(91.523): OLC Architecture User's Guide

August G. Reinig
supported by
Vandana Gandhi
Dolly Sinha;
(Intro and Requirement sections
revised by R Lechner on Oct 2 1996)
Path: $CASE/93su523/olc/base/doc/jparchitecture/newarchitecture.tex

November 10, 1997

Contents

1 **Introduction** 1

2 **Requirements** 2

3 **User's Guide** 2

3.1 Macros for building on different C compilers . . . 3
3.1.1 Token Pasting and Stringizing . . . 3
3.1.2 Function Prototypes . . . 3
3.1.3 Function Definitions . . . 4
3.2 Object Developers . . . 4
3.2.1 KEYBUFFER . . . 4
3.2.2 Reading and Writing Object Instance Data . . . 4
3.2.3 Deleting an Object Instance . . . 5
3.2.4 Looping Through All Instances of An Object . . . 5
3.3 Active Object Developers . . . 5
3.3.1 Creating Active Objects . . . 5
3.3.2 Sending Events . . . 7
3.4 GenerateEvent . . . 8
3.5 EventInstanceCreate . . . 8
3.5.1 ReceivingEvents . . . 8
3.5.2 Sending Events Immediately . . . 8
3.6 Event Processing . . . 9
3.6.1 The Event Queue . . . 9
3.6.2 ProcessEvents . . . 9
3.6.3 Event Choice Policies . . . 10
3.7 Timer Facility . . . 11
3.7.1 Timer Facility Code . . . 12
3.7.2 Testing Timer Facility . . . 12

4 **Juiceplant, the Juice Plant Simulator** 13

4.1 Command Line Qualifiers . . . 13
4.1.1 -s integer . . . 13
4.2 The Main Procedure . . . 13

5 **Testing** 14

5.1 The architecture . . . 14
5.2 JuicePlant . . . 15
5.2.1 Command Line Options . . . 15

6 **Futures** 15

1 Introduction

Large software systems are typically divided into *domains*. A domain is a separate real, hypothetical, or abstract world inhabited by a distinct set of objects that behave according to rules and policies characteristic of the domain. Domain are classified into four types according to the role each plays in the finished system.

1. Application domains.
2. Service domains.
3. Architectural domains.
4. Implementation domains.

The *architectural domain* provides generic mechanisms and structures for managing data and control for the system as a whole. The objects in the architectural domain include abstractions of data structures and units of code. Thus:

- Mechanisms supporting state machines and timers are provided by classes in the architectural domain. (The Timer class itself is in the service domain.)
- Policies and conventions for producing the application design are laid down by the architectural domain. These conventions are expressed as transformation rules that generate components of the design from components of the OOA models.

The *implementation domain* is the final step in the building of the software. In this domain the class templates of the architectural domain are populated using some language suitable for the design approach taken in the design of class templates.

This OLC Architecture Project comes under above last two domains, Architecture and Implementation. There were initially eight classes in Architecture Project, namely, State Model (SM), Active Class(AC), State(ST), Active Instance(AI), Transition(TR), Event Type(ET), Enable(EN), and Event Instance(EI). [1]

The Timer Service Class is also documented in this report. Later, a second Service Class Operator(OP) was added, with separate documentation [2]

[1] *Ref:* Data model for the olc93su523 schema in $CASE/95f522/95folc/base/doc/lcp_architecture.idraw

[2] *Ref:* User Guide on How the Operator Class Works in the Juice Plant Environment, by C Traynor.

The event types are associated with the state transitions which they enable by the associative entity or Enable (EN) table. During state model interpretation, one or more action routines are executed at each transition-dependent change of state (including re-entry to the same state).

The functions to be executed at the time of entry to a new state are defined as methods in the source code of the class definition `(*.c)` file, and the interface signatures of these methods are given in the corresponding class declaration `(*.h)` files.

In this report and in the initial implementation, each state (ST) instance contains the name and address of the function to be executed on entry into this state.

