{};
      }
      exit {
        Camera ! CAMERA_OFF{};
      }
      on POWER_OFF{} {
        go PowerOff {}
      }
    }
  }
}

func start_timer(int ms) -> int {...}
func cancel_timer(int timer_id) {...}

func start_exposure() {
  print("start_exposure\n");
}

func stop_exposure() {
  print("stop_exposure\n");
}

func save_image() {
  print("save_image\n");
}
```

(b) Proteus implementation of `Power` HSM from Figure 1, along with relevant helper functions.

Fig. 2 Proteus implementation of HSMs from Figure 1

```
actor MyActorWithVariables {
  int first;
  bool second;
}
```

All code defined within the actor has these variables in scope. These variables are only accessible within the actor; actors cannot directly access or modify each other's variables. Actor-level variables stay in memory for the duration of the program.

1. Actor Restrictions

Actors live for the duration of the program; all actors are started at program start, and similarly all actors are terminated at program end. The names of all actors are statically known, and actors cannot be dynamically started or terminated at runtime. While this is more restrictive than the usual actor model, this reflects how actors are intended to be used in practice in the systems we are targeting. Each actor either contains control code which is expected to constantly run, or serves as an interface to a fixed piece of hardware. With this in mind, there is no need for dynamic creation or termination. Moreover, while such dynamic features would make Proteus more flexible, they would also make Proteus code more difficult to statically reason about, which is contrary to our design goals.

C. Events: Definition, Sending, and Receiving

True to the actor model, actors can only communicate by sending messages to each other [5]. To be consistent with HSM terminology, messages in Proteus are called "events". Events can contain arbitrary data, though the type of the data must be declared before use in Proteus. To send an event, the user must first define what events are valid, using the `event` reserved word. Event definition is only legal at the top-level of a program. A number of events are declared at the top of Figure 2a, including `POWER_ON`, `POWER_OFF`, and `CAPTURE`. The `CAPTURE` event takes one datum, namely a single integer; all other events contain no data. While not shown in this example, events can take multiple data, as with:

```
event MULTI_DATA_EXAMPLE {int, bool};
```

Events can be sent to an actor using `!`, as is done in Figure 2b with:

```
Camera ! CAMERA_ON{};
```

The above snippet sends a `CAMERA_ON` event to the `Camera` actor. While not shown in this example, data can be sent along with an event, as with:

```
Receiver ! MULTI_DATA_EXAMPLE { 5, true };
```

The number and types of data provided must match up with the declaration; any mismatch results in a Proteus compile-time error.

To receive an event, an actor must use the `on` reserved word. `on` is used extensively in Figure 2, specifically in states. States are covered in more detail in Section III.D, though both states and actors use `on` to receive events. If an event contains data, the data can be bound to a new local variable, as is done in Figure 2a with:

```
on CAPTURE(t) {
  ...
}
```

The data is bound to the newly-declared local variable `t`, which is implicitly of type `int` (known from the declaration of `CAPTURE` events). When an event is received, the code within the `on` block is executed.

Semantically, actors are fundamentally event-driven; they wait for an input event to be received, execute an event handler, and repeat this process indefinitely. While a Proteus program can have an arbitrarily large number of actors defined, each actor operates sequentially. As such, an actor will fully process one input event before attempting to receive another one. With this in mind, in Figure 2b, if `Power` received multiple `POWER_ON` events, they would be processed sequentially. The first received would be processed immediately, whereas all others would be stored in an internal event queue hidden inside `Power`. Once the first `POWER_ON` event is processed, only then could the next `POWER_ON` event be removed from the queue for processing.

An actor may receive different kinds of events, as with:

