OpenMASK Programming Guide

David Margery

October 21, 2002 version

Chapter 1

Design Issues when Using OpenMASK

In this chapter, we will try and sum the important design decisions that have impacted the current version of OpenMASK. Some of these decision are fundamental, some are related to the cost of development, and some are simply implementation choice that could be reverted at very little cost.

1.1 A different object oriented paradigm

The main design decision of OpenMASK is in the choice of the programming paradigm used. OpenMASK is implemented in C^{++}, and makes a heavy use of genericity and inheritance. Nevertheless, in you are programming objects with the OpenMASK library, the main difference in the approach with respect to most programming languages, is that object in OpenMASK should share data and not member functions. If OpenMASK were a programming language comparable to C^{++}, the programming style would coerce you to make all member functions protected and to make some of the data of an object publicly readable.

It is important to stress that this is a very fundamental design decision that was made when designing OpenMASK. Users of the OpenMASK library often refer to this rule as *objects in OpenMASK cannot interact using function invocation*. In this respect, the OpenMASK library requests an agent oriented programming style from its users.

1.1.1 Benefits

Of course, such a fundamental choice departing from traditional programming paradigms was made expecting some benefits in return. That main benefit of this approach is that parallel and distributed executions of simulations programmed using the OpenMASK library are possible, with no change in the program.

An other benefit is very high modularity, as two simulated objects won't directly interact, but will interact using the communication primitives given with OpenMASK. As run-time inspection primitives on those communication primitives are available, it is possible to write very generic code.

Last benefit, the structure of an application becomes very constrained, and therefore makes rapid development much easier.

1.1.2 Drawbacks

The main drawback of not sharing the address space between simulated objects is that any data sharing relying on dynamic linking (for examples using abstract classes with virtual member functions) must have dynamic linking supported by the OpenMASK kernel. In OpenMASK3.0, public attributes are implemented in a way that breaks dynamic linking (until virtual outputs are implemented), whereas dynamic linking is possible with events, but not with the default type PsValuedEvent.

These drawbacks are the result of implementation choices related to the complexity of the implementation and to available time. There is not reason futures version of OpenMASK couldn't correct this drawback.

1.2 Other Implementation Issues

These implementations issues are due to lack of time, as they could be complicated to change.

1.2.1 Cannot Know when The Constructor of an Object will be Called

All is said in the title. The reason for this is that dynamic inspection capabilities are implemented by querying contructed objects. Therefore, some requests to the kernel trigger object creation and therefore, a call to the constructor of some of the objects.

1.2.2 Synchonous Data Structures For Scheduling

Figure 1.1 describes three different views of object scheduling with OpenMASK, when frequencies are specified for object activation.

The first view is an abstract view when objects with different frequencies are scheduled. It is very important to understand that in the current version of OpenMASK (3.0), all scheduling is done in a synchronous fashion, meaning sheduling of an object never overlaps between two shedulings of any other object. This explains the netxt two views, the first in simulated time, the second in processor time of scheduling.

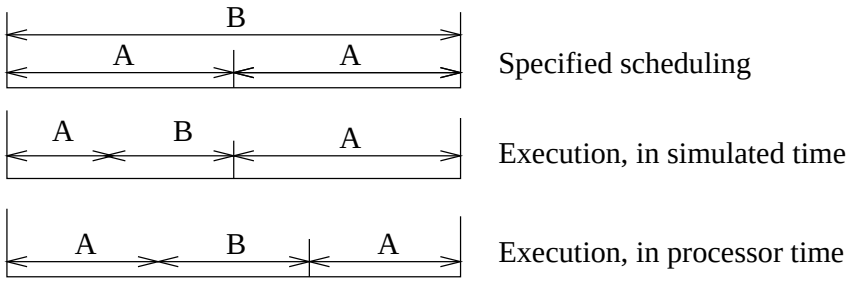

Figure 1.1: Different views of synchronous scheduling

This design decision is deeply related to timestamping in the execution kernel, and isn't a trivial matter to change from a conceptuel point of view. Nevertheless, this could evolve provided the timestamping mechanism is rethought.

1.3 Simple Implementation Issues

The `PsEventCreator::declareAlias()`s implementation supposes one of the two parameters has already been added to the EventCreatorContainer. This is certainly true when only one alias needs to be declared, when one is generated by static contruction. Things might get more difficult when more than on alias is needed (an heterogeneous system with more than 2 naming conventions).

The suggested solution is to generate for a second time (using typeid().name()) the locally used name, and use it as a key for each declared alias.

Chapter 2

A General Overview

2.1 Design Goals

The main goal of OpenMASK, previously named GASP (General Animation and Simulation Plateform), is to provide a basic animation and simulation kernel with the following functionality

1. the kernel should be animation level (descriptive, generative or behavioral) agnostic.
2. the kernel should be animation programming style (reactive, agent oriented, active object ...) agnostic
3. the kernel should be rendering library agnostic
4. the kernel should be capable of multi-threaded execution of the animation or simulation
5. the kernel should be multi-threading style (distributed computing, parallel computing, shared memory architectures) agnostic

Of course the last two points of these main objectives has greatly influenced the conception of the kernel, as we hope to be able to provide performant multi-threading. Furthermore, defining what needs to be simulated or animated does influence the programming style of the object animated or simulated. Finally, because some programming styles are easier to use for some animation levels, the presented kernel isn't quite animation level neutral.

Nevertheless, the different design tradeoffs are rather biased towards multi-threading than any predefined animation programming style or level. Furthermore, the tools provided by OpenMASK aim to enable any programmer to express his animation or simulation with the best suited tool for his work, empowering the kernel programmer to optimise using the semantic information gained by guessing that if one tool has been preferred over another, that choice was probably made with some intention.

In this paper, the terms animation and simulation are used with slightly different meanings. We use simulation when the produced result (which might not be visual) is more important than the time it takes to produce it, and animation for interactive simulations when the time to produce results or to take into account user interaction does matter. We believe that one should be able to use the same components for both, and that depending on execution context, different optimizations should be performed by the run-time kernel. Furthermore, as this paper is mostly focused on presenting how the kernel should be used to program new components, execution context is unknown and mostly irrelevant. Therefore, simulation of simulated objects will be discussed, without presuming whether the simulated objects are used in an interactive or off-line simulation.

2.1.1 OpenMASK Base concepts

In Mask, the basic building block of an application, is the *simulated object*[1]. It is inside a simulated object that all code for the evolution of the object and the communication with other objects is located. Therefore, the two main questions one has to ask are:

1. when is that code executed ? This is the activation problem.
2. how do simulated objects communicate between one other ? This is the communication problem.

The third question one would want to ask is: what is the granularity of a simulated object ? Or what does a simulated object animate ? This is a question we don't want to answer, as the kernel has to be animation level and style agnostic. Experience with OpenMASK shows that simulated objects vary in granularity from complete virtual humans with complex behaviors to a simple inert sphere, and this in the same application.

2.2 Communication between objects

Simulated objects for OpenMASK communicate by many means. The fundamental point to understand, is that communication should only be done with the tools (the classes and functions) provided with the kernel. In particular, as a general rule, no member function invocation should be used. The rational for this, is that multi-threading introduces thread safety problems. The kernel can manage these thread safety problems so that programming a simulated object doesn't require comprehension of the multi-treading paradigm used. But allowing method invocation between objects requires that objects be designed with multi-threading in mind, so that data integrity can be maintained. This is the fundamental design trade-off made during the conception of OpenMASK: to allow for performant multi-threaded run-time kernels, method invocation is forbidden between simulated objects, even if it's possible from a programming point of view, and in specific cases perfectly compatible the general framework presented here. But as these special cases require a good understanding of the general framework, they won't be discussed in this section.

OpenMASK distinguishes several communication styles between simulated objects. The basic communication style is an object reading the attributes of another object in a regular fashion. Attributes are made available for reading by other objects with outputs and control parameters, and are generally read through inputs. The other base communication style is through signals and eventListeners, designed for sporadic communication between objects. From these notions are derived the event for one-to-one communication.

2.2.1 Outputs

An *output*[2] is used when an object has an attribute (it's position for example) that it makes publicly available for reading and whose value is always pertinent: whatever the simulation date, it's value is meaningful. Therefore, interpolation and extrapolation of output values is legitimate. Nevertheless, the type or the semantic of data stored in an output may not be appropriate for default polation[3]

[1] named `PsSimulatedObject`

[2] named `PsOutput`

[3] polation is defined as one of the following : interpolation, extrapolation or antepolation. It produces a value using the values previously set by the simulated object

methods. Therefore, to each output is associated a polator, and output creation is done using the `addOuput` member function, which takes 3 parameters:

1. a name for the created output
2. a type, that inherits from PsType
3. (optional) a polator. If no polator is given, the default polator for the type is used.

