Hierarchical State Machines

Background on the creation of Proteus

- Proteus was created as a solution as the existing programming languages that were being used at Jet Propulsion Laboratory (JPL) as they had significant issues with one or more of usability, performance, safety, making them problematic for HSM-based development.
- 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++, which made it easier for JPL software engineers to fit easily into existing development toolchains, making it amenable to embedded real-time systems.
- Its basic viability can be shown by an example of utilizing multiple independent HSMs communicating with each other, and a relevant execution trace. Then, in the future they plan to apply Proteus to larger HSMS taken from real flight applications.

So what are Hierarchical State Machines (HSM) and how are they used?

- They are finite state machines whose states themselves can be other state machines. Hierarchy is a useful con- struct in many modeling formalisms and tools for software design, requirements and testing. We summarize recent work on hierarchical state machines with or without concurrency.
- They are used to design, implement, and reason about complex software systems to be deployed in flight, including the Curiosity rover's control software.
- Used for the simulation of software models to be implemented, as well as the actual implementation of flight software, including control systems.

Why the need for a new programming language (Proteus) ?

- In order to use HSMs to scale large systems, there is a need for special tooling and programming language support, where Proteus is presented.
- There are multiple preexisting HSM-based language design efforts, they are observed to have significant weaknesses.
- Their typical development approach involves the use of graphical modeling tools to draw HSMs and then automatically generate code from their internal representation. While the visualization is recognized as essential, this approach often results in opaque code which bears little resemblance to the visualization. This makes the output code difficult to inspect and reason about.
- Two textual HSM Domain Specific Languages (DSL) have already been developed at JPL, including one embedded within the Scala general purpose programming language, and another wherein programmers mix fragments of Domain specific Languages (DSL) and C code. Although, they argue that neither of these DSLs are appropriate for their purpose.

- C-based DSL is suitable for both simulations and flight software implementations, but it offers no safety guarantees, adding unnecessary risk to the development of mission-critical software.
- They argue that the existing HSM-based languages are inappropriate for the domains they target. Where Proteus comes along, trying to develop a language that is suitable for both simulation and onboard embedded implementation, without safety compromises
- Proteus offers built-in support for representing state machines, states, external events, and state transitions.
- Proteus also has integrated actor support, allowing for the definition of large systems composed of multiple asynchronously- communicating HSMs. Also, unlike the existing HSM-based DSLs, Proteus is designed from the ground up with both safety and performance in mind. Proteus programs are memory safe by construction, and are devoid of undefined behavior.
- Many common program bugs native to C/C++ are unrepresentable in Proteus This safety is granted by Proteus' fundamental design, without expensive runtime features which would preclude real-time embedded environments.
- The key of Proteus design is that it is intended to look and behave similarly to C/C++, and it compiles to C++. Much of the existing development at JPL is already performed in C/C++, so going for C++'s look and feel helps ensure adoption of Proteus.
- Also by Proteus compiling to C++, it is controllable to real-time systems, and it can integrate with existing C++ development toolchains.
- Proteus code is guaranteed memory safe, and has an abundance of HSM-related features not easily representable directly in C++.

What is being done to make sure that Proteus is working Properly with HSMs?

- To ensure the language development is moving in a positive direction, they plan to gather a series of moderately-sized HSM implementations which were previously developed at JPL.
- Proteus is very experimental, and acts as a proving ground for new ideas. They need to have high confidence that Proteus is intuitive, or else it is unlikely to be used. A major 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 until they see it in front of them.
- Ensuring that the development of the language is moving in a positive direction, they plan to gather a series of moderately-sized HSM implementations which were developed at JPL.
- They will test each Proteus Prototype with each HSM that was developed at JPL and if their implementation proves difficult, they will iterate on Proteus' design to simplify re-implementation.

- Once they are satisfied with this, they will show the re-implementation to the stakeholders at JPl and get feedback from them.
- The feedback they could possibly get from the prototype would be the core features which enable HSm-based development, discussion of the iterative process through which we are designing Proteus, and the executing of small HSMs, demonstrating basic viability.

What is necessary to understand Proteus features (starting with Actors)?

- *Actors*: which is the actor Model enables parallel programming by splitting computation into different independent components.
- Each actor executes its own code sequentially, but multiple actors can execute parallel with respect to each other. In addition, each actor also maintains the internal state upon which this code acts. In which 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 arbitrary data. In practice actors generally wait for an incoming message, do some computation to respond to the message, and then repeat indefinitely.

Background of HSM

- HSMs have been used for over 30 years, and are common in flight applications.
- HSMs are a variation of finite state machines, wherein state 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, which helps avoid repetition and improves modularity.
- HSMs transition between states in response to input events. From the view of the actor model, events are indistinguishable from messages, and Proteus exclusively uses the word "even" to refer to messages. So when a Proteus actor receives an event, it can trigger a state transition, along with the execution of user-defined code.
- HSM 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 and theorem proving.

Related work:

- There are two Domain specific languages (DSL) which were designed for HSM-based development, which are ScalaHSM and TextHSM, both developed at JPL.
- ScalaHSM is an internal DSL embedded into the Scala programming, and technically is a Scala library. It suffers from two major problems. First, Scala is a garbage-collector language, which 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 library of Scala, one must be familiar with Scala

- Problem as the JPL software engineers come from a C/C++ background, which Scala is significantly different from C/C++, which makes it time consuming to learn a new language.
- TestHSM is built as an external DSL, meaning that its syntax is separate from any other programming language. TextHSM is compiled to another language, specifically C. The use of C makes it appropriate for real-time tasks, not just simulations.
- TextHSM provides HSM-specific features to the user, but TextHSM is just a thin wrapper on top of C. Which also has its own problems; where users write directly C code, and TextHSM itself simply copies the contents of these holes into the final product, which doesn't check that the C code is correct or even syntactically valid. For these reasons TextHSM is inherently unsafe and thus risky for any mission-critical development.

What are HSMs?

- State machines can be an incredibly powerful technique, but they require an infrastructure (framework) that at a minimum provides: a run-to-completion (RTC) execution context for each state machine, queuing of events, and event-based timing services. This is really the pivotal point: Without an event-driven infrastructure, such as a Real-Time Embedded Framework, state machines are like cars without an infrastructure of roads and gas stations.
- The relationship between an event-driven framework and state machines is mutually synergistic. On one hand, the framework provides the thread context and event queuing that the state machines need to process the events in a run-to-completion fashion. On the other hand, state machines provide the structure and clear design for the event-driven behavior running inside the framework. State machines are also the most constructive part of the design amenable to modeling and automatic code generation.