A subsequent revision of the *olc93su523.sch* schema incorporated a new Function Table (FT) passive class or table [3]

In a corresponding revision of the olc Architecture, the Event Dispatcher of the state model interpreter was modified to associate the newly entered state with a sequence of (zero or more) action routines by a new StateFunction map (a binary relation that is defined by the associative entity table SF).

[4]

[5]

2 Requirements

- The architecture was to follow the data model presented to us in 93su523 class. The original model shown in the appendix was later revised. [6]
- Chgen V7 was to be used to translate schema definitions into code to be used to support the object classes and tables described by the schema file. [7]

3 User's Guide

This User's Guide assume familarity with chgen, and the Object Life Cycle text. It covers the state model interpreter but not the application of chgen, nor the

[3] *Ref:* see $CASE/95s522/95solc/aareadme; also revised data models for the Juice Plant in $CASE/95f522/95folc/base/doc/lcp_architecture.idraw
and in
$CASE/95f522/95folc/base/doc/idrawMainItimer/opfig3.idraw
by bmehta who updated the main loop timing with setitimer.

[4] *Ref:* $CASE/95f522/95folc/base/doc/95folcMainItimer_md_dr.tex

[5] *Note:* Table SF implements a symmetric M:N relation between ST and FT via associative entity SF; this relation is traversed in the ST to FT direction during state model interpretation.

[6] *Ref:* $CASE/95f522/95folc/base/doc/lcp_architecture.idraw

[7] JPsim and olcArchitecture were upgraded to chgen v8 by glazar and msrdanov in $CASE/95s522/95solc/base/Master/JPsim/olc.

service classes TImer and OPerator.

3.1 Macros for building on different C compilers

In the process of developing the OLC architecture I used several different C compilers. These compilers had different features. To develop code compilable by the various different compilers I created a common include file `olc3common.h` provide a common interface to the differing features of the different compilers. `olc3common.h` may be found in `/usr/proj3/case/93su523/case/olc-/src/SCCS`.

The compilers which have successfully compiled the OLC architecture are:

- DEC C for OpenVMS VAX T1.3 (field test version)
- VAX C V V3.2004 on OpenVMS VAX
- cc on ULTRIX/VAX V4.2
- gcc version 1.36 on ULTRIX/VAX V4.2
- cc on machine CS at the University of Massachusetts at Lowell
- gcc on machine CS at the University of Massachusetts at Lowell
- cc on a MIPS Ultrix machine at the University of Massachusetts at Lowell
- cc on DEC OSF/1 V1.2. However since this is a 64 bit system and since chgen assumes that integers and pointers are both 32 bits no code could be run here.

3.1.1 Token Pasting and Stringizing

Different C compilers have different mechanisms for supporting token pasting and making strings of macro arguments. The `P()` and `S()` macros hide these mechanisms from the developer. `P()` takes two arguments and pastes them together so that `P(AC,id)` results in the token `ACid` . When using `P()` you cannot have a space before or after the command in the argument list. Such a space causes problems for old C compilers which use commentbased token pasting.

`S()` takes one argument and makes a string of it. `S(abc)` results in the string literal `"abc"`.

3.1.2 Function Prototypes

Old C compilers do not support function prototypes whereas most recent C compilers and all Standard C compilers do so. While one can not use function prototypes and thus write code which will compile on old as well as new C

compilers this means that one does not get the benefits that function prototypes provide. The `PROTOTYPE` macro creates a function prototype for those compilers which support function prototypes and a simple function declaration for those compilers which do not support function prototypes. `olc3common.h` explains how to use this macro.

3.1.3 Function Definitions

There are two forms of function definitions. The first form uses parameter type lists and the second form uses an identifier list followed by a declaration list. The first form allows for greater type checking by C compilers but not all C compilers support this form. The `FUNCTION`*n* macros create the first form of a function definition for those compilers which support this form. Otherize, the macros create the second form of a function definition.