The most simple polator is called a naive polator[4], and is pertinent for every type, as it only returns values stored in the history FIFO (of a given length) associated to the output. Other polators use those stored values to calculate any required value.

The list of all the outputs of a simulated object are stored in an output table[5], which can be queried by any object to find the attribute of an object.

The simulated object owner of an output can change its value using the `set` member function, and query the current value of the output using `getLastExactValue`.

Advanced topics for outputs

When `set` is called, the parameter of that member function is copied into the history FIFO (which length is calculated by the kernel or read from an environnement variable) maintained by the kernel for each output. When the type of the output is large, or the copy operation costly, one can use a more advanced mechanism that enables the simulated object to use the memory block that would be written by the next set for it's computations. This is done using the `getNextPlaceHolder` member function to get a reference to the seeked memory block, and `setInPlace` to unprotect that memory block so it can be read by the connected inputs. The cost of such a mechanism is paid with distributed OpenMASK kernels: because output values are distributed, any memory allocated during the copy of a value in the output needs to be distributed with the output. Therefore, any dynamic memory allocation done between `getNextPlaceHolder` and `setInPlace` is more costly than basic memory allocation. If the memory allocated doesn't relate to the output value, distribution costs might grow unnecessarily.

From a C++ point of view, nothing prevents one simulated object to call set on the outputs of a second object, because of the way data protection (private or protected) works. Nevertheless, the code has been designed assuming only the reference version (more about that later) of a simulated object will call `set` or `setInPlace`. Data integrity is not guaranteed should someone choose to ignore that principle. Moreover, in debug mode, this will make the application fail. If your design requires that more than one object is able to use `set`, please look at the the section on *control parameters*.

2.2.2 Control Parameters

A *control parameter*[6] is a special type of output for 2 reasons. The first, is that any simulated object can suugest a new value[7] for a control parameter, and the second is that a control parameter isn't stored in the list of outputs of its simulated object, but in the list of control parameters[8]. Nevertheless, it's possible to connect an input to a control parameter.

[4] named `PsPolator`
[5] named `_outputTable`
[6] named `PsControlParameter`.
[7] by calling `set`
[8] `_controlParameterTable`

When `set` is called on a control parameter, a valued event is sent to the owner of the control parameter with the proposed new value. By default, this event is interpreted by an event listener that will change the value of the control parameter according to the last event proposing a new value received by the owner. The only way of changing that behavior, is by redefining the event listener associated to the control parameter. More about this in the section dedicated to events and event listeners.

2.2.3 Inputs

An *input*[9] is used by simulated objects to read the outputs (in the more general sense : outputs and control parameters) of other simulated objects. There are two sorts of inputs:

1. private inputs: the connection of these inputs to outputs can only be done at the request of the simulated object owner of the input.

2. public inputs: they have the properties of private inputs but also accept connections to other outputs can be done at the request of the owner of the output. When a connection is requested, a connection event is sent, and automatically handled by the default event listener associated to the input.

Public and private inputs are different only by the value of a boolean authorising connection requests originating from other simulated objects.

Reading from inputs

The default method for reading the value of an input is by using the `get()` member function of the input. This member function accepts one optional parameter that is interpreted as the delay between the current simulated date and the date of the value that is returned by `get()`. For example, `get(0)` will return a value calculated from the connected output for the current simulated date. `get(20)` will return a value calculated as being the one of the output at a simulated date 20 ms seconds before the current simulated date. Those calculations are done using the polator associated to the output. If a negative value is passsed as a parameter, a estimate for a future value of the connected output will be returned.

In the event that a produced value is required, `getLastExactValue()` should be used.

In the event that the input is used to detect changes in the value of the connected output, a derived class should be used: the *sensitive input*[10]. Inputs of that class can be queried using the `valueChanged()` member function. If a *sensitive notifying input*[11] is used, an event will be sent to the owner of the input each time a new value is given to the connected output.

2.2.4 Signals

A *signal*[12] is a piece of information released to the rest of the simulation by a simulated object. The basic signal has no value attached to it, except an identifier used to distinguish between different signals. Therefore, valued signals are introduced so that signals can be made to carry data.

For an object to be aware of signals released in the environment by other objects, it must register. If one object is only interested in signals originating from a particular object, registration

[9] named `PsInput`.
[10] named `PsSensitiveInput`.
[11] named `PsSensitiveNotifyingInput`.
[12] named `class PsSignal`.

should be done through that object. Otherwise, it is done through the *controller* (see 18). When registering for a signal, an object can decide what information it will receive when the signal is released. That information is either an event prototype or the identifier of the signal if no event prototype is specified.

Indeed, when a signal is released in the environment, all registered objects receive the corresponding event, depending on the optional event prototype registered for that signal.

System Signals

During the simulation, the controller fires system signal to notify interested objects of particular events in the simulation. These events are the creation or the destruction of an object, the recomputing of the scheduling data structures, etc.

2.2.5 Events and valued events

An *event*[13] is composed of four fields. Its sender, its receiver, the date of sending and the event identifier, which should be viewed as a string identifying the event. A valued event is an event to wich a value field is associated. That value can be of any OpenMASK conforming type[14]. Simulated objects never directly receive signals. Instead, they receive events when signals are fired in the environment by the controller or other object. They can also receive events that where sent as events by other objects, for one to one communication.

2.2.6 Event processing and event listeners

Event processing is done at the *object handle* (see 19) level. All events received by a simulated objects are ordered as they arrive at the object based on

1. their associated date
2. their arrival order

Before any event processing takes place, the `prepareEventProcessing` member function of the simulated object is called. Then each event is processed, first by looking for an event listener registered for the corresponding event identifier.

1. If one is found, it receives the event for processing. Depending on the result of that processing, the event is then propagated along the event listeners list (thus allowing redefinition or overloading).
2. If such an event listener doesn't exist, or if its treatment of the event is incomplete[15] returns false), the `processEvent` member function of the simulated object is called. After all events were processed, `eventsProcessed` is called, to enable any necessary post-processing.

[13] named `PsEvent`.
[14] ie inherits from `PsType`
[15] in which case `processEvent`

Event listeners

The way an object reacts to certain events can be part of its interface, and thus needs to be preserved through inheritance and be visible through reflection. *Event listeners*[16] serve this purpose. They are code fragments associated to event identifiers and which are called when an event with the corresponding event id is received by the object and should be created in the constructor. When two event listeners are registered for the same event, the last one registered is called first, to enable redefinition of the reaction to an event by a derived simulated object.

2.3 Activation Policy

Each simulated object has two activation methods. The first, the `compute` member function, is called at a certain frequency for objects with a frequency in the active state (see the state diagram). The second is related to event processing. For suspended objects, event processing takes place at the rhythm events are received, but not faster than the controller's frequency. For active objects, event processing takes place at the frequency of the simulated object if the `processEventsASAP` returns false (default policy), as soon as possible otherwise.

2.3.1 Simulated Object State

As implied by the previous paragraph, a simulated object can be in different computing states, as described by the following diagram (fig 2.1).

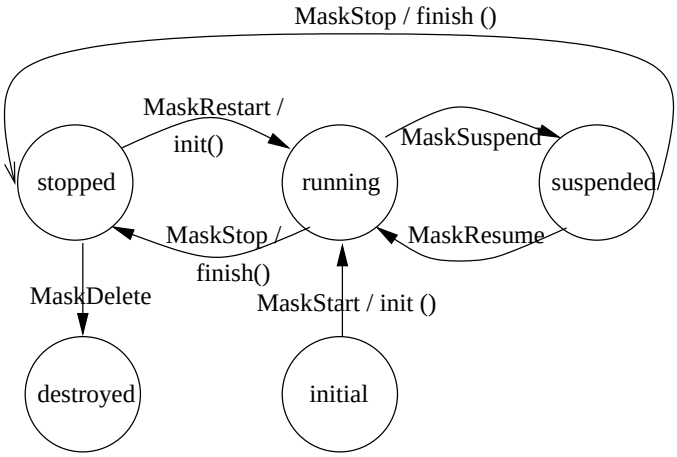

Figure 2.1: State diagram for a simulated object

2.3.2 Scheduling Policy

The base scheduling block for OpenMASK is the simulation step (see fig 2.2, where As frequency is 10Hz and Bs frequency is 30Hz). All scheduling takes place within the borders of a simulation step. For the scheduling perspective, OpenMASK is a synchronous system. One object cannot be scheduled twice while other objects are in the course of one single computing. This last point is often misunderstood, as it is possible for different objects to have different frequencies.

[16] named `PsEventListener`.

