Towards a Systems Programming Language Designed for Hierarchical State Machines

Brian McClelland[1], Daniel Tellier[1], Meyer Millman[1], Kate Beatrix Go[1], Alice Balayan[1], Michael J Munje[1], Kyle Dewey[1], Nhut Ho[1], Klaus Havelund[2], and Michel Ingham[2]

[1]California State University, Northridge, Northridge, CA, 91330
[2]Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

***Abstract*—In flight applications, Hierarchical State Machines (HSMs) are often used for writing simulation and control software, including that of the Curiosity rover. At the Jet Propulsion Laboratory (JPL), multiple domain-specific 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 approachable for JPL software engineers. Proteus itself compiles to C++, allowing it to fit easily into existing development toolchains, making it amenable to embedded real-time systems. To ensure that Proteus will be of use to its target audience, it is being iteratively developed through a series of prototypes which are regularly evaluated by key JPL stakeholders, ensuring Proteus always stays on track. 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 a relevant execution trace. In the future, we plan to apply Proteus to larger HSMs taken from real flight 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 flight, including the Curiosity rover's control software [2]. In particular, HSMs are used for the simulation of software models to be implemented, as well as the actual implementation of flight software, including 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 in collaboration with 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, including one embedded within the Scala general purpose programming language [4], and another wherein programmers mix fragments of DSL and C code (strictly used internally at JPL). 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 flight applications. While the C-based DSL is suitable for both simulations and flight software implementation, it offers essentially no safety guarantees, adding unnecessary risk to the development of mission-critical software. Overall, we argue that existing HSM-based languages are inappropriate for the domains they target. In contrast, with Proteus, we are developing a language which is suitable for both simulation and onboard embedded implementation, without safety compromises.

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 [5] support, allowing for the definition of large systems composed of multiple asynchronously-communicating HSMs. 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 unrepresentable in Proteus. This safety is granted by Proteus' fundamental design, without expensive runtime features (e.g., garbage collection) which would preclude real-time embedded environments.

Key to Proteus' design is that it is intended to look and behave similarly to C/C++, and it even compiles to C++. Much existing development at JPL is already performed in C/C++, so going for C++'s look and feel helps ensure adoptability. By compiling to C++, Proteus is amenable to real-time systems, and it can integrate with existing C++ development toolchains. However, unlike with hand-written C++, Proteus code is guaranteed memory safe, and has a plethora of HSM-related features not easily representable directly in C++.

The design and development of a novel programming lan-

guage 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 emergent behavior. On the one hand, Proteus is very experimental, and acts as a proving ground for new ideas. On the other hand, we need to have high confidence that Proteus is intuitive, or else it is unlikely to be used. A very real concern is that Proteus 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.

To ensure language development is moving in a positive direction, we plan 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 easily expressed, tedious 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. By showing potential users a prototype, this should enable specific, actionable feedback.

Overall, the contributions of this paper are as follows:

1) A discussion of Proteus' core features which enable HSM-based development (Section III)
2) A discussion of the iterative process through which we are designing Proteus (Section IV)
3) The application of Proteus to executing multiple small HSMs, demonstrating basic viability (Section V)

II. Background and Related Work

This section covers background information necessary to understand Proteus' features, starting with actors. We also cover related work, particularly in HSM-based languages.

A. Actors

The actor model enables parallel programming by splitting computation into different independent components called *actors* [5]. Each actor executes its own code sequentially, but multiple actors can execute in parallel with respect to each other. In addition to maintaining its own executable code, each actor also maintains internal state upon which this code acts. Internal state is only accessible within the same actor; actors cannot directly manipulate each other's state. Actors can communicate only by sending *messages* to each other. Messages can be arbitrarily specific and contain arbitrary data. In practice, actors generally wait for an incoming message, do some computation to respond to the message (possibly sending further messages to other actors), and then repeat indefinitely.

While actors are an independent concept from HSMs, we argue that a built-in parallelism concept like actors is practically necessary in an HSM-based language. For example, the HSM-based Curiosity rover control software has about 150 parallel threads [2], and each is viewable as a separate actor.