```
actor MultiEventExample {
  on FOO {
    // code here executed if a FOO event is received
  }
  on BAR(a, b) {
    // code here executed if a BAZ event is received
  }
}
```

As shown above, `MultiEventExample` handles both `FOO` and `BAR` events. Event processing is still sequential; if `MultiEventExample` is processing a `FOO` event, then any other received events will be internally stored in an event queue, even if a `BAR` event is received. Semantically speaking, actors wait for any incoming event to happen, and will execute the code in the `on` block corresponding to whatever event was received.

At compile time, Proteus checks that any sent or received events are declared with `event`. Similarly, Proteus checks that for each event sent, the target actor has an `on` block specifically for the sent event. If an undeclared event is sent or received, or if an event is sent to an actor which does not have a corresponding `on` block, it results in a compile-time error.

Local variables may be defined within `on` blocks. These exist in memory only while the `on` block executes.

D. HSM and HSM State Definition

Proteus has language-level support for HSMs. As Figure 2 shows, entire state machines are declared with the `statemachine` reserved word, and individual states are specified with the `state` reserved word. `statemachine` can only be used at the top-level within an actor. The initial state in a state machine or any child state must be indicated with the `initial` reserved word, as is done in Figure 2a with `initial CameraOff;`.

Actors which contain state machines execute them sequentially, one state at a time. That said, since states can be nested within other states, it is possible for execution to be present in multiple states at the same time. We refer to the most deeply nested state currently being executed as the *active state*. While sequential execution occurs within a state machine, multiple state machines can execute in parallel with respect to each other if they are defined in separate actors.

As shown with the `CameraOn` state with the `exposure_time` variable in Figure 2a, variables can be defined on states, and these variables are accessible from within the declared state and also in any child states of that state. `on` blocks can similarly be nested within states. The semantics of `on` blocks become more complex when nested states are involved. For example, consider the `CameraCapturing` state's `on` block, which accepts `TIMER` events. While `CameraCapturing` does not explicitly handle `CAMERA_OFF` events, its parent state `CameraOn` *does* handle `CAMERA_OFF` events. As such, even if `CameraCapturing` is the active state, the HSM can still respond to an input `CAMERA_OFF` event. In case of a `CAMERA_OFF` event, an automatic transition will be performed to the outer `CameraOn` state (triggering the `exit` action on `CameraCapturing`), wherein the `on` block for the `CAMERA_OFF` event will be executed. More details on state transitions and exit actions are in Sections III.E and III.F.

While not shown in Figure 2, a child state may define an `on` block that accepts the same kind of event as a parent state. In this case, if the child state is the active state, the child's `on` block will be executed instead of the parent's `on` block. If the active state does not have a corresponding `on` block, it will consult the parent's `on` blocks, and so on until it finds a state which will accept the event. As part of this process, `exit` handlers are executed in sequence. It is statically guaranteed that either the active state or one of its parents can accept an input event, per the same static guarantee that prevents sending an event to an actor which lacks a corresponding `on` block.

Variables can be defined on states, known as *state variables*. For example, `exposure_time` is a state variable on `CameraOn` in Figure 2a. State variables are in scope while execution is in the corresponding state, be it the active state or a parent of the active state. Semantically, such variables only need to consume memory while the state is being executed. However, in order to simplify compilation and make compiled code easier to reason about, such variables instead are effectively lifted to the corresponding actor and are thus in memory for the duration of program execution; see Section IV for details.

E. State Transitions

Compared to general actors, `on` blocks behave somewhat differently in states. Specifically, upon receiving an event, an HSM will usually change which state it is in via a state transition. State transitions are performed via `go`, as shown extensively in Figure 2. go specifies the state to transition to, which must be contained within the same state machine. Additionally, go takes a block of code to execute just before making the state transition.

While not shown in Figure 2, there is also a conditional version of go called goif, which will only transition to a state if an arbitrary Boolean condition is true. As with a typical if statement, goif can be chained along with else goif clauses. Also like if, the alternatives to goif are tried in sequential order, and the first condition which is true (and only that condition) has its corresponding code block executed. If a final unconditional else go is not provided, then no state transition occurs, and execution remains in the same state.

F. Entry and Exit Actions

Per the usual definition of HSMs, states can also optionally define entry and exit actions to be performed whenever execution enters or leaves a state. These actions are defined with the entry and exit reserved words, as shown in Figure 2. Because of nested states, the behavior of entry and exit is not always straightforward. Specifically, when transitioning out of a state, the exit actions are executed in order of most deeply-nested to least deeply-nested. Once all appropriate exit actions are performed, entry actions are performed, starting from the outermost state being entered and moving to the innermost state. Both of these behaviors are from the usual HSM semantics of state transitions.

G. Functions

A number of helper functions are declared at the end of Figure 2b, using the func keyword. The function bodies of start_timer and cancel_timer have been elided to keep the example concise. Similarly, the bodies of start_exposure, stop_exposure, and save_image in a real implementation would interact with underlying camera hardware, though in our example they merely print string literals to the user console. As shown, functions can take parameters, and can optionally return values. Specifically, start_timer takes an integer and returns an integer; the notation -> int means that it returns an integer. In contrast, none of the rest of the functions return anything, as they do not have any return types annotated. We opted to simply not have a return type specified, as opposed to introducing a pseudo-type like void, similar to Rust [8].

H. Methods

While Proteus is not an object-oriented programming language, it nonetheless supports a concept of methods. Syntactically, Proteus methods are merely functions which are defined at either the actor or state level. Unlike typical functions, methods have access to any actor or state variables which are in the same scope level as the method. Similarly, methods are only available to be called from with the actor or enclosing state. As an example, the Proteus code below defines and uses methods:

```
actor MyActor {
  int actorLevelVariable = 0;

  func onActor(int x) {
    actorLevelVariable = x;
  }

  statemachine {
    state Outer {
      int outerLevelVariable = 1;
      func onOuter() -> int {
        onActor(outerLevelVariable + 1);
        return actorLevelVariable + outerLevelVariable;
      }

      state InnerAlpha {
        int alphaVariable = 2;
        func onAlpha() -> int {
          return onOuter() + alphaVariable;
        }
```