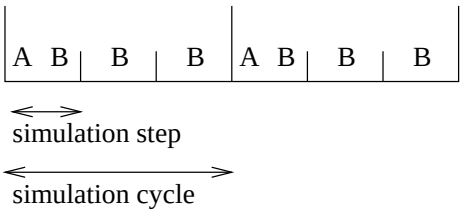

Figure 2.2: Activation cycles with OpenMASK

The controller computes the greatest common divisor (gcd) and the lowest common multiplier (lcm) of all the non zero frequencies. The gcd then becomes the cycle frequency of the scheduler and the lcm the step frequency. Therefore, for the scheduler, there are a certain number of steps in an activation cycle. For an object having a frequency equal to the cycle frequency, it will be scheduled on the first[17] simulation step of each cycle (As case in fig 2.2). Nevertheless, the next simulation step will only start when all objects scheduled in the current simulation step are finished.

This policy implies that the time it takes to compute each simulation step can vary from one step to another. Therefore, even if simulated time (time as seen by simulated objects) flows regularly, from an observers perspective, it will could seem jerky. One should therefore be very cautious when planning multifrequence programs, for the results could be different of those expected.

In a near future, that problem needs to be overcome, because of the need (in haptic feedback interactive simulations for example) for the cooperation of objects whose frequency aren't all of the same order of magnitude. The problem here isn't of a technical nature, but rather of a research problem because the synchronous activation mode was deeply rooted in the conception of the OpenMASK framework.

2.4 Object Management

2.4.1 Base classes for Object Management

The management of simulated objects is done throught a few classes. Each object is known to other objects throught it's *object description*[18], which is static and stored in the simulation tree. An object description is composed of its name, its class, its frequency, its process and its configuration parameters.

The *controller*[19] is responsible for object activation and other functions that cannot be performed at the simulated object level. Therefore the controller performs object creation and destruction, signal handling, and maintains the simulation date.

The controller handles object through an *object handle*[20], which is the data structure the controller uses to store any run-time information it needs for object activation and communications. In distributed version, a *reference object handle* is used for the reference version (the version activated) of the simulated object, and a *miror object handle* is used by controllers handling copies

[17] This is an implementors choice: further versions would need to be able to specify that parameter for each object
[18] named `PsObjectDescription`.
[19] named `PsController`.
[20] named `PsnObjectHandle`.

of the simulated objects if they are needed for inter object communication.

2.4.2 The Simulation Tree

For a given application, simulation object are organised in a *simulation tree* [21]. This simulation tree is an abstract construct, the root of that tree being the object descriptor associated to the controller. Semantic of the simulation tree is left to the application designer.

The two main OpenMASK functionalities are affected by the way object are structured in the simulation tree.

1. object creation, as interpretation of the string representing the created object's class is done, at first at the simulated object's father (in the simulation tree) level, and propagated down to the controller if necessary.

2. search functions. When looking for objects of a precise type, the search can be global, of limited to the descendants, the sons, or the brothers of an object in the simulation tree.

It is important to understand that the simulation tree is different from the geometric graph and from the inheritance tree.

2.4.3 Object Creation

When ?

Simulated objects in OpenMASK can be created either

1. Statically: their description is specified in the simulation tree given to the controller at construction.

2. Dynamically, at the programmer's initiative[22].

3. Dynamically, as needed by other objects. This is true of object created during distributed execution.

In all those cases, the object is sent a MaskStart system event which triggers, at the next simulation step, the successive call of init and compute. For static objects, MaskStart is sent at a date preceding the initial simulation date. For the other methods, MaskStart is sent at the date of the request triggering the dynamic creation.

How ?

In C++, there is no way of creating an object using the string describing its class. Therefore, a specific mechanism is used. All objects, including the controller, can create objects of a certain class, thus creating what we call sub-objects. This is done using an *object creator*[23] which creates new objects when one of its specific member functions[24] is called. A generic specialisation of the object creator[25] creates an object of the type of its template parameter. Therefore, by associating a

[21] named `PsSimulationTree`.
[22] using the controller's `createObject` member function
[23] class `PsSimulatedObjectCreator`
[24] named `createSimulatedObject`
[25] named `PsSimpleSimulatedObjectCreator`

string and an object creator in a map, specific to each simulated object, one is able to create objects using a string describing its class. This creation is performed by the father (in the simulation tree) of the object which will be created, or by the nearest ancestor that is able to create the correct object.

Integration in the activation mechanism

For simulated objects with a simulation frequency, two options are available at creation.

1. The simulation adapts all its data structures so as to respect the given frequency (default behavior). Beware: the fact that the controller's frequency can change may affect a simulated object's behavior. Therefore, the controller fires a signal whenever this happens: `MaskControllerFrequencyChange`.

2. The frequency of the object descriptor is adapted before effective creation. This can be done using a controller's member function[26], which will compute the nearest available frequency to the frequency specified.

2.5 Distribution and Parallelism

In the design of OpenMASK, everything has been prepared for easy distribution. All object interaction take place through the controller or object created by the controller. Therefore, there is no need to redesign simulated objects when replacing the classic controller with a distributed or parallel controller. Indeed, all synchronization and data integrity protections have to be implemented precisely in the objects provided by the kernel. Moreover, the simulated object is an ideal candidate to be used as the unit of code that is distributed. With the number of objects in a simulation being at least an order of magnitude greater than the number of available processors, even the most basic partition algorithm (round robin) produces acceptable results. Therefore, the problem to be solved when considering distribution or parallelization is data coherence and integrity. Data integrity is a problem for the parallel controller, described in the first subsection and data coherence is a problem for the distributed controller, as described in the second subsection.

2.5.1 Parallel controller

The parallel controller (for SMP machines) is a very straightforward adaptation of the classic controller, where objects are scheduled (using a round robin algorithm, that schedules only the direct descendants of the root object of the simulation tree, sub objects being scheduled by the same scheduler than their father object) for activation on one of the few schedulers that run in parallel in their own thread. Synchronization between threads happens at the end of each simulation step, before the controller does any maintenance tasks. Data integrity is ensured with simple mutual exclusion locks associated to each output and event list.

The results achieved by this parallel controller are very dependent on the use of dynamically allocated memory by the simulated objects as this allocation is system wide contention point. In the results presented in table 2.1 and figure 2.4, one fifths of the simulated objects heavily use dynamic allocation. Here, the application used during the tests is an urban simulation, consisting of 30 cars, with complex mechanical simulation for the car physics, and car driver simulation for the drivers behavior and perception. A total of 5 simulated objects are used for each simulated car, and

[26] named `computeAdequateFrequency`

the complete simulation uses 187 simulated objects. Because of the dynamic nature of interactions between car drivers, communication patterns between simulated objects is of a dynamic nature. An illustration of this application is shown in fig 2.3

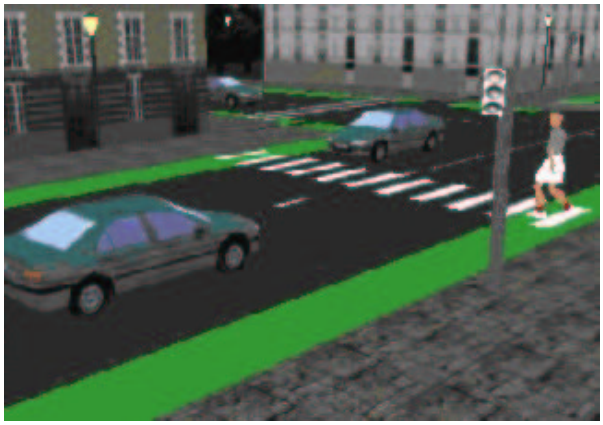

Figure 2.3: A snapshot from the urban simulation

number of threads	execution in s	speedup
1	23,378	1
2	13,387	1,75
3	10,605	2,20
4	8,941	2,61

Table 2.1: results depending on the number of activities, on a 4-way SMP

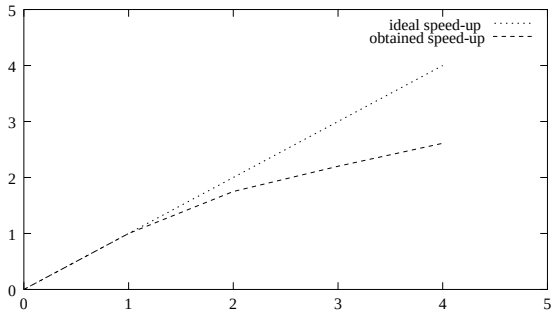

Figure 2.4: Speed-up obtained using the parallel Controller

Recent advances in the parallel controller

A specialized memory management scheme was put in place, and the new results of the parallel controller are now the following, using the same benchmark. Execution time is an average of 5 runs.

number of threads	execution in s	speedup
1	26,211	1
2	14,182	1,85
3	10,473	2,50
4	8,434	3.11

Table 2.2: results depending on the number of threads, on a 4-way SMP

2.5.2 Distributed controller

The principles used by the distributed controller have already been presented in [2, 3, 1] and can be summarized by two points:

1. each time an object needs access to the public data of a distant object (its class, outputs, control parameters, input or event listeners) a local copy called a mirror is created locally. This mirror is then synchronized with the original copy so as to ensured coherence.

