Proteus Notes:

Proteus is a prototype Programming language and compiler that is being developed as a project for CSUN autonomy Research Center for STEAM in collaboration with NASA Jet propulsion Laboratory (JPL).

Goals for Proteus:

- Become safer Language for System Engineers to use
- Usable for Autonomous Systems

How Proteus works:

Monitor verifies traces are satisfied

- If these traces were to be satisfied certain properties will make some judgment call the program

Runs on models certain to Actors and Hierarchical state machines to name a few.

RV = Runtime Verification = Method for analyzing program behavior using a log generated during execution.

- Important to have fully functioning and correct programs which is a must before proceeding with RV

Because of RV it makes it useful to implement state machines

Tech Being Used:

Proteus Compiler written in C++ and its output C++ programs (17 standard)

Software used for third party is CMake for building the project, Icov for generating coverage reports, and Cach2 as a unit testing framework, and Cxx to parse command line options

Actors:

- Model for concurrent systems in which each component is an independent entity; (actor).
- Actors share no resources
- Carry out their actions concurrently
- Only affect each other indirectly by sending messages to each other
    - When handling msgs
        - They could become modified
        - Send more messages

        - Or create more actors
- Actor model is advantageous for concurrent programming
- It's safer
- Bugs are impossible as actors do not share resources
- Handle events by defining event handlers → these also perform state transitions
- Actors are given an HSM with a state machine keyword
- **Actors cannot be destroyed or created and exist for duration of the program (same for states)**
- **Cannot access data the data of its super states and actors**

Hierarchical State Machines:

Hierarchical State Machines (HSMs):

- HSMs are an extension of the formalism of state machines.
- They are designed to model complex systems effectively.

Challenges with State Machines:

- Standard state machines are useful for describing dynamic behavior.
- However, as complexity increases, the number of states and transitions also increases.
- This can make standard state machines challenging to use for real-world systems.

Addressing Complexity with HSMs:

- HSMs address the complexity issue by allowing states to be HSMs themselves.
- Sub-states inherit the transitions of their parent states.
- This hierarchy enables key abstractions for reducing complexity.

Benefits of HSMs:

HSMs provide several benefits, including:

- Reuse of behavior: Sub-states can inherit and reuse the behavior of their parent states.
- Detail hiding: Complex details can be hidden within sub-states, simplifying the overall model.
- Modularity: HSMs promote a modular approach, making it easier to manage and understand complex systems.

Transitions: can have guard conditions == "[ ]"

- Prevent a transition form being taken unless the condition evaluates as true
- Marked with go or goif (for guarded transitions)

(back) Slashes "/" == mark actions

- Could be performed upon entry to a state
- Exit from a state
- During a transition

Entry/exit actions are performed for each state nested order as transitions are taken

Proteus:

- Based upon
    - actors (may have zero's or 1 HSM) and HSM (each belongs to 1 actor)
    - Definitions of actors
    - Events
        - Sent with <actor> ! <event> syntax

Static Construction:

- A state can always safely access the data of its super states and actor

RunTime Verification:

- RV is a dynamic analysis method used to check the correctness of a program during its execution.

Method of Operation:

- During RV, the program being analyzed is instrumented, meaning it's modified to produce a sequence of events or a trace.

Monitoring Trace:

- A monitor is used to analyze the trace generated by the instrumented program.
- The monitor's role is to verify whether the trace satisfies specified properties.

Response to Non-Conformance:

- If the trace does not meet the specified properties, the monitor can take actions like: Raising an alarm. Attempting to correct the fault or issue.

Online and Offline RV:

- RV can be performed in real-time during program execution.
- Alternatively, it can be done offline by analyzing trace logs after the program's execution has completed.

Variability in Property Expression:

- Properties to be verified can be expressed in different ways, including:
- Formal specification languages based on linear temporal logic.
- General-purpose programming languages, allowing for flexibility in expressing and checking properties.

Proteus Compiler's Initial State:

- At the project's outset, the Proteus compiler had limitations.
- It could generate C++ code from Proteus programs, but the resulting C++ code didn't produce working executable programs.
- The compiled programs were non-functional, not executing as intended.

Incomplete Code Generation:

- Proteus code was successfully translated into C++ code, but the translation was incomplete.
- The generated C++ code lacked essential components necessary to tie the program together.

Validity vs. Correctness:

- The compiled C++ code did not produce errors during compilation, which meant it was valid C++ code.
- However, the actual correctness or functionality of the code was untested beyond narrow unit tests.
- This lack of correctness testing posed a significant challenge for implementing runtime verification (RV).

First Phase of the Project:

- The first phase of the project was dedicated to completing code generation and ensuring its correctness.
- This involved implementing code generation for actors, states, and events.
- A runtime support library was created to enhance the functionality of the generated code.
- A system for testing whole programs was developed as part of this phase.

Complexity of the Work:

- The passage acknowledges that the code generation and runtime library tasks were substantial and involved a significant amount of work.
- These tasks were not directly related to runtime verification but were essential for the project's success.

Actor Runtime Core:

- The actor runtime is described as the core of a compiled Proteus program.
- It's implemented as a class within the runtime library.

Basic Actor Model and State Machine:

- The actor runtime class implements a fundamental actor model and state machine.

- This model operates independently on its own thread.

Key Functions and Features:

- The actor runtime maintains several important features:
- Event Queue: It manages an event queue to store incoming events.
- Current State: It holds a reference to the current state of the actor.
- Event Processing: Events sent to the actor are queued until they can be processed by the event loop.
- Event Handling: When an event is dequeued, it is passed to the current state for handling.
- State Transition: The current state handles the event and returns the new current state.

Template Class Parameters:

- The actor runtime is a template class that takes two parameters:
- State Type (S): Specific to each compiled Proteus program, it must have methods S* init() and S* next(E*).
- Event Type (E): There are no specific interface requirements for this type.

Multiple Actor Runtimes:

- In a Proteus program, each Proteus actor has its own actor runtime.
- All computation within a Proteus program occurs within the event loops of these actor runtimes.

EXAMPLE :

```
template <typename S, typename E>
class Actor {
        EventQueue<E> _queue;
        S* _state = nullptr;
        …
        public:
                void initialize(S* initial_state);
                void send(E const& e);
                void terminate();
};
```

Figure 3: The actor runtime's interface, queue, and state

Proteus Events in C++:

- Each Proteus event is represented as a C++ struct.
- The C++ struct mirrors the Proteus event's parameters.

- Since Proteus event parameters are unnamed, the C++ struct parameters are given numbered names.

Prefixing Proteus Names:

- To avoid naming collisions with C++ keywords or other names, Proteus names are prefixed when translated to C++.

Translation Example:

- The passage provides an example of the translation from Proteus to C++.
- For instance, a Proteus event E with parameters int and int is translated to a C++ struct called prot_event_E with parameters prot_type_int _0 and prot_type_int _1.

Event Union for Storing Different Event Types:

- To enable storage of different event types in a single queue, each Proteus program generates an "event union."
- The event union is designed as a single type capable of holding any event type.

Tagged Union:

- The event union is a tagged union, meaning it keeps track of the active union member.
- For each event in the program, the event union is provided with a constructor that sets the correct tag and union member.

Data Type Simplicity:

- All Proteus data types are noted to be "trivial" with no constructors, destructors, etc.
- As a result, the event union remains sound without needing to define a destructor or assignment operators.

Proteus
event E {int,int};
—----------
C++
```
struct prot_event_E {
        prot_type_int _0;
        prot_type_int _1;
};
```
Figure 4: C++ translation of events
Left off : Page 10