```
      }
      state InnerBeta {
        int betaVariable = 3;
        func onBeta() -> int {
          return onOuter() + betaVariable;
        }
      }
    }
  }
}
```

Any attempt to call a method that is not part of the enclosing scope will result in a compile-time error. For example, in the `onBeta` method above, the `onAlpha` method cannot be called, as `onAlpha` is defined in another non-parent state. Similarly, while `onAlpha` can be called from within the `InnerAlpha` state, `onAlpha` cannot be called at the `MyActor` level.

IV. Compilation of Proteus Core Features

In this section, we discuss out requirements of compiled code, and describe how the features introduced in Section III are compiled. We specifically focus on what this translation *is*, as opposed to how it is performed.

A. Requirements of Compiled Code

From our discussions with scientists and systems engineers at JPL, there are a number of requirements we have of our compiled code. These requirements have helped define exactly what are compiled translations are. We describe these below.

1. Strongly Preferred Compilation Target: C/C++

The majority of flight-ready software at JPL is written in C or C++, and systems engineers are generally familiar with these languages. For Proteus to fit into existing development toolchains well, it was clear that the target language needed to be either C or C++. We ended up selecting C++ as a target language, given its additional features on top of C.

C++ was also chosen because it needs no complex runtime system like the Java Virtual Machine. This makes it appropriate for embedded and hard real-time systems, both of which are commonly seen in space software. Lastly, the choice of C++ meant that systems engineers could at least theoretically modify compiled Proteus code, in case they wanted to perform actions possible in C++ but not in Proteus.

2. No Heap Memory Usage

Due to the embedded nature of possible Proteus compilation targets, heap memory might not be available. Additionally, code that uses heap memory is typically more difficult to reason about, both from a human and a machine standpoint. With heap memory in play, the number of possible software states is bounded by the size of the heap, which can be prohibitively large for analysis. For these reasons, Proteus intentionally lacks the ability to manipulate heap memory. Similarly, compiled Proteus code cannot somehow use any heap memory during its execution.

3. Output Should be Human-Understandable

Early in our discussions, it became clear that the compiled code must be understandable by systems engineers, not merely readable. Theoretically, any C++ code is human-readable, but this does not mean that humans would easily be able to understand what the code does. With understandable code, a systems engineer should be able to easily make changes to the C++ code, particularly if they were familiar with the input Proteus code. Additionally, a systems engineer should be able to convince themselves that the code works correctly, even if the systems engineer is unaware of the input Proteus code.

Understandability is an important property to attain, albeit difficult to precisely define. Ultimately, it is up to JPL systems engineers to say whether or not the output code is understandable or not. We have not yet reached the stage where we show the systems engineers output code and qualitatively ask them to assess the code's understandability. However, in the meantime, we have internally been doing these sorts of understandability assessments. As guiding principles, we are avoiding any compilation strategies which would heavily mangle the code or require the reader to understand complex, unfamiliar algorithms.

4. Compiled Code Must be Safe

Given that compiled Proteus code may be used in safety and mission-critical scenarios, it is of the utmost importance that the code does not perform any unintentional, unreliable, or flat-out incorrect actions. Preventing such undesirable behavior in general is a complex and difficult problem which we plan to address more completely in the future. However, for now, we restrict ourselves to two smaller, though nonetheless critical properties:

- Compiled code must be memory safe. That is, code can only access memory which was already initialized, and the underlying value being accessed in memory corresponds to whatever is intended to be accessed.
- Compiled code must be free of undefined behavior. C++ has a number of undefined behaviors (e.g., null pointer dereference and array access out of bounds) wherein programs may exhibit *any* possible behavior upon executing undefined behavior, so programs which trigger undefined behavior are almost assuredly incorrect.

B. Actors

Multiple actor libraries already exist for C++ (e.g., CAF [9], Rotor [10], and libagents [11]). However, to the best of our knowledge, all existing actor libraries we found manipulate heap memory, violating a central design constraint. Additionally, most libraries had additional features which we did not need, and were overall more complex than necessary for the task at hand. Lastly, upon consideration, the amount of work needed to write our own solution appeared to be relatively small. For these reasons, we chose to implement our own actor library.

The Proteus actor library consists of a generic `Actor` class parameterized on two types `S` and `E`, respectively for state and event. `Actor<S,E>` contains an event queue of `E`s and a pointer to the current state of type `S`. The event queue is synchronized to protect against simultaneous access on multiple threads. There are two public methods:

```
void initialize(S* initialState)
void receiveEvent(E const& event)
```

`initialize` does the initial transition and then launches the actor's event loop on a new thread. `receiveEvent` simply enqueues an event.

The library in turn expects very little of types `S` and `E`. In C++, the constraints on generic type parameters are implicit and based on usage (so-called "duck-typing"). Things of type `E` are only copied to and from the queue and passed to functions, so they only need to be default-constructable and copyable. Things of type `S` have the following methods called:

```
void init()
S* next(E const&)
```

The `init` method is assumed to do any work needed to transition to the initial state. The `next` method is assumed to handle the given event, transition if necessary, and return the new state.

The actor's event loop waits for the queue to contain events, dequeues one, passes it to the `next` method on the current state, and replaces the current state with the result. If no events are in the event queue, the actor will block until an event is received. The details of how events are dispatched can be seen in Section IV.C.

Each use of `actor` in Proteus ultimately creates a single instance of `Actor<S,E>` where `S` is the base state type and `E` is the event union type (see Section IV.C for details). This instance is global and static and exists for the duration of the program. At the start of the program, each actor has its `initialize` method called with a pointer to its initial state (for actors without HSMs, the initial "state" is the actor struct). When an actor send an actor a message, it calls the `receiveEvent` method on its static instance. This is the only way that actors communicate with each other.

In addition, creating an `actor` in Proteus also creates a C++ `struct`. The items defined in the actor correspond directly to items in the `struct`: instance variables become instance variables, methods become methods, and `on` blocks similarly become methods. We use C++'s method overloading to distinguish between the separate `on` blocks, where each method bears the same name (specifically, `on`) but takes an input event of a different type.

C. Events

Here we discuss how events are represented, sent, and received. We divide our discussion accordingly.

