Introduction and Overview

This paper describes construction and execution of a Hierarchical State Machine (HSM). Eight sub-components – objects used to build the machine – are described first. These eight components are the building blocks available to the programmer and are arranged in a static geometry *prior to execution.* The dynamic behavior of the HSM – how it acts *during execution* – is presented second, including two techniques to introduce multi-threaded execution.

Keywords: state machine, state machinery, hierarchy of hierarchy, two level hierarchy, dynamic trees, stack-like tree, constrained tree, linked state machines, multi-threaded state machine.

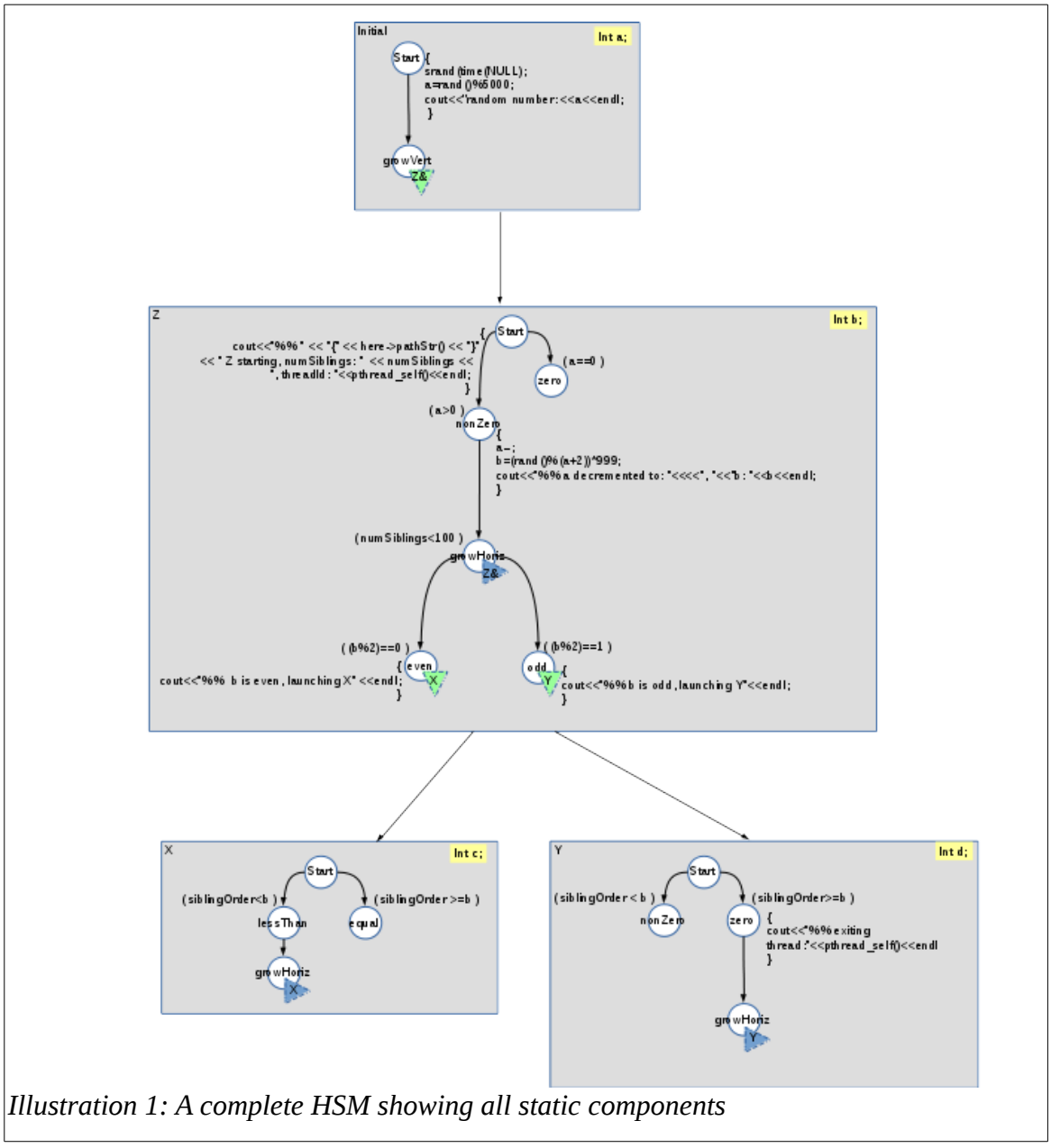

Illustration 1: A complete HSM showing all static components

Static Components

An example fully-constructed machine is shown in Figure 1. This example machine selects a random integer "a" between 0 and 5000; if a is even, the machine calculates and prints a! (a Factorial). If a is odd, the machine prints a Fibonacci set of integers with size=a.