B. HSM Background

HSMs have been used for over 30 years [1], and are common in flight applications [2]. HSMs are a variation of finite state machines, wherein states and whole state machines can be nested within other states. Any variables introduced in a parent state are accessible to child states, and child states similarly inherit behavior from parent states (e.g., state transitions), avoiding needless repetition and improving modularity.

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. 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.

HSMs are more restrictive than arbitrary code, but they provide abstractions which are easier for both humans and machines to reason about, including automated reasoning techniques like model checking [6] and theorem proving [7]. In practice, JPL makes heavy use of HSMs in flight software.

C. Related Work

We are aware of two DSLs which were designed for HSM-based development, herein called ScalaHSM [4] and TextHSM. Both were developed internally at JPL. ScalaHSM is an *internal* DSL embedded into the Scala programming language, and is thus technically a Scala library. While ScalaHSM is useful for simulations, it suffers from two major issues. For one, since Scala is a garbage-collected language, ScalaHSM is not appropriate for the implementation of real-time code; the garbage collector can impart sizable delays at possibly critical moments. Additionally, since ScalaHSM is a Scala library, its users must be familiar with Scala. This is unfortunately not the case for most JPL software engineers, who usually have a C/C++ background. Scala is significantly different from C/C++, making it a difficult and time-consuming task for a typical engineer to pick up and use ScalaHSM. ScalaHSM is therefore 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. TextHSM has seen ongoing use at JPL. Like traditional programming languages, TextHSM is compiled to another language, specifically C. The use of C makes it appropriate for real-time tasks, not just simulations. Like Proteus, TextHSM provides HSM-specific features to the user. However, TextHSM is merely a thin wrapper on top of C. TextHSM effectively has "holes" wherein users directly write C code, and TextHSM itself simply copies the contents of these holes into the final 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.

Hobbs et al. [8] proposes a holistic solution for the design, implementation, and verification of collision avoidance systems. While more specialized than Proteus, this use of a holistic system to replace multiple ad-hoc processes is in the same spirit as Proteus' approach to flight software.

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 features enabling HSM-based development. We introduce these features via example HSMs implemented in Proteus.

A. Examples

Our examples are based on a simplified scenario involving separate, but related, power and camera controls for a space vehicle. The corresponding HSMs are shown in Figure 1. The `Power` HSM operates with two events `POWER_ON` and `POWER_OFF`, which are assumed to come 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 another 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 Figures 2 and 3 implements the HSMs in Figure 1. The rest of this section discusses 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 is outside of this paper's scope. The compilation avoids heap allocation and other C++ features which are problematic for our target embedded environments.

B. Actors

Actors are specified within named `actor` blocks. Actor blocks are only valid at the top level of a program, disallowing nesting. At runtime, each actor runs in a separate thread. Figures 2 and 3 use two actors, for the `Camera` and `Power` HSMs, respectively. 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:

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

All code defined within the `MyActorWithVariables` actor would have 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.

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 our target audience typically uses actors. Each actor usually either contains control code which is expected to constantly run, or serves as an interface to a fixed piece of hardware; neither case needs dynamic actor creation or termination. Moreover, while such dynamic features would make Proteus more flexible, they would also make Proteus code more difficult to reason about, which is contrary to our design goals.

C. Events: Definition, Sending, and Receiving

True to the actor model [5], Proteus actors can only communicate with each other via exchanging messages (events). To send an event, the user must first define what events are valid, along with the types of any contained data, 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 2, including `POWER_ON`, `POWER_OFF`, and `CAPTURE`. The `CAPTURE` event holds a single integer, and all other events contain no data. Events can also be defined holding multiple, mixed data types. Events can be sent to an actor using `!`, as is done in Figure 3 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, which would be listed within the curly braces. The number and types of data provided must agree with the declaration, or else a Proteus compile-time error results.

