

UC Berkeley Computer Science

Subject: Ideas About Simulator Structure
Date: January 17, 2006
From: Ivan Sutherland
UCIES #2006-is16

References:

UCIES# 2005-is14: FLEET – A One-Instruction Computer, Ivan Sutherland, 5 December 2005
Reference to Igor's memos

IGOR's SIMULATOR

Igor Benko has used Java to craft a simulator for FLEET. It is event driven; i.e. it keeps a queue of Events and does them in time sequence. To “do” an Event, the simulator calls a method, named `execute()`, in an instance of class Command associated with the Event. Different parts of the simulated structure have different subclasses of Command whose `execute()` methods do the required work. Executing a Command nearly always results in adding more Events to the queue to represent subsequent actions.

The simulator uses three kinds of atoms to represent the elements of the FLEET system. For the moment I shall assign letters (A, B, C) but not names to these three kinds of things, they are:

- A) Things that can hold state to represent the values on wires or in registers
- B) Things that act on and modify those states, and
- C) Links between the two other types.

The structure connects things of type A and type B by things of type C. A link (C) always associates one A with one B. Any trip through such a structure will enter these items in the sequence . . . A C B C A C B C . . .

The links (C) solve what I think of as the “many-many” problem. Each instance of type A can connect to an arbitrary number of instances of type B and each instance of type B can connect to an arbitrary number of instances of type A. Interposing a link (C) to couple type A objects and type B objects helps simplify the pointer structure. Each A knows only about the links (C) attached to it, and each B knows only about the links (C) attached to it. Each link (C) knows about exactly one A and one B. Notice the symmetry in this structure.

Thinking of A and B as nodes and C as edges makes a bipartite graph. The nodes form two disjoint sub-sets and the edges connect a node from one sub-set to a node from the other sub-set. This view emphasizes the symmetry between A's and B's.

This document is a product of a collaboration between Sun Microsystems and the University of California at Berkeley.. The ideas contained herein are freely available for any academic purpose.

An alternate view prefers to ignore the links (C). This view thinks of Bs as nodes and As as edges. This view matches better the idea of actions coupled by wires that hold state. However, it requires allowing edges (A) to connect multiple nodes.

The simulator assembles these three types of objects, A, B and C into a hierarchy of Components, Groups and so forth. The hierarchy provides simple ways to compose structures from other compositions as well as from the atomic elements A B C.

You will notice that I have avoided naming the types of objects A, B and C. Descriptive names for them remain to be found. In his initial simulator, Igor assigns the name `state` to type A, the name `Action` to type B and the name `Terminal` to type C. I seek better names. State seems inappropriate for something that isn't a state in the logic sense; Action seems inappropriate for a type of object that makes changes; and Terminal invites confusion with connection places for compositions.

THE EXISTING CONDITION

I think of the existing version of Igor's simulator as a working draft. He has concentrated on the task of making code capable of simulating FLEET systems. In addition to the basic simulation code, Igor has provided ways to parse text descriptions of the structure to simulate, ways to observe and debug the structure, and ways to control and report on simulation progress.

We will be using the simulator to experiment with some FLEET structures. We will learn the structure of the simulator itself, seeking to simplify it if possible, but seeking especially to improve the understandability of its code.

Sun has released Igor's simulator for us to use. We can do anything we wish with it: modify it, use it, talk about it. The only things we can't do with it are 1) use it in a critical, illegal, or military action, and 2) sell it. I hope that over the semester our efforts will improve the simulator's function and especially make it more understandable.

A PROPOSAL ABOUT STRUCTURE

Table 1 below outlines some names. The Igor Name column shows the names of the classes as now assigned in Igor's simulator. The column called Ivan Name offers a suggestion for names that seem to me more suggestive of function. I shall use these name in the rest of this memo. The final column is left blank for your favorite names. A class discussion of names is likely.

Table 1: Names for things			
Type of thing	Igor Name	Ivan Name	Other Name
A	State	StateHolder	

B	Action	Actor	
C	Terminal	Link	
Composite	Component a subclass of Group	Component	
Attachment to a Component	Terminal	Port	
Data	Object	Value	

LINKS ARE SUBORDINATE

I suggest that Links be treated as subordinate to Actors and StateHolders. Only Actors and StateHolders will participate in the hierarchy. Thus a Component will be a set of Actors and StateHolders and Components. The links in such a Component will belong to it only implicitly. Moreover, Links can couple one Component to another without belonging to either, just as Links can couple a StateHolder to an Action without belonging to either.