```
event FOO{};
event BAR{int};
event BAZ{bool, int}.
```

(a) Proteus code

```
struct prot_event_FOO {};
struct prot_event_BAR {
  int _0;
};
struct prot_event_BAZ {
  bool _0;
  int _1;
};
```

(b) C++ code

Fig. 3 Proteus event code along with C++ translation.

1. Event Representation and Sending Events

Each Proteus event maps to a separate `struct` in the output C++ code. These individual `struct`s contain all data that is specific to each event. An example translation is shown in Figure 3.

As Figure 3 shows, names are slightly mangled to avoid any naming conflicts with other emitted C++ code. Field names are given names based on their position of the definition. Note that since users cannot directly access these fields (i.e., `on` blocks extract them into new local variables), more descriptive names are unnecessary.

While this event representation works for individual events, there is a need to collapse all events into a single event type when working with our actor library. This is specifically necessary for `receiveEvent` from Section IV.B, which takes a single generic event type `E` to be stored in the actor's event queue. Inheritance and polymorphism are typically employed in C++ for addressing this sort of issue, wherein a base class `Event` would be defined with one subclass per specific kind of event. However, we intentionally opted not to go this route, ultimately because of concerns with memory. For polymorphism to work, we would need to work with *pointers* to events, not events themselves, as each event can have different sizes. Heap allocation is usually used for this purpose, but given our requirements, we cannot use heap allocation.

Rather than take the polymorphism approach, we instead opted to use a `union` type in C++, allowing us to put the memory allocated for an event directly in a `struct`. This `struct` itself could then be passed around on C++'s stack, which is more naturally copied into place in the actor. This `struct` additionally needs a tag indicating what the specific kind of event the `union` represents; the tag is represented with an `enum`. This is shown below, building on the definitions from Figure 3a.

```
struct prot_impl_Event {
  enum class Tag {
    tag_FOO,
    tag_BAR,
    tag_BAZ
  };

  Tag tag;

  union {
    prot_event_FOO union_FOO;
    prot_event_BAR union_BAR;
    prot_event_BAZ union_BAZ;
  };

  prot_impl_Event(prot_event_FOO e) : tag(tag_FOO), union_FOO(e) {};
  prot_impl_Event(prot_event_BAR e) : tag(tag_BAR), union_BAR(e) {};
  prot_impl_Event(prot_event_BAZ e) : tag(tag_BAZ), union_BAZ(e) {};
```

With the above code in mind, `receiveEvent` in our actor library takes a `prot_impl_Event`. Each of the constructors of `prot_impl_Event` takes a different specific kind of event, and merely copies the event into the `union` and appropriately sets `tag` for the corresponding event.

With event representation in mind, sending events reduces to making the right `receiveEvent` call. For example, with `MyActor ! FOO {}`, this will translate to the following:

```
prot_event_FOO fooEvent();
prot_impl_Event event(fooEvent);
proto_actor_MyActor.receiveEvent(event);
```

2. Receiving Events

`receiveEvent` puts the input event into the corresponding actor's event queue. Given that no heap allocation is permitted, event queues have statically-known, fixed sizes; it is considered a critical system fault if the event queue size is ever exceeded at runtime. Currently this will crash the program, but we plan for more graceful handling in furture versions of Proteus.

Once the event is in the event queue, if the corresponding actor was blocked on an input event, it will be unblocked. From here, the actor will look at the `tag` of the corresponding event, and then call the appropriate `on` method for the tag. From here, the process repeats indefinitely.

D. HSMs, States, and Transitions

HSM states are represented in C++ similarly to actors; each ends up being represented with a separate class, and instances of these classes are all statically allocated. HSMs themselves become states with this representation. However, unlike actors, classes for individual states will extend other state classes, in a manner that mirrors the inheritance of states in the HSM. Methods for `on` blocks are similarly inherited, and these methods will be overridden if a child state provides its own `on` block definition for a given event.

That all said, there is a key distinction in the `on` method between arbitrary actors and HSMs. With actors, `on` returns `void`; that is, once actors dispatch on an event, they simply go to the next event. However, with HSMs, there is additionally a concept of a next state. That is, when `on` is executed from within an HSM state, the HSM may transition to a different state. To perform this transition, `on` for HSMs returns a pointer to the instance corresponding to the next state. With this in mind, `on` for HSMs has a signature like the following:

```
virtual prot_impl_State* on(prot_event_FOO const&);
```

. . . where `prot_impl_State` is the base class for all states. A separate `on` method is defined for each specific kind of event the HSM can accept.

With the above `on` definition in mind, the driver loop for HSMs looks a bit different than for typical actors, as illustrated below:

```
prot_impl_State curState = initialState;
while (true) {
  prot_impl_Event curEvent = readEvent();
  switch (curEvent->tag) {
    case tag_FOO:
      curState = curEvent->on(curEvent->union_FOO);
      break;
    case tag_BAR:
      curState = curEvent->on(curEvent->union_BAR);
      break;
    case tag_BAZ:
      curState = curEvent->on(curEvent->union_BAZ);
      break;
  }
}
```

As shown, in addition to processing the current event, `on` returns a pointer to the next state. This process continues indefinitely throughout program execution.

E. Entry and Exit Actions

Entry and exit actions, like event handling, are handled with polymorphism. The base state class contains virtual methods `enter` and `exit` with default bodies that do nothing. When a state defines either of these actions, its implementation class overrides these and inserts any code that is meant to run. During transitions, states that are being left have `exit` called and states that are being entered have `enter` called. If the state doesn't define these actions, the base state's default body is selected by virtual dispatch and nothing happens.

The specific way in which these actions get called is a little more complex, and this relates to the choice we made for event handling. One possible way of handling events is to fully elaborate every event handler for every state, since it is known at compile time what will handle which event for every active state. If done this way, then it's possible for every transition path to be known statically. However, this can create an explosion of code in the C++ event handlers with every case copied into a large switch statement for every state.

Instead, we use C++'s existing virtual functions for this purpose, leading to code that is simpler to both generate and read. We use the following algorithm for executing entry and exit actions, which makes use of one stack of states per HSM. Given states `from` and `to`:

- Every state knows its parent state, its depth in the state tree, and a pointer to a state stack exclusive to its HSM.
- Any time `from` "moves up", we call `from->exit()`.
- Any time `to` "moves up", we push it onto the stack.
- First, move either `from` or `to` up (whichever is deeper) until they're at the same depth.
- Then, move `from` and `to` up simultaneously until they join at the same parent.
- Finally, pop the stack until empty, and for each state `s` popped, call `s->enter()`.

This produces the correct actions in the correct order.

F. Functions and Methods

Top-level Proteus functions are translated directly into top-level C++ functions, with minimal name mangling. Methods are similarly translated directly to C++ methods, either on their corresponding actor class for plan actors, or on their corresponding state class for HSM states.

V. Proteus Design Process

The creation of any new programming language is a huge risk, given the sheer amount of work this entails. There is a real danger that the target audience will not embrace the language for a variety of reasons, ultimately leading to its failure to be adopted and subsequent abandonment. To mitigate these risks, we are employing an iterative design and implementation process, and involving scientists and systems engineers in Proteus' design process.

Specifically, Proteus is being developed as a series of prototypes. At the completion of each prototype, we solicit feedback from key stakeholders at JPL. We then use this feedback to guide the next development cycle of Proteus, and overall repeat the process. By regularly keeping the target audience in the loop, this helps ensure that development never gets too far off track from their expectations.

So far, these iterations have been fairly small and informal, and have only directly involved the co-authored JPL stakeholders. However, these interactions have already led to key language improvements. Notably, in a prior prototype, HSMs, actors, and states were all conflated into the same construct which had relatively unintuitive semantics. Most importantly, the behavior of this construct was not immediately obvious to people familiar with HSMs, even though it was intended to represent an HSM state. Since then, we have separated out this one concept into the states, actors, and HSMs previously seen in this paper.

Notably, with this prototype-driven development model, we have found that the prototypes do not need to be complete to be useful. As of this writing, we are only now reaching the point where any Proteus code can be directly executed by itself, as actors were only recently introduced. Nonetheless, even without executable programs, the target audience could still understand what the code *should* do, and this was enough information with which to pivot the language design.

VI. Execution

In this section, we demonstrate Proteus' execution by compiling and running the program from Figure 2. This is shown via illustrative traces of what the programs did during execution.

A. Execution Tracing

The Proteus compiler can be invoked with an additional flag to insert trace messages into the output C++ program. This extra output allows one to trace the execution of the program by the sequence of messages at runtime. Each trace line indicates that a certain action has just happened or is being initiated. For example:

```
Power: send POWER_ON to Camera
```

The name beginning each line is the thread on which the action is taking place; this normally corresponds to an actor, but can be any independent entity such as a timer. The different trace actions are:

- initialize to <State>: The actor is starting up and transitioning to its initial state. `State` is the most-initial state of the HSM's root state (following the chain of initial states). This line will be followed by a sequence of `enter` actions as the transition happens.
- enter/exit <State>: The actor is entering or leaving `State` and is about to execute its `enter` or `exit` blocks. Anything that happens during the execution of this block will follow this line. This trace message is printed regardless of the state actually having an `enter` or `exit` block defined, in order to show the sequence followed during transitions.
- send <Event> to <Actor>: The current thread is now placing `Event` into `Actor's` event queue. The event will be received whenever `Actor` dequeues it.
- handle <Event> via <State>: The actor has dequeued `Event` from its event queue, and the event handler that will be handling it is the one defined in `State`. `State` will either be the actor's current state or a (direct or indirect) parent. Anything that happens during event handling will follow this line.
- transition <State1> to <State2>: As the result of handling an event, the actor has begun transitioning from its current state `State1` to its next state `State2`. `State2` is the most-initial state of the state specified by the event handler, so it may not match what is written in the program. If `State1` is different from `State2`, a sequence of `exit` and `enter` actions will follow.
- start Timer <TimerId>: The actor has started an asynchronous timer. When time elapses, the timer will send a `TIMER` event back to the actor.
- cancel Timer <TimerId>: The actor has stopped the specified timer prematurely; it will not be sending its `TIMER` event.

B. Tracing the Example Program

The example, if left alone, would sit indefinitely in an off state. So in order to test it, we added a third actor, `ENV`, simulating some external source of events. Since this actor was an implementation detail of testing, it was omitted everywhere except for its event trace messages. `ENV` runs a "script" of send- and wait-statements in its HSM root state's entry block. Two different scripts were tested, resulting in two difference execution traces.

Figure 4 shows the first script and its output (separators have been added for readability; each block is the direct result of one "event" from outside, be it initialization, an event from `ENV`, or a timer firing). The following points of correctness can be noted from the trace:

- Initial states are properly followed.
- Entry/exit blocks bracket their states. For example, `Power` sends `CAMERA_ON` and `CAMERA_OFF` to `Camera` upon the entry and exit, respectively, of state `PowerOn`.
- Actors follow the right sequences of entry/exit actions when transitioning between states. For example, when `Camera` transitions from `CameraOff` to `CameraIdle`, it follows the sequence: `exit CameraOff`, `enter CameraOn`, `enter CameraIdle`.
- The `print` messages appear where they should: `start_exposure` upon entering `CameraCapturing`, `stop_exposure` upon exiting `CameraCapturing`, and `save_image` upon handling of the `TIMER` event.
- Events are handled by the correct states and event handling results in the correct transitions.

Figure 5 shows the second script and its output. Figure 5 differs from Figure 4 only in that it does not wait long enough for the capture timer to fire before sending `POWER_OFF`. This changes the path of execution and causes the last block of the trace to differ from before. In Figure 4, `Camera` transitioned from `CameraIdle` to `CameraOff`; in Figure 5, it transitions from `CameraCapturing` to `CameraOff`. It can be seen that it correctly cancels the timer and never receives the `TIMER` event nor calls `save_image`.

While these are simple inputs for a simple example, the simplicity allows one to inspect the output and reason that it is behaving according to the model. This provides some assurance that the compiler is correct, at least for this example.