Several initial observations can be made from the example which apply to any HSM:

1. The overall HSM contains several smaller named sub-machines arranged in a hierarchy. In Figure 1 the sub-machines are shown as grey rectangles and are named "Initial", "Z", "X", "and "Y".
2. Each sub-machine is composed of several states, also arranged in a hierarchy. In Figure 1 the states are shown as white circles.
3. The overall HSM contains exactly one sub-machine named "Initial" at the top of its hierarchy of sub-machines.
4. Each sub-machine contains exactly one state named "Start" at the top of its hierarchy of states.

The programmer solving non-trivial problems may construct an HSM with an unlimited number of sub-machines and an unlimited number of states. These four basic characteristics will be retained regardless of the complexity of the HSM.

1. Sub-machine

The sub-machine is the highest-level component and represents a basic unit of work - similar to a function in a traditional language. Sub-machines are used to decompose a large problem into several smaller, more manageable units of logical work. The "Initial" sub-machine in every HSM is created automatically and is always named "Initial"; the programmer assigns a unique name to each additional sub-machine created below it in the hierarchy. The programmer may insert as many sub-machines as desired into whatever hierarchy is necessary to solve the problem at hand.

2. Variable

Variables are declared by the programmer within each sub-machine. In the illustration, the variable declarations are in yellow rectangles. Variable names are chosen by the programmer and must be unique within each sub-machine. Allocation of storage for those variables does not occur until the sub-machine is instantiated during execution.

Several predefined variables are also available to the programmer, these are described below. In Figure 1, the predefined variables numSiblings and siblingOrder are used.

3-4. States and edges

Each sub-machine includes one or more states, also arranged into a hierarchy via edges. In the example, the states are white circles, and the edges are directional arrows interconnecting the states. During execution, whenever a sub-machine is instantiated, a *current-state* pointer is initialized to the top-most or Start state*;* the current-state pointer advances downwards by traversing the edges through various *intermediate-states*, until finally reaching one of the leaf or *final-states.* State transitions may proceed only along the prescribed edges. The hierarchy of states and edges within each sub-machine are also arranged by the programmer according to the problem at hand.

Whenever an instantiated sub-machine reaches a final-state, execution of that instance is complete -- no further changes to the current-state are possible, and that instance of the sub-machine becomes dormant.

The “Start” state in every sub-machine is created automatically and is always named “Start”; the programmer assigns a unique name to each additional state created below the Start state. The programmer may insert as many states as desired into whatever hierarchy is necessary to solve the problem at hand.

One important characteristic of HSM's becomes clear here – loops are not allowed in the states of a sub-machine. This is the definition of a hierarchy – it cannot contain loops – and is an important restriction for the HSM programmer. Repetition is accommodated via other means described below.

Note the variety of different shaped state hierarchies in Figure 1. There is no relationship between the shape of the “outer hierarchy” of sub-machines and the “inner hierarchy” of states, and there is no relationship between the various “inner hierarchies” in each sub-machine.

5. Predicates

Predicates are Boolean expressions associated with the edges. In the illustration, the predicates are enclosed in parenthesis. The Predicates are specified by the programmer and may refer to the variables declared within and above the sub-machine in which they appear; predicates may also refer to predefined variables or to constants. Predicates control transitions into the next-states during sub-machine execution. The predicates of all potential next-states are evaluated, and if any one of these predicates evaluates true the current-state will transition into the corresponding next-state. If multiple predicates evaluate true at the same time, the current-state may transition into any one of those multiple states (the order of evaluation is implementation specific).

If the programmer does not assign a predicate to a state, the default predicate (true) is used. Transitions into states without a predicate may proceed immediately.

The programmer cannot apply a predicate to the Start state of any sub-machine -- this state has no inbound edge and is always entered automatically when the sub-machine is instantiated.

6. Code fragments

Code fragments are optional executable code associated with the states. In the illustration, the code fragments are enclosed in curly braces. These code fragments are executed upon successful transition into the corresponding state. Code fragments define *what* work the HSM will actually perform during execution (the predicates above control *when that* work is performed). Code fragments are specified by the programmer and may refer to any variable declared within and above the sub-machine in which they appear. Code fragments are typically called synchronously during machine execution – further transitions are blocked until the current-state code fragment completes. For this reason, code fragments are typically just a few instructions, and typically don't include long-running loops or lengthy waits. It is possible, however, to declare a code fragment as “asynchronous” by use of trailing ampersand notation { ... }&. Asynchronous code fragments execute in a new separate thread; further transitions from the current-state may proceed

immediately once the new thread is launched, without waiting for the new thread to exit. As with any multi-threading, it is the programmer's responsibility to ensure data consistency and avoid race conditions.