2. coherence and synchronization are ensured by an original algorithm that ensures that at a given time, data is been transfered across the network and simulation steps are being calculated. A parameter of the algorithm, called latency, is used to bound the number of simulation steps that can be calculated without having received all synchronization data from a preceding simulation step, thus enforcing constrained relaxed coherence.

Three distributed controllers have been implemented. One that uses PVM, and whose preliminary results are presented thereafter (in table 2.3), and two using a distributed shared memory implemented over a PC cluster. Work is still ongoing to improve the current results

number of nodes	execution time (ms)	speedup
1	91 882	1
2	64 500	1.42
3	51 800	1.77
4	46 000	1.99
5	40 000	2.29

Table 2.3: speed-up obtained using the PVM distributed controller

Chapter 3

A Programming Guide

In this chapter, we present the main functionalities of OpenMASK, structured in logical groups, we try to give an overview of all available functionalities of the kernel.

3.1 Structure of an OpenMASK Application

3.1.1 Relation between components

— inserted block structure here —

3.1.2 Usable elements

1. the simulated object: this is the standard programming level, explained in this chapter.
2. the kernel, by specialization of the controller
3. the duplicated object : reserved for experimented OpenMASK programmers

3.2 Types in OpenMASK

Strictly speaking, types in OpenMASK are objects that implement the interface of packable objects (so that they can be communicated across address spaces), that define an associated polator (to help solve the problem of multi-resolution and of different working frequencies) `virtual PsPolatorNT * createPolator (void) = 0`, that have a default constructor and that can be tested for equality (`operator ==`). Types are user creatable, and for efficiency reasons are implemented through genericity (templates). Therefore, inheriting from an interface isn't necessary as long as it's implemented.

3.2.1 The PsPackable and PsFlowable Interface

The PsPackable interface consists of the following two member functions

```
virtual void unpack (PsIncomingSynchronisationMessage &) = 0 ;
virtual void pack (PsOutgoingSynchronisationMessage &) const = 0 ;
```

The PsFlowable interface is the classic interface implementing the stream operators through member functions. It also implements the PsPackable interface. Therefore, the PsFlowable has the following interface:

```
virtual void extract (istream & = cin) = 0 ;
virtual void insertInStream (ostream & = cout) const = 0 ;

virtual void unpack (PsIncomingSynchronisationMessage &) ;
virtual void pack (PsOutgoingSynchronisationMessage &) const ;
```

And the following operators are defined:

```
istream & operator >> (istream &, PsFlowable &) ;
ostream & operator << (ostream &, const PsFlowable &) ;
```

3.2.2 Base Types

The following types are defined by the OpenMASK kernel:

1. defined in `PsNumericType.h` PsInt, PsDouble, PsFloat, PsLong
2. defined in `PsString.h` PsString. Warning. For PsString, `insertInStream` is geared towards human readable strings, meaning the number of characters isn't printed. Therefore, extract will not extract strings with whitespaces in them. On the contrary, pack and unpack use the number of characters. This has an impact for any user type that aggregates a PsString. If pack and unpack aren't defined for that new type, and whitespace are used in one of the aggregated strings, error will happen.
3. defined in `PsName.h` PsName. An identifier, using a name server. Enables fast comparisons between names and compact representation of the names used.
4. specific types, defined in `PsNumericType.h` : PsDate, PsFrequency
5. Standard library types defined in `PsList.h, PsMap.h, PsPair.h` PsList, PsMap, PsPair

3.3 The simulated object

3.3.1 Interface

The interface of a simulated object is composed of the following elements

1. an unique name, a class name (generally a bijection of the existing C^{++} classes.
2. a list of outputs, inputs, control parameters and event listeners
3. a list of configuration parameters
4. a list of scheduling parameter

3.3.2 Implementation

Communicating with Other Simulated Objects

1. by updating the public attributes (see 3.3.3 of the object (outputs and control parameters)
2. by firing signals in the environment.
3. by sending events to other objects

Being activated by the controller

1. when events are received
   (a) at the objects frequency
   (b) at the controllers frequency
   (c) when event are received by event listeners
2. at a regular frequency

Using functionalities provided by the controller

1. to find other objects in the simulation tree by their name or type to gain access to their interface.
2. to create, destroy or change the frequency of other objects
3. to register interest in signals fired in the environment or by specific objects

3.3.3 Attributes of an Object

Public attributes of an object are defined as identified and typed (see 3.2, page 15) public data of an object. They come in two flavors:

1. Outputs, which can only be read by other objects
2. Control Parameters, which are outputs for which new values can be proposed

They are added by the following member function of the PsSimulatedObject class.

```
addOutput<PsInt>("OutputName") ;
addOutput<PsInt>("OutputName",pointerToPolator) ;
addControlParameter<PsFloat>("controlParam") ;
addControlParameter<PsFloat>("controlParam",pPolator);
```

These member functions return a reference to the created attribute, or to the attribute of the same name and type already created. If an attribute of same name but of different type is already created, a PsUnallowedOverloadingException is thrown.

After the attribute has been created, a pointer to the created attribute can be found using

```
getPointerToOutputNamed("OutputName");
getPointerToControlParameterNamed("ControlParameterName");
```

To change the value of a public attribute, two paths are available:

1. `attribute.set (newValue) ;`, which will copy the value in the history Fifo.
2. ```
   Type & value = attribute.getNextPlaceHolder();
   value.change();
   attribute.setInPlace() ;
   ```
   , which gives access to the memory space of the Fifo.

Information on attributes can be obtained with the following member functions.
   ```

- const PsName & getName() ;
- const PsSimulatedObject & getOwner() ;
- void empty() ;
- const Type & getLastExactValue() ;

Here, `empty()` empties the history Fifo and `getLastExactValue()` bypasses the polator to get the last value produced for the attribute.

Reading a public attribute

The control parameter is only an interface, and concrete control parameters are an instance of `PsGenericControlParameter`,whose inheritance diagram is described in figure 3.1. Therefore, all classic operations on outputs are possible on control parameters.

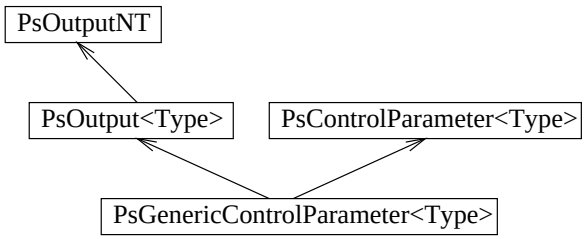

Figure 3.1: Inheritance diagram for PsGenericControlParamer

Inputs

Therefore, one can understand inputs as a protected[1] service for establishing data connections between two objects. Inputs come in two flavors: *private inputs*, whose connection to a output of another object can only be established by the owner of the input, and *public inputs*, whose connection to an output can be requested by the owner of the output, or by the controller of the owner of the input.

Different types of inputs can be used, and these different types are related by inheritance as described in the following diagram (fig 3.2

The standard input is of type `PsInput<Type>`, and defines the two classic member functions (get and getLastExectValue). The sensible input add the `hasChanged(bool keepChanged)` member function, which returns true is the value of the connected output has changed. The sensible notifying input sends an event (of signature `getAssociatedEventId()`) to it's owner whenever the value of the connected output has changed.

Public inputs only differ in implementation from private outputs by a boolean given at construction. Nevertheless, public input are considered as part of the interface of a simulated object, and should therefore all be created by the constructor of that object. When a connection request is made on a public input, an event (of signature `getConnectionEventId()`) is sent to the simulated object owner of the input, and is processed by the default eventListener (of class

[1]As a protected service, there is a cost associated to the establishment of a connection between an input and an output. In particular, mutual exclusion is needed on some of the data structures of the connected output, and therefore rapid connections-disconnection cycles can become bottlenecks when using a multi-threaded scheduler. Inputs might not be needed in those cases.

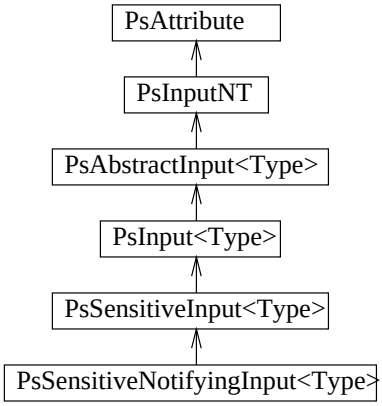

Figure 3.2: Inheritance diagram for the different types of inputs

`PsInputConnectionEventListener<Type>`) associated to that public input which then connects the input[2].