STATEHOLDERS AND ACTORS

A StateHolder can be empty or full. If full, it will hold a particular class of data element that I shall call a Value. In the simulator as it now exists, a "State" holds the most general Object which can be anything. Does the StateHolder class need a boolean to represent full vs. empty, or does the null Value suffice, or should we provide a particular Value to represent empty?

StateHolders must notify all adjacent Actors when becoming full or empty to give those Actors a chance to do something if required. Similarly, when an Actor takes its action it must inform any of its adjacent StateHolders to become full or empty as appropriate. Moreover, if the StateHolder becomes full, the Actor must provide a Value for the StateHolder to remember.

COMPONENTS

A Component is a composite of StateHolders, Actors and Components. The Links that couple two parts of the same Component can be thought of as "belonging" to it as well, but I believe that this ownership should remain implicit.

We want to form compositions of Components. Because there are two kinds of atoms, namely Actors and StateHolders, we must be careful about how to link Components. For example, there are four ways we might define a FIFO Component. It might begin with either a StateHolder or an Actor, and it might also end with either. The Links that couple the beginning and end of the FIFO to other Actors, StateHolders or

Components lie outside the Component. We do need to specify, however, the external interface of the Component.

Let us suppose that the Ports of a Component are the StateHolders and Actors that are visible from outside it. We can connect Links to these Ports, or at least to unlinked parts of these Ports. A FIFO that begins and ends with Actors, for example, would have inputs available on its input Actor and outputs available on its output Actor to which fresh links could attach. The available places to attach Links are implicit given that the component identifies the StateHolders or Actors that are visible from outside.

REPRESENTATION

An abstract class Actor must represent what's essential about an Actor. An Actor must be able to attach Links. It must be able to enumerate the Links attached to it. But we need not specify in advance how a particular class of Actor actually represents its attachment to Links. Some Actors might need to separate input and output attachments, some might need to use arrays of Links to distinguish them by index, and so forth.

An abstract class StateHolder must represent what's essential about a StateHolder. A StateHolder is similar to an Actor in that it must be able to attach Links and enumerate the Links attached to it. In addition, of course, it must be able to hold a Value. Different StateHolders may wish to restrict the type of Value they are permitted to hold. Similarly, they may wish to represent their attachment to Links in various ways, for example coupling them into interfaces, characterizing them as Value sources or Value destinations, and so forth.

Let us suppose that the abstract specification for Actor and StateHolder require only that it be able to enumerate its existing Links, and specify something about the Link places that remain vacant. Such a definition might form the basis for Components as well. The Component could do a similar enumeration by composing the enumerations of all of the Actors and StateHolders on its periphery, namely its Ports. Naturally it would have to remove from such an enumeration any Link that couples two pieces of the same Component.

I think this representation make it possible to specify Components, as Jo likes to do, that encompass part but not all of an Actor. The part of the Actor that belongs to the Component is the part that has Links connecting it to other parts of the Component. The remaining connections remain to be assigned.

Suppose that there are two components, A and B. Suppose further that each has an Actor of type X as a Port. Coupling those two Ports together would superimpose one Port on the other. We could easily check that no Link connecting A's Port to other parts of A conflicts with a Link connecting B's Port to another part of B. One of the two superimposed Actors would be eliminated, leaving a fully connected structure that might

please Jo. Such “recursive merge” is reminiscent of a concept that appeared in Sketchpad.

SOME IMPLICATIONS

Can a Component consist of a single Actor or a single StateHolder? I should hope so. Such a Component treats its single part as its Port. Attaching to such a Component amounts to the same thing as attaching to its contained part.

Must we be careful about defining Ports on Components. Yes, because there are two distinct types of Ports. Thus there are four different FIFO definitions depending on whether we choose to begin or end with an Actor or with a StateHolder. If we choose to define FIFOs with Actors as both Ports, we will either need to insert a StateHolder between FIFOs or, by superposition of the Actors, eliminate the redundant Actor. FIFO definitions that start with an Actor and end with a StateHolder will easily compose into longer FIFOs.

The idea that a Component can have Actors or StateHolders as Ports generalizes to having Components as Ports. Such composite Ports may be very useful in building complex structures. For example, we often use a pair of StateHolders together, one to carry Values forward and the other to carry acknowledgements. We might define a StateHolderPair as a Component. The StateHolderPair would have its StateHolders as Ports, four in all, two references to each. One might connect such a StateHolderPair to an Actor.

Moreover, the interface to a FIFO that begins and ends with StateHolders might be very similar. Such a FIFO would have four StateHolders as its Ports, just as did the StateHolderPair, except the four Ports would refer to four distinct StateHolders instead of just referring twice to each of the same two.