To receive an event, an actor must use the `on` reserved word. `on` is used extensively in Figures 2 and 3, 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 2 with:

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

The data is bound to the newly-declared local variable `t`, which is implicitly known to be of type `int` from the

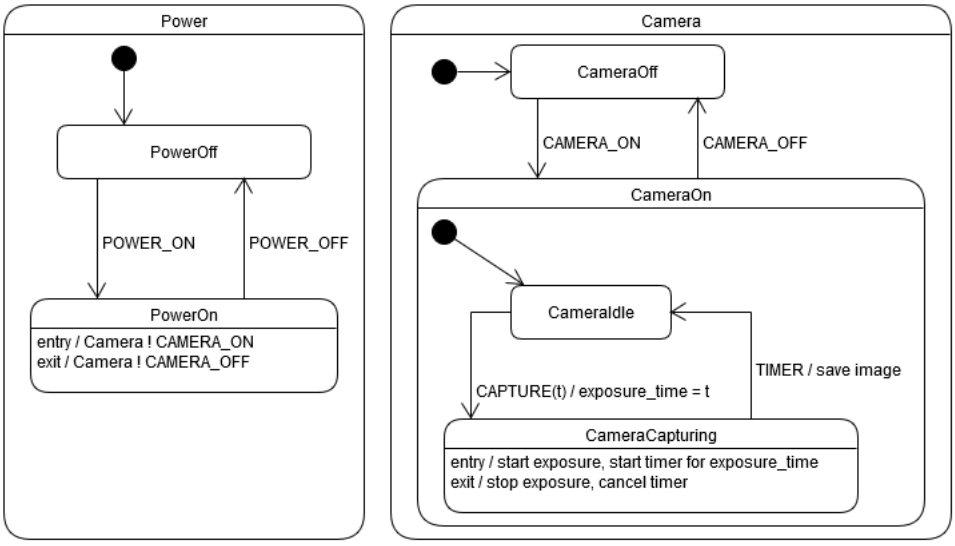

Fig. 1: Example HSMs for managing power and camera control.

`CAPTURE` event declaration. When an event is received, the code within the corresponding `on` block is executed.

Semantically, actors are fundamentally event-driven; they wait to receive an input event, execute the corresponding event handler, and repeat this process indefinitely. An actor will fully process one input event before attempting to process another one. With this in mind, in Figure 3, if `Power` received multiple `POWER_ON` events simultaneously, only one would be processed immediately. All others would be stored in an internal event queue hidden inside `Power`. Only once the first `POWER_ON` event was fully processed would the next `POWER_ON` event be removed from the queue for processing. Event queues have unbounded size for testing (requiring dynamic memory allocation), though these can be changed to have statically-known size with minimal modification.

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

```
actor MultiEventExample {
  on FOO {
    // executed if FOO received
  }
  on BAR(a, b) {
    // executed if BAR 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 separate `BAR` event is later 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 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.

D. HSM and HSM State Definition

Proteus has language-level support for HSMs. As Figures 2 and 3 show, state machines and states are declared with `statemachine` and `state`, respectively. The `statemachine` reserved word 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 `initial`, as Figure 2 does with `initial CameraOff;`.

Actors containing state machines execute them sequentially, one state at a time. That said, since states can be nested in other states, execution may be present in multiple states simultaneously. We refer to the most deeply-nested state currently being executed in an actor 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.

Variables can be defined in states, known as *state variables*. These variables are accessible from within the declared state and also in any child states of that state. The `CameraOn` state in Figure 2 shows this with the `exposure_time` state variable. States can contain `on` blocks, which have more complex semantics when nested states are involved. For example, consider the `CameraCapturing` state's `on` block in Figure 2, 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 a `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

```
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();
          }
        }
      }
    }
  }
}
```

Fig. 2: Proteus implementation of events and `Camera` HSM from Figure 1.

`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 Figures 2 and 3, 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.