All inputs implement the following member functions :

1. general member functions:
   - getConnectedOutput()
   - getName()
   - getOwner()
2. member functions for input connection:
   - bool connect (const PsName & objectName, const PsName & outputName);
   - bool connectToControlParameter (const PsName & objectName, const PsName & outputName);
   - bool connect (PsSimulatedObject & object, const PsName & outputName);
   - bool connectToControlParameter (PsSimulatedObject & object, const PsName & outputName);
   - bool connect (PsSimulatedObject * pointerToObject, const PsName & outputName);
   - bool connectToControlParameter (PsSimulatedObject * pointerToObject, const PsName & outputName);
3. member function for input connection with initial value:
   - bool connect (const PsName & objectName, const PsName & outputName, const Type & initialValue) ;
   - bool connectToControlParameter (const PsName & objectName, const PsName & outputName, const Type & initialValue) ;
   - bool connect (PsSimulatedObject & object, const PsName & outputName, const Type & initialValue) ;
   - bool connectToControlParameter (PsSimulatedObject & object, const PsName & outputName, const Type & initialValue);

[2]At the time of writting (2002/05/04, OpenMASK3.0-RC1), only outputs of the output table can connect to public inputs

- bool connect (PsSimulatedObject * pointerToObject, const PsName & outputName, const Type & initialValue);
- bool connectToControlParameter (PsSimulatedObject * pointerToObject, const PsName & outputName, const Type & initialValue);

4. member function for input disconnection:
    - diconnect();

Adding an input to a simulated object is done using one of the following methods. Input are private by default, and ask for `PsPolatorNT::defaultPrecisionLevel`

```
addInput<Type>("name") ;
addInput<Type>("name", true); (public input) ;
addInput<Type>("name", false); (private input) ;
addInput<Type>("name", false, PsPolatorNT::Cubic) ;
addInput<Type>("name", false, 42) ;
```

Reading a public attribute is classically done using one of the two following member functions of *inputs*.

`get(PsDate offset)` reads the output through it's polator (see 3.3.4, page 21), at an optional offset from the current simulated date given by the optional parameter (whose default value is 0). The value returned by this member function is an estimate (whose precision depends of the polator of the read attribute and the precision level requested on that polator) of the value of that attribute at a date d, where $d = currentSimulatedDate - offset$.

`getDistanceToExactValue()` return the offset between the date asked for by the last call to get and the date of the nearest exact value used to compute the result of `get()`

`getLastExactValue()` reads the last exact value set in the output.

`getDateOfLastExactValue()` returns the date of the last value set in the output

Reading an input can throw exceptions:

`PsInvalidOutputException` if the input isn't connected to a valid output. This can happen if the output the input was connected to was destroyed, or if no connections have been established

`PsUnInitialisedOutputException` if no value has been set in the connected output.

Conclusion

Public attributes of an object are

1. outputs and control parameters, which are public attribute, read either directly or using inputs, either directly or through their associated polator.
2. public inputs, defining the precision level needed when reading an output through its polator, and that can be connected either by their owner or by the owner of the connected output.

3.3.4 Polators

To each public attribute of an object is associated to an history circular FIFO. The length of this FIFO is calculated by the current controller , but can be overridden by an environment variable named MaskHISTORYLENGTH.

This history FIFO is read through an object called polator which adapts the returned value to the date at which it is requested, independently of the production date of the values in the FIFO. In version OpenMASK3.0, the polator cannot be changed at run-time.

Each output defines the polator uses for that output. For each input, a precision level asked to the output polator is defined. The default values are respectively the default polator for the type of the output and the default precision level defined in the abstract polator. This default value is a public static member of the abstract polator class `PsPolatorNT::defaultPrecisionLevel`.

Abstract Polator

The abstract polator class `PsPolatorNT` also defines 4 default precision levels:

```
enum PsPolatorNT::PsPolationLevels { Constant, Linear, Quadratic, Cubic  }
```

as well as a constructor with a parameter which is the number of value needed in the history Fifo for the highest polation level defined for the polator. This number of values is queried by the controller when the length of the Fifo is calculated by the controller.

Concrete Polators

The first usable polators define 3 member functions called by the kernel to perform polation:

- virtual const Type & interpolate (..)
- virtual const Type & antepolate (..)
- virtual const Type & extrapolate (..)

Their precise parameter list, including the precision level requested by the input can be found in the file PsPolator.h

Numeric polation functions

The following functions are defined as static functions of the `PsNumericType` class.

- linearInterpolate
- quadraticInterpolate
- cubicInterpolate
- linearExtrapolate
- quadraticExtrapolate
- cubicExtrapolate

3.3.5 Class of a simulated object

The classes defined in the implementation language (C^{++}) aren't usable to create an object of the correct class using it's name. Therefore, a mechanism specific to OpenMASK is provided by the execution kernel, which associates a class name (of type `PsName`) to an object creator (an

instance of `PsSimulatedObjectCreator`). That name can be different from the name used in the implementation language, but for evident reason, it is usually the same.

To each object, a especially to the controller, is associated a map associating a class name and a corresponding object creator. When a son (in the simulation tree) of an object must be created, the execution kernel of OpenMASK looks in the map to find an object creator for the class of the object to be created. If one is found, the son is created using it, otherwise, the creation request is passed to the father of the object who has received the creation request.

A simulated object creator is an object that implements the

- createSimulatedObject(PsController & controller, PsnObjectDescriptor & objectDescription)

member function, as define by the abstract class `PsSimulatedObjectCreator`. A generic implementation of this abstract class is defined as `PsSimpleSimulatedObjectCreator<Type>` , and a call to the `createSimulatedObject` member function of this generic class creates an object of type `Type` (which should be a derived class of `PsSimulatedObject`) by calling the standard constructor. Of course, other simulated object creator can be implemented, in particular if additional parameters need to be passed to the constructor of the object being created.

To add an simulated object creator to an object , two member function are available:

```
addInstanceCreator("derivedClass",pointerToObjetCreator) ;
generateAndAddInstanceCreator<Type>("derivedClass") ;
```

The second method presented here adds an instance of the generic simulated object creator, whereas the first is the most general version. Once one of these member functions has been called, `derivedClass` is considered as a sub-type of the calling object.

3.3.6 Configuration and Scheduling Parameters

Configuration and scheduling parameters are described before the creation of a simulated object. Configuration parameters are used by the object to parameter its state and behavior, and scheduling parameters are used by the controller to schedule the object.

All those parameters are described by a `PsConfigurationParameterDescriptor`, which structure configuration parameters in a hierarchy, and enables access to sub parameters either by their name, or by their position in the sub-hierarchy. Parameters are stored as strings, and therefore it is always possible to get an string associated to a configuration parameter descriptor. If the configuration parameter is an `PsUniqueConfigurationParameter`, the result is the expected one (the string describing that parameter), if it is a `PsMultipleConfigurationParameter`, a string describing a the sub-hierarchy will be computed and returned.

Here are a few code samples of access to respectively the first parameter, and the parameter named initialPos.

```
getConfigurationParameters().getNumberOfSubItems() ;
  if (n > 0)
    {
       getConfigurationParameters().
         getSubDescriptorByPosition (0)->getAssociatedString() ;
    }
```

```
assert (getConfigurationParameters().getSubDescriptorNamed("initialPos") != NULL) ;
getConfigurationParameters().
  getSubDescriptorNamed("initialPos")->getAssociatedString() ;
```

3.3.7 Conclusion

The interface of a simulated object is composed of:

1. a unique name, a class (a name generally corresponding to the C^{++} name of that class
2. a list of outputs, control parameters, public inputs and event listeners (see 3.4.4, 24)
3. a list of sub-type the object can create
4. configuration parameters
5. scheduling parameters

3.4 Signals and events

Signals and events are used for discrete communication, where the communication is meaningful only at the time of the communication. These signals and events are fired when the sending object is activated, either for events processing or during usual activation.

Signals are emitted into the virtual world, whereas events are send to specified objects. Once emitted into the environment, signals are sent to interested simulated objects in the form of events.

3.4.1 Base concepts

To each event or signal is associated a `PsEventIdentifier`, which is a `PsName` used to tag the event or the signal and that defines what we call the signature of the event or signal. Data can be associated to signals or events, transforming them in valued signals or valued events `PsValuedEvent<Type>`.