7. Forward Links

Forward links are optional tags associated with states. In Exmaple 1 the forward links are green and blue triangles. Forward links refer to target sub-machines by name and, when encountered during execution, instantiate the target sub-machine. *Forward* in this context means “forward in time”; these links specify the temporal relationships between execution of the various sub-machines. Forward links can be recursive -- they can target the same sub-machine in which they appear.

Two types of forward links are available to the programmer -- “vertical” (in the illustration, green triangles pointing down) and “horizontal” (blue triangles pointing to the right). The difference between the two is described below.

Forward links allow the programmer to control the logical flow of the overall machine during execution by controlling when the sub-machines are instantiated. Recursive forward links allow the programmer to introduce repetition.

When an executing sub-machine traverses a state containing a forward link, a new sub-machine is instantiated for the target sub-machine. The referring sub-machine continues execution from its current-state without waiting for the target in any way; the newly instantiated target sub-machine begins execution at its Start state.

If a forward link occurs in a non-Final state, both the referring and referred-to sub-machines will execute simultaneously, each with its own current-state and each with it's own instantiated variables.

8. Threading Primitives

The HSM programmer also uses the & character to specify asynchronous multi-threaded sub-machine execution. It is possible to declare a forward link as “asynchronous” by use of trailing ampersand notation &. Asynchronous forward links cause execution of the target sub-machine within a new thread. Any additional sub-machines instantiated by that target are executed within the new thread as well. As with any multi-threading, it is the programmer's responsibility to maintain practical limits on the number of threads launched. This can be achived using predicates and the predefined variables described below.

Dynamic behavior during execution

HSM execution may begin after the programmer has arranged all the sub-components.

Due to the concurrency possible between the sub-machines, an HSM will usually require several current-state pointers, one per instantiated sub-machine. A complicated machine with hundreds of concurrent sub-machines will require hundreds of current-state pointers. Another tree structure – a dynamic, execution-time tree structure – is used to store the current-state pointers and instantiated variables declared by the sub-machines. One current-state pointer for one sub-machine instance is stored in each node of the dynamic tree. This dynamic tree is analogous to a run-time stack – except it is a tree, not a stack. Like stacks, there are certain restrictions on how the tree grows and contracts.

The dynamic tree contains exactly one node at the start of execution – pointing to the Start state in the Initial sub-machine. As execution proceeds, this dynamic tree grows as additional sub-machines are instantiated and contracts as the instantiated sub-machines become dormant. Eventually, once all instantiated sub-machines become dormant, execution of the overall HSM is complete.

In Figure 1, one variable "a" would be instantiated in the root node of the dynamic tree, along with a current-state pointer initialized to the Initial state of the Initial sub-machine. As the initial sub-machine progresses down through its states, it may encounter one or more forward links to other target sub-machines, causing insertion of a new dynamic node into the dynamic tree for the target -- the new node includes a new current-state pointer to the target's Initial state, and new instances of all variables declared in the target. This process of transitioning and forward-linking continues recursively until all instantiated sub-machines reach a final state. Once this occurs, execution of the HSM is complete.

Each instance of an executing sub-machine is associated with exactly one node in the dynamic tree. When a vertical forward link is encountered during execution, a new node is inserted as a child of the current node (the tree grows "wider"*).* When a *horizontal* forward link is encountered, a new node is inserted as a sibling of the current node (the tree grows "deeper"*).*

Nodes in the dynamic tree can be physically deleted from the run-time tree only after two conditions are met: 1) the sub-machine instantiated at that node reaches a Final state, and 2) all sub-machines instantiated below it have also reached a Final state. This ensures that deletion of nodes in the run-time tree occurs only "at the leafs" of the dynamic tree, and that any instantiated variables remain instantiated whenever access might occur.

There is one special consideration for the geometry of the Initial sub-machine: since it is not possible for the root node in a tree to have siblings, the initial sub-machine cannot contain any *horizontal* forward links – only *vertical*.

Predefined Variables

The following variables are always available to the programmer:

- numChildren: the number of children of the current sub-machine in the dynamic tree.
- numSiblings: the number of siblings of the current sub-machine in the dynamic tree.
- nextChild: the number to be assigned to the next child node.