```
wait 2000
Power ! POWER_ON{}
wait 2000
Camera ! CAPTURE{1000}
wait 2000
Power ! POWER_OFF{}
```

(a) ENV script

```
Camera: initialize to CameraOff
Camera: enter Camera
Camera: enter CameraOff
Power: initialize to PowerOff
Power: enter Power
Power: enter PowerOff
----------------
ENV: send POWER_ON to Power
Power: handle POWER_ON via PowerOff
Power: transition PowerOff to PowerOn
Power: exit PowerOff
Power: enter PowerOn
Power: send CAMERA_ON to Camera
Camera: handle CAMERA_ON via CameraOff
Camera: transition CameraOff to CameraIdle
Camera: exit CameraOff
Camera: enter CameraOn
Camera: enter CameraIdle
----------------
ENV: send CAPTURE to Camera
Camera: handle CAPTURE via CameraIdle
Camera: transition CameraIdle to CameraCapturing
Camera: exit CameraIdle
Camera: enter CameraCapturing
Timer 1: start
start_exposure
----------------
Timer 1: send TIMER to Camera
Camera: handle TIMER via CameraCapturing
save_image
Camera: transition CameraCapturing to CameraIdle
Camera: exit CameraCapturing
stop_exposure
Camera: enter CameraIdle
----------------
ENV: send POWER_OFF to Power
Power: handle POWER_OFF via PowerOn
Power: transition PowerOn to PowerOff
Power: exit PowerOn
Power: send CAMERA_OFF to Camera
Power: enter PowerOff
Camera: handle CAMERA_OFF via CameraOn
Camera: transition CameraIdle to CameraOff
Camera: exit CameraIdle
Camera: exit CameraOn
Camera: enter CameraOff
```

(b) Execution trace

Fig. 4 Script 1 and its resulting trace

```
wait 2000
Power ! POWER_ON{}
wait 2000
Camera ! CAPTURE{1000}
wait 500
Power ! POWER_OFF{}
```

(a) ENV script