```
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");
}
```

Fig. 3: Proteus implementation of `Power` HSM from Figure 1, along with relevant helper functions.

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.

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 Figures 2 and 3. `go` specifies the state to transition to, which must be contained within the same state machine. Additionally, `go` takes an optional block of code to execute just before making the state transition.

While not shown in Figures 2 and 3, there is also a conditional version of `go` called `goif`, which will only transition to a state if a provided Boolean condition is also true. As with a typical `if` statement, `goif` can be chained with `else goif` clauses. Also like `if`, the alternatives to `goif` are tried sequentially, 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 Figures 2 and 3. 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 then performed, starting from the outermost state being entered and moving to the innermost state. Both of these behaviors are per the usual HSM semantics of state transitions.

G. Functions

A number of helper functions are declared at the end of Figure 3, 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 text 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. This notation for return types is inspired by Rust [9], and avoids the need for a pseudo-type like `void`.

Proteus currently lacks support for pointers or arrays, as we have been focusing on the HSM-related components of Proteus. We plan to include these features in later versions of Proteus, though in a manner which does not compromise on safety (e.g., C++-style references, checked array access, etc.).

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 as the method. Similarly, methods are only available to be called from within the actor or enclosing state.

IV. 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 Proteus, ultimately leading to a failure to be adopted and subsequent abandonment. To mitigate these risks, we are employing an iterative design and implementation process, and JPL software engineers are involved in this process.

Proteus is being developed as a series of prototypes. Upon a prototype's completion, we solicit feedback from key JPL stakeholders, and use it to guide the development of the next prototype. By regularly involving the target audience, we ensure that development never deviates far from expectations, even if expectations are initially unclear.

So far, these iterations have been small and informal, and have only directly involved co-authoring JPL stakeholders. However, these interactions have already led to key language improvements. For example, in a prior prototype, HSMs, actors, and states were all conflated into the same construct which had a complex, 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 from Section III.

We have found that these prototypes do not need to be complete to be useful. Even if a feature is not yet fully implemented, as long as the target audience understands what the feature *should* do, useful feedback can be solicited. As such, pivoting can be done early in prototype development.

V. Execution

This section demonstrates that Proteus can compile and run programs. We implement the example from Figures 2 and 3 in Proteus, and produce an execution trace showing what the resulting program does. We first discuss this capability to produce an execution trace.

A. Execution Tracing

The Proteus compiler can be invoked from the command line with a flag which strategically adds print statements to the output C++ program. These print statements produce a human-readable trace of the program's execution, noting major HSM-related actions. For example, consider the following trace line:

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

This states that from within the `Power` actor, a `POWER_ON` event is sent to the `Camera` actor. The name at the start of each line is the thread on which the action is taking place. The thread is typically an actor, but can also be a timer. The various possible trace actions are:

- `init 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 until no further state nesting is present. This line will be followed by a sequence of `enter` actions as the transition happens.

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

Fig. 4: ENV code.

- `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 active state or a (direct or indirect) parent. Anything that happens during event handling will follow this line.
- `trans {State1}->{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 from Figures 2 and 3, without any additional components, would sit indefinitely in an off state. This reflects the fact that there is an assumed external generator of `POWER_ON` and `POWER_OFF` events, be they from ground control or some other source. As such, for testing purposes, we added a third `ENV` actor to provide these crucial power-related events. `ENV` sends these events with `wait` statements in between, which temporarily suspend execution of the calling actor for a provided number of milliseconds. `ENV`'s code is enclosed within a single state, and is shown in Figure 4.

Figure 5 shows the trace output of the resulting Proteus program. Separators, indicated with multiple hyphens, have been added for readability, dividing the output into blocks. Each block results from either initialization or an external event. By observing this trace, we can deduce the following:

- Initial states are properly followed.