`olc3common.h` explains how to use these macros and gives examples of the two different forms of function definitions.

3.2 Object Developers

In the process of developing the OLC architecture I noticed much commonality in the code I was writing for the various objects I was creating. By using the following macros I was able to generate the code for these macros rather than write it. For example, `EventInstance.h` exports fourteen routines, only two of which are code manually. The other twelve are macro expansions.

The include file `olc3common.h` provides these macros and may be found int `/usr/proj3/case/93su523/case/olc/src/SCCS`. Documentation on how to use these macros and sample expansion of these macros are contained in `olc3common.h`.

3.2.1 KEYBUFFER

Chgen has a concept of an object key and the text representation of that key. `KEYBUFFER(name)` defines a character array large enough to hold the text representation of an object key. It is shorthand for `char name[HCG_KEY_SIZE+1]`.

3.2.2 Reading and Writing Object Instance Data

An object instance often has internal data associated with it. Users of the object usually have to be able to read this data, and often have to be able to write this data. The `GETROUTINE()` macro creates a function definition which allows a user to read a particular field of an object. The `SETROUTINE()` macro createa a function definition which allows a user to write to a particular field of an object. The `COPYROUTINE()` macro allows the user to copy the characters of a text field of an object into a user supplied character array.

3.2.3 Deleting an Object Instance

The `DELETEROUTINE()` macro creates a function definition for an object instance delete routine. As with a direct call to `pr_delete` this routine does not delete an children of the object instance.

3.2.4 Looping Through All Instances of An Object

As an alternative to `table_loop()`, the `GETFIRST()` and `GETNEXT()` macros create function definitions which allow the user to loop through all instances of an object. There is no way for the user to specify interest in a particular subset of object instances.

3.3 Active Object Developers

Modified

An Active Object is any object that has a state model associated with it. In the OLC architecture, each active object is related to an ActiveClass and each instance of an active object is related to an ActiveInstance.

ActiveInstance objects receive events. When an ActiveInstance receives an event, the event usually causes a State transition. Events may also be ignored. When a state transition occurs, the event processor, 3.6, updates an the current State of the ActiveInstance calls the new current State's action routine 3.5.1.

The OLC text has a third class of events, Can't Happen Events. The OLC architecture does not handle Can't Happen Events. Active Object which wish to detect Can't Happen Events should extend their StateModels to have a Received A Can't Happen state and all Can't Happen Events

3.3.1 Creating Active Objects

To create the first instance of an active object, the developer must create the ActiveInstance related to the instance of the object. To do this the developer must first create a complete State Model. Following this, the developer creates the ActiveClass that uses this State Model. Following this, the developer creates an ActiveInstance of this ActiveClass. This ActiveInstance can then be used in the creation of the instance of the active object (with whatever additional information may be required for the particular active object's creation routine).

Creating additional instances of an active object requires creation of an additional ActiveInstance but does not require a new State Model or ActiveClass.

End Modified

StateModelCreateCompletely State Models contains States, EventTypes, Transitions, and several other objects (see the Appendix for the State Model's object model). While it is possible to create a complete State Model out of the individual building blocks the `StateModelCreateCompletely` routine will

build the State Model with one routine call. The interface to `StateModelCreateCompletely` is described in `statemodel.h` found in `/usr/proj3/case/93su523/case/olc/src/SCCS`.

Those wishing to build a State Model piece by piece should look at `StateModelCreateCompletely` to see how it does it, then forget the idea and use StateModelCreateCompletely.