```
Camera: initialize to CameraOff
Camera: enter Camera
Camera: enter CameraOff
Power: initialize to PowerOff
Power: enter Power
Power: enter PowerOff
----------------
ENV: send POWER_ON to Power
Power: handle POWER_ON via PowerOff
Power: transition PowerOff to PowerOn
Power: exit PowerOff
Power: enter PowerOn
Power: send CAMERA_ON to Camera
Camera: handle CAMERA_ON via CameraOff
Camera: transition CameraOff to CameraIdle
Camera: exit CameraOff
Camera: enter CameraOn
Camera: enter CameraIdle
----------------
ENV: send CAPTURE to Camera
Camera: handle CAPTURE via CameraIdle
Camera: transition CameraIdle to CameraCapturing
Camera: exit CameraIdle
Camera: enter CameraCapturing
Camera: start Timer 1
start_exposure
----------------
ENV: send POWER_OFF to Power
Power: handle POWER_OFF via PowerOn
Power: transition PowerOn to PowerOff
Power: exit PowerOn
Power: send CAMERA_OFF to Camera
Power: enter PowerOff
Camera: handle CAMERA_OFF via CameraOn
Camera: transition CameraCapturing to CameraOff
Camera: exit CameraCapturing
stop_exposure
Camera: cancel Timer 1
Camera: exit CameraOn
Camera: enter CameraOff
```

(b) Execution trace

Fig. 5 Script 2 and its resulting trace

VII. Future Work and Development Roadmap

Proteus is still a work in progress, though we are well on our way to a usable version 1.0. This section describes a roadmap of how we are planning to enhance in Proteus in the near future, as well as ore advanced features we are contemplating adding in the future.

A. Short-Term Development Goals

As shown in Section VI, Proteus is already working on small examples. Next, we plan to scale up the input HSMs to be more realistic. Specifically, we are planning to take an existing large HSM which was implemented in either ScalaHSM or TextHSM and porting it over to Proteus. We anticipate that this process will require language iteration, as additional Proteus features may be needed to get this larger example working. Once the example is confirmed to work, we want to give actual systems engineers the same exercise. At that point, we will know that Proteus *can* implement the HSM, though this is moot if the target audience cannot perform the same implementation themselves. If the systems engineers struggle (and we anticipate they will), this will entail further Proteus language development to make it easier to use. We consider the porting of a realistic HSM to Proteus by systems engineers to be a major developmental milestone.

B. Long-Term Planned Features

Beyond these short-term goals, we have a number of features which we are planning to integrate into Proteus. We anticipate that some of these will be practically necessary to achieve our short-term goals, whereas others extend Proteus in non-trivial ways. We divide this section by the particular planned feature, and order them based on how soon we are planning to integrate them.

1. Foreign Function Interface

Proteus currently has no built-in way to interact with any other language. While this keeps it simple to reason about what Proteus code does, this is impractically restrictive. For example, most hardware Proteus interacts with will require some form of communication with the operating system, practically necessitating a foreign function interface (FFI) for calling directly into C++. This is an engineering challenge, as we want to give systems engineers flexibility to call unladen C++, but we also want to preserve Proteus' safety properties.

2. User-Defined Data Types

While Proteus allows for the definition of custom events, events themselves are not user-defined data. That is, events cannot contain other events, and events cannot be assigned to variables. While this restriction has greatly simplified early development, it is very restrictive. Notably, most common data structures like linked lists cannot be implemented in Proteus without user-defined data types. As such, in future versions of Proteus, we want to add user-defined data types.

Adding user-defined data types will have to be done very carefully, in order to strike a balance between utility, safety, and performance. Without heap allocation, many kinds of common data structures (e.g., linked lists) are technically possible though arguably not very useful; without the heap, these cannot expand and contract arbitrarily during execution, for example. If later versions of Proteus selectively allow for heap allocation, this will also bring up the question of when memory is deallocated. Garbage collection is commonly used for this purpose, but its unpredictable pauses make it untenable for real-time space applications. Ownership types (as seen in Rust [8]) in contrast are very predictable and overall friendly for embedded applications, though they add such a significant burden on the programmer that they may be considered impractical for use by systems engineers. In contrast to these two approaches, reference counting (as used in Swift [12]) carries relatively minimal programmer burden and is usually far more predictable than garbage collection. However, reference counting still has a negative performance impact, and it is easy to accidentally leak memory when cyclic data structures are used, so this too may be inappropriate.

3. A Module System

Proteus currently requires that all code be contained within a single input file. While this is acceptable for small programs, this does not scale well to larger, more realistic HSMs, particularly if users define their own data types. A module system (e.g., Java's packages or C++'s namespaces) would allow code to be divided into more manageable, logically connected units, spread across a multitude of files. We do not anticipate that adding a module system will be particularly complex, though at the moment there has not been a demand for it.

4. Typeclasses

Nearly all modern languages support *ad-hoc polymorphism*, that is, the ability to define a single interface which can execute different behaviors depending on the specific implementation used at runtime. This is perhaps most commonly seen with inheritance and virtual dispatch in class-based object-oriented languages, but it is also seen in functional languages with higher-order functions. As Proteus grows, particularly after user-defined data types are implemented, we foresee this becoming an advantageous feature to have. To this end, we are planning to add support for *typeclasses* [13], such as those seen in Haskell and Rust, to Proteus. Like object-oriented classes, typeclasses allow for multiple methods to be abstracted behind the same common interface, and called uniformly, yielding different observable behaviors. However, unlike object-oriented classes, this mechanism tends to be simpler to implement, and it usually is more amenable to optimizations. Given the fact that we are targeting systems where performance may be critical, we think the potential performance gains of typeclasses over object-oriented classes makes typeclasses a good fit.

5. More Static Checking