It should be noted that event are implemented in such a way that derived type are preserved (due to the use of pointers), even in a distributed execution, providing a few extra steps are taken when a new type of event (derived from `PsEvent` is created. Please see 3.4.5 page 25 for more details.

3.4.2 Firing an event or a signal

the simulated object class has a few member functions to enable an object to fire signals and to send events:

- `sendEvent ()`
- `sendValuedEvent <Type> ()`
- `fireSignal ()`
- `fireValuedSignal <Type> ()`

These member functions can be used in the following ways :

- sendEvent("receiver", "change state") ;

- fireValuedSignal<PsInt>("newSwitchValue", 4);

3.4.3 Receiving signals

To be notified that a particular signal has been fired in the environment, an object has to register, either with the controller if it wants to be notified of all signals with a certain signature or with the object he wants to be notified only of signals fired by certain objects. The signature of the event sent is by default the signature of the fired signal, but an alternative signature can be specified at registration to avoid name conflicts.

Registration methods

- for signals fired in the environment `registerForSignal ()`

- for signals fiered by a particular object `registerForSignalBy ()`

- `cancelRegistrationForSignal ()`

- `cancelRegistrationForSignalBy ()`

3.4.4 Event processing

The events received by an object are stored by the controller in the object handle associated to that object. If an inactive object receives an event, it will be scheduled for event processing by the controller, at the processing speed of the controller. If the object is active, event processing will take place just before the call to the `compute` member function of that object, unless the `processEventsASAP` of the object returns true. Then, the object is scheduled for event processing as soon as possible by the controller. It should be noted that event processing can only happen once at a given simulation date.

Events are stored by their object handle, oldest received first. The older relationship between two events is defined as the older relationship between two simulation dates, and if not sufficient the oldest received. In a non distributed simulation, this order implies that events are received in the order they are send, whereas in a distributed simulation, no such guaranty is made.

Event listeners

The way an object reacts to specific events can be considered as being part of the interface of an object, especially in the case of standard communication protocols between objects. To guaranty that a reaction is preserved through inheritance, event reaction code can be encapsulated in objects called `PsEventListeners`, created at simulated object construction (use the `addEventListener` member function). This also enables that creation of code sharing for event reaction.

Event listeners register to events they are interested in by one of the following two methods

1. by calling `_owner.registerEventListenerForEvent ( *this,"eventId") ;` as much as needed in their `registerEvents` member functions.

2. by being registered by their owner object using the same member function

Event processing algorithm

For each event received by an object handle, find the event listener the most recently created for events of that signature. The `processEvent` member function of that event listener will then be called with the examined event. If the return value is `true`, the event is considered processed and destroyed. Otherwise, the next event listener for events of that signature is called.

If no event listeners are found, or if the last event listener returns `false` after having processed the event, the `processEvent` member function of the simulated object is called. If it returns `true` (the default behaviour), the event is then deleted, other wise, responsibility of destruction is passed to the object.

Before any processEvent member function is called on an object or an event listener, the `prepareEventProcessing` member function of that object is called. At the end of an event processing step at a particular simulation date, the `eventsProcessed` member function is called.

3.4.5 Distributed events

To be able to distribute events across a network, these have to be encoded in synchronization streams. Therefore, to enable use of derived types of `PsEvent`, one has to encode events in a generic fashion and enable creation if the correct derived type during decoding. Therefore, when events are encoded, the name of the derived type is encoded, then the data associated to the `PsEvent` class is encoded, and finally additional data for the derived type is encoded.

Therefore, decoding takes place in two steps:

1. the name of the derived type, and all information common to events is decoded, and an object of the correct type is created using that information
2. the created event is asked to decode any additional information present in the synchronization stream.

This algorithm has 2 drawbacks:

1. Derived types of `PsEvent` must have a constructor with the same signature as the base constructor for events
2. A distributed infrastructure has to be used so that it is possible to create an event using the name of the class of that event. It is this last drawback which is the most difficult to guaranty, and therefore is the focus of the rest of this subsection.

This second drawback is circumvented in the current version of OpenMASK in the following way. To each derived class of PsEvent must be associated a class that derives from `PsEventCreator` that overloads the `createRealEvent` member function so that it created the correct derived event. This EventCreator must be then associated to the name of the new derived class of PsEvent being created. For valued events, this is done by defining a protected inner class of `PsValuedEvent` < `Type` > derived of `PsEventCreator`, and a static data member of this class in the `PsValuedEvent` < `Type` > class. By construction of `PsEventCreator`, the construction of this static data member will register the new event creator with the OpenMASK kernel with the name specified when implementing the constructor of this new event creator.

The fundamental problem lies with the choice of the name chosen. Indeed, the computer generated name (using typeid().name()) is implementation defined (different name mangling scheme, and different reaction to typedefs). Therefore, when more than one compiler is involved in producing

the code used in a distributed simulation problems can arise. Therefore, in the event a encoded name is not recognized, the simulation is stopped and the name of the unrecognized types is printed as well as all the recognized types. The main should then be recompiled after having added a call to `PsEventCreator::declareAlias` with the unrecognized name and the corresponding recognized name[3].

3.5 Using the controller

There only exist one controller by address space of a program, therefore, in the most classical cases, there is only one controller by simulation. The controller is needed for all global management, implementing signal dispatching, object creation and destruction, object scheduling, access to the simulation tree and to a description of all objects in the system.

3.5.1 Object descriptors

In theory, an object is completely described for the other object by its object descriptor. Each object can query its descriptor by the `getObjectDescriptor()` member function, can have access to the descriptor of its father in the simulation tree with `getFathersDescriptor()`, and to a descriptor of an any other simulated object in the system through the `getObjectDesciptorOfObject()` member function of the controller.

In practice, most of the interface of an object is inspectable by using member functions of the `PsSimulatedObject` class. One can get access to a pointer of (a copy of) an object by calling the `getPointerToSimulatedNamed` member function of the controller. Then, the following member functions should then return the expected results:

- getOutputTable()
- getControlParameterTable()
- getInputTable()
- getEventListeners()
- getEncapsulatedClassesTable ()

3.5.2 The simulation tree

The simulation tree is a hierarchical structure that can be used by all who want to structure the simulated objects they use for a simulation. The nodes of this tree are the object descriptors used by the kernel, and the tree is used for object creation to interpret the class name given to new objects. For the rest, the semantic associated to the structuration is left to the imagination of the one programming an application.

The simulation can then be used to specialize searches of objects, according to their class for example. To start the search at the root of the simulation, the following member functions can be called upon the controller:

- listSonsOfType<Type> (PsName)
- listDescendantsOfType<Type>(PsName)

[3]It should be noted that the current implementation of that system supposes than one of the two names declared as aliases has already been added to the data structure containing the correspondence between a name and an event creator. This implies that event creator are instantiated as static objects

and for requests starting at the current level of the simulation tree, the current level being the node describing the object on which the following member functions are called.

- listSonsOfType<Type> ()
- listBrothersOfType<Type> ()
- listDescendantsOfType<Type>()

3.5.3 System signals

The following system signal are sent in the environment.

- `PsSystemEventIdentifier::MaskObjectCreated`, when an object is created
- `PsSystemEventIdentifier::MaskObjectDestroyed`, when an object is destroyed
- `PsSystemEventIdentifier::MaskControllerFrequencyChange`, when the creation of an object has required changing the frequency of the controller.

3.5.4 Dynamic object creation

To create objects not described in the initial simulation passed as argument to the controller upon creation, one has to use the following controllers member function:

```
createObject(const PsObjectDescriptor & newObjectDescription,
             const PsName & fathersName )
```

Effective creation only happens at the following simulation step of the controller, once the scheduling data structures of the controller have been adapted to integrate the new object at the specified frequency. In order to avoid this adaptation, the `computeAdequateFrequency` member function can be called to compute a frequency adapted to the controllers existing data structures. Nevertheless, events can still be sent to uncreated object as well as registration to signals. Figure 3.3 shows the possible states of a simulated object.

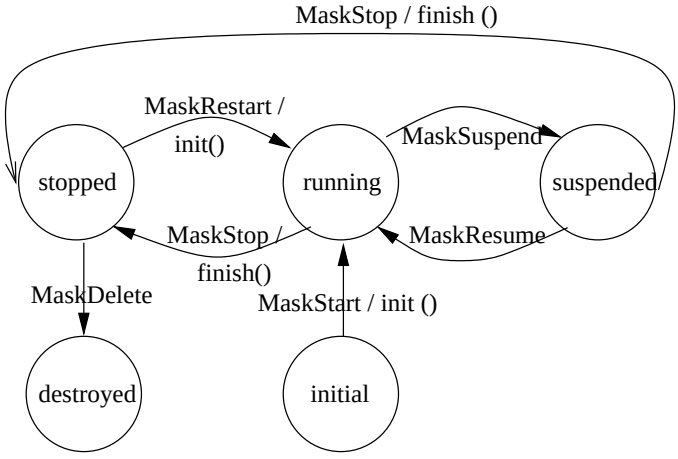

Figure 3.3: The possible states of a simulated object

3.6 Conclusion

This programming guide needs to be extended to include documentation on distribution, duplicated objects.