```
Camera: init to CameraOff
Camera: enter Camera
Camera: enter CameraOff
Power: init to PowerOff
Power: enter Power
Power: enter PowerOff
----------------
ENV: send POWER_ON to Power
Power: handle POWER_ON via PowerOff
Power: trans PowerOff->PowerOn
Power: exit PowerOff
Power: enter PowerOn
Power: send CAMERA_ON to Camera
Camera: handle CAMERA_ON via CameraOff
Camera: trans CameraOff->CameraIdle
Camera: exit CameraOff
Camera: enter CameraOn
Camera: enter CameraIdle
----------------
ENV: send CAPTURE to Camera
Camera: handle CAPTURE via CameraIdle
Camera: trans CameraIdle->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: trans CameraCapturing->CameraIdle
Camera: exit CameraCapturing
stop_exposure
Camera: enter CameraIdle
----------------
ENV: send POWER_OFF to Power
Power: handle POWER_OFF via PowerOn
Power: trans PowerOn->PowerOff
Power: exit PowerOn
Power: send CAMERA_OFF to Camera
Power: enter PowerOff
Camera: handle CAMERA_OFF via CameraOn
Camera: trans CameraIdle->CameraOff
Camera: exit CameraIdle
Camera: exit CameraOn
Camera: enter CameraOff
```

Fig. 5: The resulting trace from Figures 2, 3, and 4.

- 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 the `TIMER` event.
- Events are handled by the correct states and event handling results in the correct transitions.

From these observations, we can conclude that Proteus is executing these HSMs correctly; all of these observations follow directly from what the HSM says to do. The HSM in play is simple enough to trace by hand without Proteus, yet complex enough that it stresses most of Proteus' HSM capabilities. This provides some reasonable assurance that Proteus itself is correct, and we internally have a much larger and more complex test suite to further ensure this.

VI. Conclusions and Future Work

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 of flight software. Proteus is being developed in collaboration with JPL software engineers, who will be Proteus' primary users. To ensure Proteus will be usable by our target audience, we are iteratively developing it in concert with key JPL stakeholders, via a prototype-driven process.

While Proteus is still early in development, we have shown that it can already compile and run programs involving actors, events, and multiple cooperating HSMs. For future work, we plan to port larger HSMs to Proteus, taken directly from flight applications. These porting experiences will further expose the strengths and weaknesses of Proteus, and we will use this information to refine Proteus' design. We are currently adding more features to Proteus, including runtime verification components (e.g., [10], [11], [12]), user-defined data types, and typeclasses [13]. We are also investigating adding visualizations, more static checks, and model checking [6]. Ultimately, we want Proteus to be used for HSM-based simulations and control software in flight 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] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, p. 231–274, Jun. 1987. [Online]. Available: https://doi.org/10.1016/0167-6423(87)90035-9

[2] K. Havelund and R. Joshi, "Modeling rover communication using hierarchical state machines with scala," in *Computer Safety, Reliability, and Security*, S. Tonetta, E. Schoitsch, and F. Bitsch, Eds. Cham: Springer International Publishing, 2017, pp. 447–461.

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

[4] K. Havelund and R. Joshi, "Modeling and monitoring of hierarchical state machines in scala," in *Software Engineering for Resilient Systems*, A. Romanovsky and E. A. Troubitsyna, Eds. Cham: Springer International Publishing, 2017, pp. 21–36.

[5] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, p. 235–245.

[6] R. Alur and M. Yannakakis, "Model checking of hierarchical state machines," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 3, p. 273–303, May 2001. [Online]. Available: https://doi.org/10.1145/503502.503503

[7] R. C. Cardoso, M. Farrell, M. Luckcuck, A. Ferrando, and M. Fisher, "Heterogeneous verification of an autonomous curiosity rover," in *NASA Formal Methods*, R. Lee, S. Jha, and A. Mavridou, Eds. Cham: Springer International Publishing, 2020, pp. 353–360.

[8] K. L. Hobbs, J. Davis, L. Wagner, and E. Feron, "Formal specification and analysis of spacecraft collision avoidance run time assurance requirements," in *2021 IEEE Aerospace Conference (50100)*, 2021, pp. 1–16.

[9] N. D. Matsakis and F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, vol. 34, no. 3. ACM, 2014, pp. 103–104.

[10] K. Havelund and D. Peled, "Efficient runtime verification of first-order temporal properties," in *Model Checking Software*, M. d. M. Gallardo and P. Merino, Eds. Cham: Springer International Publishing, 2018, pp. 26–47.

[11] K. Havelund, "Rule-based runtime verification revisited," *Int. J. Softw. Tools Technol. Transf.*, vol. 17, no. 2, p. 143–170, Apr. 2015. [Online]. Available: https://doi.org/10.1007/s10009-014-0309-2

[12] K. Havelund, "Data automata in scala," in *2014 Theoretical Aspects of Software Engineering Conference*, 2014, pp. 1–9.

[13] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 60–76. [Online]. Available: https://doi.org/10.1145/75277.75283