An example on how to use `StateModelCreateCompletely` may be found in `test_architecture.c` found in `/usr/proj3/case/93su523/case/olc/src/SCCS` and is reproduced below.

```
/*
** Create the following state machine
**
**      AGR1                 AGR1
**    +------+     +----------------------+
**    |      |     v                      |
**    |   +------------------+   +-------------------+
**    |   | The first state |   | The second state |
**    |   +------------------+   +-------------------+
**    |      ^     |          AGR2        ^
**    +------+     +----------------------+
**
*/

static EventTypeList EventTypes[] = {
    {"AGR1", "Move to State 1"},
    {"AGR2", "Move to State 2"}};

static StateList States[] = {
    {"TheFirstState", "ActRtn1", ActRtn1},
    {"TheSecondState", "ActRtn2", ActRtn2}};

static StateTransitionList Transitions[] = {
    {"TheFirstState",  "TheSecondState", "AGR2"},
    {"TheFirstState",  "TheFirstState",  "AGR1"},
    {"TheSecondState", "TheFirstState",  "AGR1"}};

SMid = StateModelCreateCompletely(
            "August's State Machine", "AGR",
            ARRAY_SIZE(EventTypes),   EventTypes,
            ARRAY_SIZE(States),       States,
            ARRAY_SIZE(Transitions),  Transitions);
```

The EventTypes array names and describes all the events in the State Model. Events are the labels on the arrows in a state machine diagram. As the descrip-

tion field of the Event Type is the last field in the schema file for the architecture, it may contain spaces. See `eventttype.h` in `/usr/proj3/case/93su523/case-/olc/src/SCCS` for EventType name and description maximum lengths.

The States array names and provides the action routine names and pointers for all states in the State Model. States are the boxes in a state machine diagram. The action routine name may not contain spaces.

While the name field of the State is the last field in the schema file for the architecture, it may not contain spaces. The state name is used by the ActiveInstance object to indicate the instance's current state in the state routine. This allows the current state information to be stored via `pr_dump` and reloaded via `pr_load`. The `STid` of the state may not be used because that may change version number across a `pr_dump pr_load` pair. Storing the `STid` as a data field in the ActiveInstance would result in the field having the incorrect value. Attempts to store the `STid` as a foreign key field were unsuccessful.

The Transitions array describes all the transitions in the State Model. Transitions are the arrows in a state machine diagram. The FromState is the box from which the arrow emanates and the ToState is the box pointed to by the arrow head.

The example uses static arrays because not all C compilers support initialization of automatic structures. The `ARRAY_SIZE()` macro computes the number of entries in an array and is defined in `statemodel.h`.

ActiveClassCreate After creating an StateModel the Active Object coder must create the ActiveClass for the object. An ActiveClass has a name and an associated StateModel, nothing else. The name may include spaces. The routine to create an Active Class is `ActiveClassCreate` exported by `activeclass.h` which may be found in `/usr/proj3/case/93su523/case/olc/src/SCCS`.

The test program mentioned in 3.3.1 shows a sample call to `ActiveClass-Create`.

ActiveInstanceCreate `ActiveInstanceCreate` creates a specific instance of an ActiveClass. Each ActiveInstance has a name (spaces are allowed), and a current state. The initial current state one of the arguements to `Active-InstanceCreate`. Different Active Instances may start at different states in the State Model associated with the Active Instance's Active Class.

The test program mentioned in 3.3.1 shows a the creation of two Active Instances, each starting in different states.

3.3.2 Sending Events

When an active object wishes to direct some other object to do something, it sends an event to the other object. There are two routines to do so. The first identifiers the sender, receiver, and event type via chgen `hcg_keys`. The second uses the name/label of the sender, receiver, and event type. If you

have a mixture of names and keys, it is easier to turn a key into a name via `ActiveInstanceGetName` or `EventTYpeGetLabel` than it is to turn a namento a key.

`eventinstance.h` exports both routines. It is found in `/usr/proj3/case-/93su523/case/olc/src/SCCS`.

An EventInstance carries with it five pieces of event data consisting of two integers, two floating point numbers, and one character string. The event sender and receiver must agree to the meaning of this data. The OLC architecture does not look at the data or put any interpretation on it.