Chapter 4

OpenMASKs kernel programming guide

4.1 The name server

For efficiency reasons when comparing or copying names, all objects of class `PsName` only store an id wich is obtained throught a name server, which stores all correspondances between ids and the corresponding human readable strings.

The problems with implementing a name server are the following

1. The name server must be started before any PsName is constructed. PsNames could be declared static by users of the OpenMASK lib, and therefore the use of a complete default name server is mandatory.

2. The optimal name server depends on the execution context. Indeed, locks are needed for multi-threaded executions, a central name server is needed for distributed execution and small memory footprint name server is needed for memory contrained applications (a name server that is able to reuse ids when one is sure that id is no longer used by any PsName).

The solution adopted in the OpenMASK kernel is the following : a very capable general name server serves as the default name server for all applications. This name server does reference counting, mutual exclusion and tracks the location of all PsNames created in the system.

When a controller is created, it may replace that default name server by a name server optimized for that controller, in a non reversible way. The new name server only needs to implement the functions needed for the created execution kernel, and could forget all unecessary information that was present in the original name server. Nevertheless, the original name server is not modified, and could be restored by the application programmer if the same application runs different controllers in a sequential fashion.

4.2 Memory management in the OpenMASK kernel

On an SMP machine, calls to the standard memory management functions for dynamically allocated memory is a classic bottleneck of multi-threaded programs. The OpenMASK multi-threaded schedulers therefore implement a very basic memory manger associated to each thread, so that calls to the system memory management functions are far and in between.

4.2.1 Taking over memory management

Memory managers

The principles of this memory management are very simple. First, the OpenMASK kernel takes control of all[1] memory management in the virtual address space, by redefining the global `new` and `delete` operators, as authorized by the ISO standard.

The replacement operator `new` is implemented by an instance of a subclass of `PsnMemoryManager`, which defines the `allocateSizeRemembered` member function.

The memory managers stack

In order to be able to define specific allocation contexts, the memory manager used for allocation is the one referenced by the top of the stack of memory managers for the current thread. Therefore, each thread can use it's one memory manager, and the default memory manager for a thread can be temporarly overrridden if a specific allocation context is needed.

An implicit stack implemented using the `HeapStackTop` class is used to implement this stack in an easy to use way. The constructor of a `HeapStackTop` takes a pointer to a memory manager as only parameter, and installs that memory manager as the top of the stack of memory managers for the calling thread. It also keeps the old top memory manager as a data member, so that when the object is destroyed, the old memory manager can be reinstalled as the top memory manager of the calling thread[2].

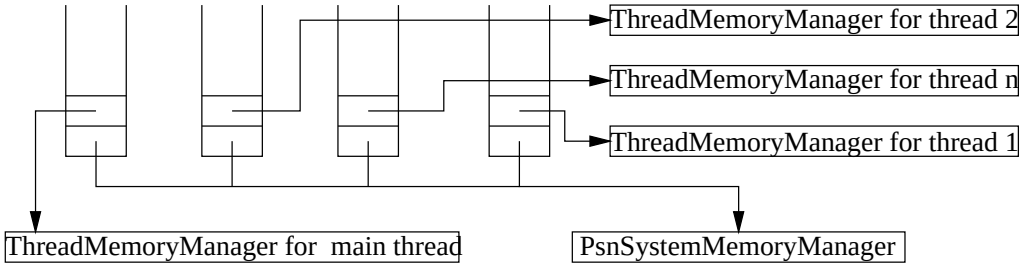

Figure 4.1: The memory managers stack

The default memory manager

Upon creation of a process linked with the OpenMASK library, a default memory manager is installed for the main thread. This default memory manager is called the system memory manager, of class `PsnSystemMemoryManager`, and does nothing except calling `malloc` and `free` when it is used for allocations or deallocations of memory. That system memory manager is always accessible through the static function of the `HeapStackTop` class called `getSystemMemoryManager`.

[1] Provided all allocations, especially for standard library allocators, are implemented whithout directly interfacing with the system provided memory allocation functions. This isn't allways the case, especially for standard library implementation associated to versions of gcc < 3.0

[2] which should be the thread that created the `HeapStackTop` object in the first place

Freeing memory

Whereas memory is allways allocated by the memory manager at the top of the implicit stack of memory managers for a thread, memory should be freed by the memory manager that had allocated it. Therefore, the difficult task is to find the memory manager that has allocated a memory block being freed. To implement this, the `PsnMemoryManager` class keeps a list of all constructed memory managers, and all memory managers are able to test if a given adress was allocated by them. This is true of all memory managers except the system memory manager.

Therefore, after creating a memory manager, it's init member function should be called so that it is inserted in the list of memory managers. The `whichMemoryManager` static function of the memory manager class then goes throught that list until it finds the memory manager that has allocated the memory block being freed. If none is found, the system memory manager is supposed to have had allocated the memory block.

Nevertheless, in order to allow for performant allocation-deallocation cycles, the global `delete` operator allways calls the `freeSizeRemembered` member function of the current memory manager (the one on top of the memory manager stack for this thread), which will perform deallocation. Therefore, standard implementation of the `freeSizeRemembered` member function is to free the momory block if it was allocated by the called memory manager, and to handle it to the memory manager found by the `whichMemoryManager` class function otherwise.

4.2.2 The thread aware memory managers

When using the multitheaded schedulers, each created thread starts execution by installing the system memory manager at the top of the implicit memory manager's stack, and then by installing a memory manager that is thread aware. This memory manager, of class `PsnThreadMemoryManager` is supposed to be efficient when used in multi-threaded contexts. This efficiency is obtained by

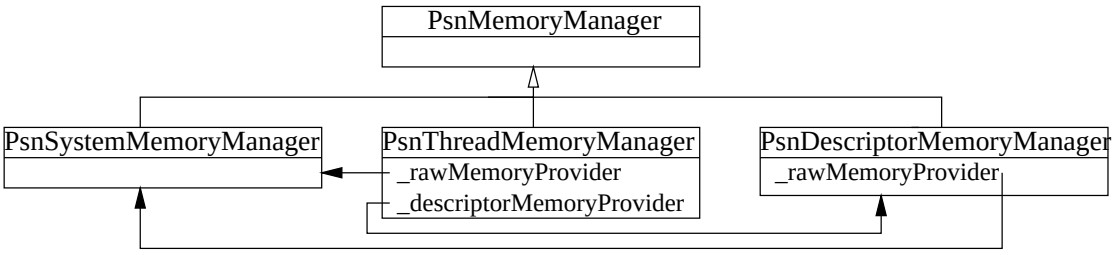

Figure 4.2: Inheritance relations between the memory managers

1. using preallocation of large memory blocks obtained throught a helper memory manager (in general the system memory manager). Therefore, allocation can be done mostly without contention.

2. keeping track of the available memory blocks in data structures whose memory needs are fullfilled without contention. This is done by allocating and freeing the element of the list of free memory blocks using a dedicated memory manager implemented in a way that mostly avoids memory contention (the `PsnDescriptorMemoryManager`)

3. implementing `freeSizeRemembered` without in a way that garanties that no two execution threads will coexist in objects of class `PsnThreadMemoryManager`, thus avoiding a contention

point. This is done by delaying any calls to `freeSizeRemembered` that weren't made by the thread using the memory manager by maintaining a global FIFO of unfreed memory blocks. This FIFO is emptied by the main thread when is garantied that all threads are blocked on a synchronisation barrier.

4.3 Distributed Kernels: Miror Creation

When using a distributed kernel, simulated objects are assigned to one of the nodes of the distributed simulation. Whenever these objects are used on an other node, a miror of the reference object is created. Whatever the reason, this miror creation allways ghappens after a call, either by the kernel or by an user-space simulated object of the `getPointerToSimulatedObject` member function of the controller. The correct implementation of miror creat the ion is to construct the object as described by the object descriptor, having therefore only the constructor of the object called, and the to register the miror to the reference object. Once this construction is done, the outputs and control paramters of the object are either uninitialised or initialised with out of date values. Therefore, the first get on one of these outputs should be a blocking call, waiting fot correct values to arrive from the referential, if none have yet arrived after the registration. But correct implementation of this isn't trivial[3], and therefore, miror creation is made a blocking call that waits for initial values comming from the referential.