Currently Proteus statically ensures types are compatible, and will also ensure that actors can only receive messages for which they have defined `on` blocks. While this is more static checking than is performed in either ScalaHSM or TextHSM, it is possible to go well beyond this threshold. A major advantage of using HSMs is that they lend themselves to techniques like model checking, static analysis, and theorem proving, and we want to incorporate similar capabilities into Proteus. Specifically, users should be able to define properties which are expected to hold of HSMs, and then somehow see if those properties hold. We have already looked into the use of linear temporal logic for phrasing these properties and performing subsequent analysis (as with DAUT [14]), and are planning to augment Proteus with these sorts of capabilities.

6. Dynamic Fault Detection and Response

While further static checking will prevent more errors, not all errors can practically be prevented at compile time. Software bugs may slip past early phases, and end up in the final product. Moreover, even if software is bug-free, hardware failures are inevitable in harsh space environments, so we cannot simply hope that the system will perform as expected. To this end, we plan to augment Proteus with a runtime verification component (e.g., [15, 16]) which can be used to dynamically detect if a fault (be it in software or hardware) has occurred. From there, recovery actions can be executed to somehow correct the fault. Currently, we foresee programmers defining custom kinds of faults which can occur, mirroring any properties which should statically hold of the system overall (as in Section VII.B.5). Along with each property that *should* hold, programmers can define recovery actions to execute if the system ever detects that a property is violated. Some care must be taken here to ensure that the runtime verification component is computationally inexpensive enough to practically run concurrently with the implementation code (which may be in a hard real-time environment). However, the runtime verification component still needs to preserve accuracy and be expressive enough to be useful, which may conflict with performance.

VIII. Conclusion

In this paper, we have presented Proteus: an HSM-based programming language emphasizing both safety and performance, which is suitable for both simulations and implementations. Proteus is being developed in collaboration with scientists and systems engineers at JPL, who will be the primary users of Proteus. To ensure Proteus will be usable by our target audience, we are developing it iteratively through a series of prototypes which they regularly provide feedback on. We use this feedback to refine Proteus' design, and ultimately make sure we are producing something of value to our audience.

While Proteus is still young in its development, we have shown that it already produces executable programs containing actors, events, and multiple HSMs coordinating with each other. From here, we plan to implement larger HSMs taken directly from actual space applications, and use our development experiences from these to further refine Proteus'. We additionally plan to add more features related to program correctness, in the form additional static and dynamic checks. Ultimately, we want Proteus to be used for HSM-based simulations and control software in space and aeronautics applications, and we are actively working towards that goal.

Acknowledgements Thanks to Simran Gill, Eileen Quiroz, and Frank Serdenia for their contributions to the Proteus compiler. Funded by NASA Minority University Research and Education Project (MUREP) Institutional Research

Opportunity (MIRO) NNH18ZHA008C-MIROG7. The research was carried out in part at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.*, Vol. 8, No. 3, 1987, p. 231–274. https://doi.org/10.1016/0167-6423(87)90035-9, URL https://doi.org/10.1016/0167-6423(87)90035-9.

[2] Havelund, K., and Joshi, R., "Modeling Rover Communication Using Hierarchical State Machines with Scala," *Computer Safety, Reliability, and Security*, edited by S. Tonetta, E. Schoitsch, and F. Bitsch, Springer International Publishing, Cham, 2017, pp. 447–461.

[3] Fowler, M., *Domain-specific languages*, Addison-Wesley, Upper Saddle River, N.J., 2011. URL http://proquest.safaribooksonline.com/640http://proquest.safaribooksonline.com/?fpi=9780132107549.

[4] Havelund, K., and Joshi, R., "Modeling and Monitoring of Hierarchical State Machines in Scala," *Software Engineering for Resilient Systems*, edited by A. Romanovsky and E. A. Troubitsyna, Springer International Publishing, Cham, 2017, pp. 21–36.

[5] Hewitt, C., Bishop, P., and Steiger, R., "A Universal Modular ACTOR Formalism for Artificial Intelligence," *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973, p. 235–245.

[6] Alur, R., and Yannakakis, M., "Model Checking of Hierarchical State Machines," *ACM Trans. Program. Lang. Syst.*, Vol. 23, No. 3, 2001, p. 273–303. https://doi.org/10.1145/503502.503503, URL https://doi.org/10.1145/503502.503503.

[7] Cardoso, R. C., Farrell, M., Luckcuck, M., Ferrando, A., and Fisher, M., "Heterogeneous Verification of an Autonomous Curiosity Rover," *NASA Formal Methods*, edited by R. Lee, S. Jha, and A. Mavridou, Springer International Publishing, Cham, 2020, pp. 353–360.

[8] Matsakis, N. D., and Klock II, F. S., "The Rust Language," *ACM SIGAda Ada Letters*, Vol. 34, ACM, 2014, pp. 103–104.

[9] Charousset, D., Schmidt, T. C., Hiesgen, R., and Wählisch, M., "Native Actors: A Scalable Software Platform for Distributed, Heterogeneous Environments," *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, Association for Computing Machinery, New York, NY, USA, 2013, p. 87–96. https://doi.org/10.1145/2541329.2541336, URL https://doi.org/10.1145/2541329.2541336.

[10] Baidakou, I., "Rotor Actor Framework," ???? URL https://github.com/basiliscos/cpp-rotor.

[11] "libagents: A Multi-threaded C++11 Implementation of the Actor Model," ???? URL https://sourceforge.net/projects/libagents/.

[12] "The Swift Programming Language," ???? URL https://developer.apple.com/swift/.

[13] Wadler, P., and Blott, S., "How to Make Ad-Hoc Polymorphism Less Ad Hoc," *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Association for Computing Machinery, New York, NY, USA, 1989, p. 60–76. https://doi.org/10.1145/75277.75283, URL https://doi.org/10.1145/75277.75283.

[14] Havelund, K., "Data Automata in Scala," *2014 Theoretical Aspects of Software Engineering Conference*, 2014, pp. 1–9.

[15] Havelund, K., and Peled, D., "Efficient Runtime Verification of First-Order Temporal Properties," *Model Checking Software*, edited by M. d. M. Gallardo and P. Merino, Springer International Publishing, Cham, 2018, pp. 26–47.

[16] Havelund, K., "Rule-Based Runtime Verification Revisited," *Int. J. Softw. Tools Technol. Transf.*, Vol. 17, No. 2, 2015, p. 143–170. https://doi.org/10.1007/s10009-014-0309-2, URL https://doi.org/10.1007/s10009-014-0309-2.