3.4 GenerateEvent

`GenerateEvent()` sends the labeled event from one named Active Instance to some other named Active Instance. It converts the event label and the active instance names into chgen keys and calls `EventInstanceCreate`.

3.5 EventInstanceCreate

`EventInstanceCreate` sends an event from on Active Instance to some other Active Instance. Actually delivery of the event occurs at some future time, governed by the event processing policies currently in place 3.6.

3.5.1 ReceivingEvents

When creating a StateModel, the developer must specify an action routine for each state in the StateModel. When the event processor 3.6 changes the current state of an ActiveInstance, it calls the action routine associated with the new current state. The action is responsible for ensuring that the StateModel is in a consistent state upon exit.

Action routines are normal C functions and may do almost whatever processing they wish, including deleting the Active Instance receiving the event. They may generate events, set timers, perform calculations, etc. They should not process other events 3.6 and they must eventually return. If an action routine fails to return no other event processing will take place and the simulator will hang.

3.5.2 Sending Events Immediately

New Section

There is nothing to prevent an ActiveInstance sending an event to itself immediately, or for that matter to any other ActiveInstance. The `ActiveInstanceSetState` routine is available to all and sets the current state of the ActiveInstance. A State's action routine is available through `StateGetActFunc`. It's currently not possible to get the create an EventInstance and get it's EIid to pass to the new current State's action routine.

The action routine sending the immediate event can send on the id of the EventInstance it received, if the data therein is appropriate for the immediate event. Alternatively, it can call some other action routine that has a non standard interface. The alternate action routine must leave the state machine associated with ActiveInstance receiving the immediate event in a consistent state.

ActiveInstances sending immediate events to other ActiveInstances (or too itself) **MUST NOT** also use `GenerateEvent` or `EventInstanceCreate` to send events to these other ActiveInstances. Sending both immediate and normal events to an ActiveInstance can result that active instances receiving events out of order, violating the rules about events on page 106 of the OLC text. New

3.6 Event Processing

`processevent.h` exports and documents the event processing routines. It may be found in `/usr/proj3/case/93su523/case/olc/src/SCCS`.

3.6.1 The Event Queue

`GenerateEvent`, 3.4, and `EventInstanceCreate`, 3.5, add EventInstances to the Event Queue. There are no other supported mechanisms to do so. In particular, nothing should try to do an explicit `pr_add` of an EventInstance.

`ProcessEvents`, 3.6.2, removes events from the event queue. There are not other supported mechanisms to do so. In particular, Action Routines should not us `pr_delete` on the `EIid` passed to them.

The Event Queue is a partially ordered queue. Event Instances added to the queue will always be processed subject the following constraint:

$$\begin{aligned} \forall EIid_1, EIid_2 : \quad & AIid1(EIid_1) = AIid1(EIid_2) \\ \wedge \quad & AIid2(EIid_1) = AIid2(EIid_2) \\ \wedge \quad & Generated(AIid1(EIid_1), EIid_1) \\ & \text{before}\ Generated(AIid1(EIid_2), EIid_2) \\ \Rightarrow \quad & Processed(EIid_1)\ \text{before}\ Processed(EIid_2) \end{aligned}$$

That is, if an ActiveInstance sends an two events to another ActiveInstance, the receiving ActiveInstance will receive the events in the order the sending ActiveInstance sent them. No additional ordering of the Event Queue is guaranteed.

3.6.2 ProcessEvents

The `ProcessEvent` routine handles all event processing. It

1. Chooses an EventInstance to process. Following the choice guideline specified by its caller.

2. Locates the ActiveInstance receiving the event and updates its current state.

3. Calls the ActiveInstance's new current State's action routine, passing to the routine, the EventInstance's id so tthat the action routine can access the event data.

4. Upon return from the action routine, deletes the EventInstance.

It repeats this either until there are no more events to process or until it has processed the number of events the caller requested.

`ProcessEvent` is **NOT** reenttrant. Thus, it should only be called by the main loop controlling the simulation. If `ProcessEvent` were to be called from within an action routine while processing an event, it is possible that `process-event` will choose that event currently being processed as the next event to process, and reprocess it. This is due to `ProcessEvent` not deleting the EventInstance until after the action routine returns.

3.6.3 Event Choice Policies

As documented in `processevent.h`, there are four policies `ProcessEvent` can use when choosing the next event to process.

- `EventQueueOrder` Choose the next event in the event queue. Under the current implimentation of a single event FIFO queue and a single thread of control, this means that events are processed in the order genereated.

- `ActiveInstanceOrder` Loop through the ActiveInstances, processing one event per Active Instance.

- `ActiveClassOrder` Loop through the Active Classes, processing one event per Active Instance in the class.

- `RandomOrder` Choose an EventInstance at random, subject to the partial ordering contraint and process that event. The current implementation randomly choose one of the first twenty EventInstances outstanding with equal probability. If there are fewer than twentry EventInstances outstanding then the choice is amoung these, again with equal probability. The reason for limiting the choice to a maximum of twenty events is to reduce the likely hood that an EventInstance will never be chosen.

Consider the case were each EventInstance causes the generation of two more EventInstances. As time goes on, it becomes increasingly likely that a particular EventInstance will never be processed. With the choice restricted to the first twenty EventInstances, will still possible, it is highly unlikely that a particular EventInstance will never be chosen. End New

3.7 Timer Facility

A Timer is a mechanism that can be used by an action to generate an event at some time in future.

The timer implementation consists of the following actions:

- CreateTimer
- SetTimer
- DeleteTimer
- ResetTimer and
- TimerTick.

To use timer facility one has to first call CreateTimer. CreateTimer will return a Timerid key. You can now call setTimer to fire an event after a specified interval of time. ResetTimer will reset the time to go to zero. DeleteTimer will delete the entry from the TimerView table. TimerTick goes through the TimerView table and reduces remaining time for each entry by clocktick.

- **CreateTime:** when called returns a key to an entry in Timerview table.
- **SetTimer:** This is called to set a timer to generate an event after a specified interval(time-to-go: ttg) of time parameters for SetTimer are :
  1. Timerid: which timer to set
  2. ttg: time to go
  3. Evlbl: event label to be fored at the end of ttg
  4. AIid1: active instance id for superclass
  5. AIid2: active instance for client
- **ResetTimer :** this is called with a Timerid to reset ttg entry of Timerid to zero. The entry is removed from the timer table Timerview.
- **DeleteTimer :** this is called with Timerid to remove the entry Timerid from table Timerview.
- **TimerTick :** This reduces ttg for every entry in Timerview table by clocktick (DELTA). If an entry is set to zero the corresponding event is fired.

3.7.1 Timer Facility Code

Timer facility is located in

/usr/proj3/case/93su523/olc/dsinha/timer.c

This file has five routines: CreateTimer, SetTimer, ResetTimer, DeleteTimer and TimerTick.

The schema file for Timer is located in

/usr/proj3/case/93su523/olc/dsinha/timer.sch

The include file timer.h can be generated by running chgen version 7.0

3.7.2 Testing Timer Facility

To test Timer facility I have written a test program, which can be found in file main.c. It first initializes Timerview table it then calls CreateTimer to generate timerkey. It then sets the time-to-go (ttg) for timerid (pkey1) to 60 seconds by calling SetTimer.

It then creates a second (pkey2) and third (pkey3) timer entry with ttg set at 45 and 30 seconds respectively.

Now we dump the Timerview table on file timer1.dat. Expected content of the file timer1.dat is

```
TI000001 AI010101 AI020201 EV000101         60
TI000002 AI010002 AI020002 EV000102         45
TI000003 AI010003 AI020003 EV000103         30
```

We now let the clock go 20 ticks to accomplish this we call routine TimerTick 20 times. Each time it is called it reduces ttg for each entry by Timerview by one tick (DELTA). At the end of 20 calls it dumps Timerview in file timer2.dat. Expected contents of file timer2.dat is:

```
TI000001 AI010101 AI020201 EV000101         40
TI000002 AI010002 AI020002 EV000102         25
TI000003 AI010003 AI020003 EV000103         10
```

It now tests DeleteTimer by deleting entry for pkey2. It then dumps the Timerview file in timer3.dat. Expected contents of file timer3.dat is:

```
TI000001 AI010101 AI020201 EV000101         40
TI000003 AI010003 AI020003 EV000103         10
```

It now tests TimerFire. This is accomplished by calling TimerTick 10 more times. At the tength time ttg for pkey3 will become zero and the corresponding event should be fired. Also the entry pkey3 should be deleted from the

Timerview. Note test program does not generate any event rather prints the Timerid value. The call to GenerateEvent is commented out and should be uncommented eventually. It then dumps the Timerview file in timer4.dat. Expected contents of file timer4.dat is:

```
TI000001 AI010101 AI020201 EV000101        30
```

4 Juiceplant, the Juice Plant Simulator

New

4.1 Command Line Qualifiers

All command line qualifiers must follow the rules specified by `getopt(3)` found in most UNIX libraries and all POSIX compliant run time libraries.

4.1.1 -s integer

This qualifier indicates how many real time seconds should pass for each simulated second that passes. The argument to is an integer. It indicates the number of seconds to sleep between each time through the main loop.

4.2 The Main Procedure

The main procedure for the Juice Plant Simulater generally follows that specified in the OLC text (page 189), though there are a few changes. The main procedure

- Parses the command line
- Uses `pr_load` to load data from the chgen database
- Invokes the class initialization routines. These routines must:
  1. If an Active Class, create the Finite State Machine for the Active
  2. If an Active Class, create the ActiveClass
  3. Create an instances of the class unless `pr_load` has already done so.
  4. Some Class initialization routine must send an event to an existing ActiveInstance to kick off the simulation
- Enters the main loop which
  1. Prints out the current simulated time
  2. Calls the Update procedures to update the tank fluid levels and temperatures.

3. Calls TimerFire() to tell the timer that a second of simulated time has gone by. Events sent by timers going off will be processed in the simulated second the timer go off, rather than a second later.
4. Calls the event processing routines for ActiveClasses that do not use the common OLC architecture routines.
5. Calls `ProcessEvents` to process ALL outstanding events in random event order. This processing takes no simulated time.
6. Sleeps for the command line specified number of seconds.

The main loop exits after 100 seconds of simulation, for no other reason than there is as yet, no other way to terminate the simulation.

- After terminating the main loop, the main procedure use `pr_dump` to dump the current state of the simulation.

5 Testing

5.1 The architecture

The testing methodology followed is described in detail in `Testing.txt` under `/usr/proj3/case/93su523/olc/base/doc` , also description of `Bug.txt` and `testlist.txt` in `/usr/proj3/case/93su523/olc/base/src/SCCS` is given in `Testing.txt` .

As usual with most programming projects, the OLC architecture has not been adequately tested. `test_architecture.c` implements a simple test of the architecture using a very simple two state state machine. There is one ActiveClass, and two ActiveInstances of this class. The ActiveInstances send events to each other.

The action routines for the States in the State machines dump out information about the EventInstance whose id the routines receives as arguments. This, combined with the `EventInstanceReport` in `ProcessEvent.c` show that one can read all the fields of an EventInstance so event data is transmitted correctly from ActiveInstance to ActiveInstance.

An admittedly not so careful perusal of the output of `test_architecture` shows that events do go where they are supposed to, when they are supposed to.

Examining of `test.dat"` after the completion of `test_architecture` shows the chgen database looks correct.

Currently, `test_architecture` runs on all tested compilers except for those on OpenVMS VAX. On OpenVMS VAX, the `test_architecture` access violates. Some preliminary investigation has been done but no conclusions about the cause have been reached.

5.2 JuicePlant

Very little testing of the actually simulator has been done due to the lack of code.

5.2.1 Command Line Options

The `-s` option works. The only way to test this is to manually run the simulation with different values and watch the program run faster or slower. Fortunately, this is very simple code, relatively unimportant, and easy to see if it is working correctly.

End New

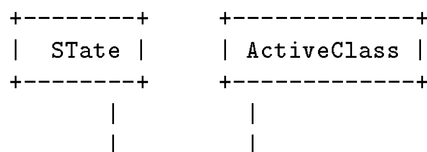
6 Futures

- Not all states need action routines. Some states are nothing more than wait states, waiting for a specific event to occur. There is no actual processing done upon entry to the state. The midterm for 91.523 had examples of such states in the light computer problem. The event processor should check if the state's action routine pointer is not 0 before calling it.

- The space character is much too useful a character to not allow in text chgen text fields. Chgen should be modified to use some character other than a space to delimite text fields, perhaps the double quote character, ". The `strtok` function can handle changing separator strings and chgen knows when to expect a text field so this should not be too hard.

- In addition to looping through all instances of an object, macros to generate routines to loop through all instances with a particular child, parent, or field value would be useful.

- The one-to-one relationship between ActiveClass and StateModel should be relaxed. There is no reason why several ActiveClasses could not share a StateModel. Consider the simple On/Off State Model. Many different ActiveClass might want to share model. Doing this requires a change in the object model for the architecture. Instead of a State knowing the action routine associated with it, there would be a relation between the State and several ActionRoutine objects. There would also be a relation between an Active Class and several ActionRoutine objects. Something like:



```
  V        V
+---------------+
| ActionRoutine |
+- - - - - - - -+
| Name          |
| Function Ptr  |
+---------------+
```

When doing a state transition, the event processor would have to look through the ActionRoutine objects for an object related both to the Active Instance's new current state and to the ActiveInstance's ActiveClass. This would also allow for the implementation of multiple action routines associated with a state transition. The event processor would simply look for all such objects, instead of just the first. If order of execution were important, the some sort or order criteria would have to be implemented.

- The interface to `SetTimer()`, as specified in the OLC text, does not allow for the passing of any event data (nor or the name of active instance sending the event). The interface to `SetTimer()` should be extended to accept event data and the name of the ActiveInstance calling SetTimer().

- Currently, the StateModels must be created each time JuicePlant is run or the Action Routine pointers may be incorrect. `pr_load` will not work if `juiceplant.exe` is rebuilt as the action routine pointers will be incorrect. There are no routines to set the action routine pointers in a state. The `SETROUTINE()` macro can be used to create a routine to set the action routine pointer of one state. A routine that accepts a StateList (from `statemodel.h`) and sets the action routine pointer of all the states therein would be easy to write. However, there should be a more automated way to do this, i.e. `pr_load()` should load the data base and reset the action routine pointers somehow.

New Item

- Make `ProcessEvent reentrant`. To do this, `ProcessEvent` must delete the EventInstance being processed before calling the action routine to. So that it can do this, the EventInstanceData must be separated from the EventInstance. Then, `ProcessEvent` would delete the EventInstance before calling the action routine and pass to the action routine the EventInstanceData id.

New Item

- Increase the number of `ProcessEvent` selection policies. One could prioritize on EventTypes, ActiveClasses, or ActiveInstances, processing all events of one type before moving to another type.

New Item

- Since `sleep(3)` only sleeps for whole seconds, there is no way to speed up a simulation, other than to have it run as fast as possible by never sleeping. To make simulated time run faster than real time, but not at

full speed, `TimerFire` should take an integer argument telling it how many ticks to count off. This can be initially controlled through a command line switch, meaning the new switch and the existing `-s` switch would control the speed of simulation. Alternatively, one could use `setitimer(2)`.