[3]Performer(2.5) seems to overrides the throw call, and it would mean, if implemented as in the Mome controller, deriving a PsDistributedOutput

Appendix A

Tricks, tips and FAQs

A.1 Compiling with OpenMASK

The header file `OpenMASK.h` defines three Macros:

`OpenMASK` , with no value,

`OpenMASKDATEOFVERSION` , whose value is the date at which the used version was last checked out

`OpenMASKVERSION` , whose value is set by hand at each major release, and whose value is the number of the version used

Appendix B

The C^{++} used with OpenMASK

B.1 The black magic of constructors

A simulation object written for OpenMASK inherits of the `PsSimulatedObject` class and therefore allways has a contructor with the following lines

```
mySimulationObject ( PsController & ctrl,
                     const PsObjectDescriptor & description) :
   PsSimulatedObject(ctrl,description)
   {
   }
```

The reason for this construct is that an simulated object in OpenMASK is created with its description and a reference to the simulations controller so that all member function of an object can be called in the construcutor whithout producing bugs. When implementing an object constructor in C^{++}, code fragments can be written between the argument list and the opening { of the constructors code. The code written there is called an initialisation list, because it can be used to initialise any class member. In particular, initialisation of the ancestors is mandatory in the initialisation list (therefore, in this example, the call to `PsSimulatedObject(ctrl,description)`), as well as initialisation of any member which is of reference to type type.

For example, suppose the class `mySimulationObject` has a member declared as `PsOutput<PsInt> & _myInput`. Then, the contructor would be something like :

```
mySimulationObject ( PsController & ctrl,
                     const PsObjectDescriptor & description) :
   PsSimulatedObject(ctrl,description),
   _myinput ( addInput<PsInt>("nameOfInput") )
   {
   }
```

B.2 The `const` modifier

In C^{++}, modifiers can be added to declarations. In the examples above, `const` is an example of such a modifier. Its meaning if relatively straightforward. Objects which are declared `const` cannot be modified by the code fragment that knows than as const objects. This means that only member function declared with the const modifier (after the list of parameters) can be called on that object.

For example, in the constructor of a `PsSimulatedObject`, modification of the `PsObjectDescriptor` parameter of the constructor is not possible.

B.3 Template member functions

In the OpenMASK kernel, genericity in the form of templates is heavely used. In paticular, template member function are used for input and output creation. Declared as

```
template <typename Type>
PsInput<Type> & addInput( const PsName & inputName,
                          bool makeConnectable = false,
                          int polationLevel = PsPolatorNT::defaultPrecisionLevel ) ;
```

they can be called very simply with `addInput<PsInt>("name")` when they are called on the current object (implicit `this`). When they are called using `.` or `->`, the template keyword must be inserted as in the following example: `getSimulatedObject().template addInput<PsInt>("name")`.

B.4 Exceptions

Correct exception handling is delicate, and is beyond the scope of this general introduction. In OpenMASK, all thrown exception are derived types of `PsException`, including `PsUserException`. Therefore, catching an exception generated by OpenMASK can be done the following way:

```
try
  {
    code_that_can_generate_exceptions() ;
  }
catch (PsException & e)
  {
    if (can_treat_exception (e) )
     {
       treat_exception (e) ;
     }
    else
     {
       //propagate the exception
       throw e ;
     }
  }
```

When a fatal error is generated by a simulated object, the recommended way to deal with the problem is to throw an exception. The following code fragments illustrated first a simple case, then a more compilated one.

```
  if (fatal_error() ) throw PsException("Explanation of fatal error") ;

  if (fatal_error() )
    {
```

```
    PsException e ;
    e<<"Object named "<<getName()<<" generated a fatal exception \n";
    throw e ;
}
```

If exceptions are thrown and not catch, they will cause the program to abort. Therefore, it is good practice to encapsulate the whole main with a try catch that will print the message associated to the exception before exiting.

B.5 Casts

In C, when type conversions are needed, the explicit cast operator is used, as in this example :

```
mySimulationObject * obj ;
obj = (mySimulationObject *) getPointerToSimulatedObjectNamed (getName()) ;
```

In C^{++}, this operation is very dangerous and should never be used, even if it's correct. Indeed, no verification on the validity of the operation is perfromed, and more importantly it doesn't allways performed the expected operation. This is in particular true for explicit cast operformed on classes that inherit from more than one other class at one point in the inheritance graph. Here, depending on the compilers implementation, the results are unpredicatable and may lead to unexpected and difficult to debug run-time crashes.

In C^{++}, three operators are available for type conversion:

1. reinterpret_cast, which should only be usefull when dealing with basic type to basic type conversions (in particular pointer to int style conversions)

2. const_cast, which should never be used unless fully aware of the consequences on the programs semantic. It is in particular usefull when `char *` have to be passed as parameters to C libraries and that only `const char *` are availble. This can be legitimate because `const` is known by C

3. dynamic_cast, which is the most usefull because it enables type conversions from a class to a derived class. If the conversion fails (a run-time check is performed by that operator), the result is NULL or a bad_cast exception depending if the operand are of pointer or reference type respectivel. Dynamic_casts are only possible with polymorphic types, i.e classes with at least one virtual member function (therefore, it is _very_ good practice to allways write a destructor, and declare it virtual).

Therefore, the example should be written

```
mySimulationObject * obj ;
obj = dynamic_cast<mySimulationObject *> getPointerToSimulatedObjectNamed (getName()) ;
//the cast should not fail: use an assertion during development
//compile with -DNDEBUG to avoid the test in production use
assert (obj != NULL) ;
```

B.6 Underscores (_)

A naming convention used in the OpenMASK kernel and in the examples is to name any protected data member with a name starting with _ We believe this improves code readability, and this is the reason that naming convention was used.

Appendix C

The OpenMASK3 loader

The use of a loader to load the initial simulation tree is a natural idea. Therefore, two loaders are provided with the OpenMASK distribution. The first is a loader compatible with the file format used in the latest version of Gasp. The second is a richer format, using a VRML-like syntax. This can be confusing, as the file format is designed to describe a simulation tree and not a geometric scene graph. This is the source of much confusion, so please take time to think about the differences.

In the appendix, we rapidly present an overview of the syntax of the new simulation loader that can be found under $(OpenMASKDIR)/contrib/loaders/simpleOpenMask3Loader/ and that is available in the libSimpleOpenMASK3Loader.so library under the name SimpleOpenMASK3Loader.

C.1 Overview

The recognized grammar is only slightly more complex as the native grammar available in the kernel for object descriptors. The additional recognized key words are the following:

include to include a file described as a quoted string as in

```
include "descriptions.OpenMASK3''
```

Define to give a name to a part of the file as in

```
Define StandardParams UserParams {
   initialPosition [ 1.2 4.2 3.0]
   initialSpeed 3.6
   complexparam {
      name view1
      size [1280 768]
  }
}
```

Use to use a previously defined part of the file as in

```
Use StandardParams
```

DefUse to define and use a sub part of the file at the same time.

C.2 Semantic of instructions

C.2.1 Construction of the simulation tree

The parsed files are read upon construction of the loader, and a simulation tree is obtained through the `getRootObjectDescriptor` member function. The loader assumes that if it only finds one object at the root of the file, the root of the file describes the controller.

C.2.2 Overloading of parameters

With the SimpleOpanMASK3Loader, it is possible to overload parameter. Considering the preceding declarations, the following

```
myObject {
  Class "ObjectClass"
  Scheduling {
    Frequency 42
  }
  Use StandardParams
  UserParams {
    initialSpeed 0
  }
}
```

will be loaded as:

```
myObject {
  Class ObjectClass
  Scheduling {
    Frequency 42
  }
  UserParams {
   initialPosition [ 1.2 4.2 3.0]
   initialSpeed 0
   complexparam {
      name view1
      size [1280 768]
  }
}
```

Thus overloading the initialSpeed parameter. The overloading follows the semantic of the `appendSubDescriptorsOf` member function of the `PsMultipleConfigurationParameter` class.

C.3 Recognized grammar

sons ::= (
ID "{" objectDescription "}"
| "include" string

```
                              | definition
                              | defOrUse
                              )+ ;
objectDescription         ::= partialObjectDescription ;
partialObjectDescription  ::= (
                              key ID
                              | key "{" listnamedparam "}"
                              | definition
                              | defOrUse
                              | "include" string
                              )+ ;
listnamedparam            ::= ( namedparam )+ ;
listparam                 ::= ( param )+ ;
namedparam                ::= key param
                              | "include" string
                              | definition
                              | defOrUse ;
param                     ::= "{" listnamedparam "}" | "[" listparam "]" | paramValue ;
paramValue                ::= ( entier | flottant | string | ID ) ;
definition                ::= "Define" ID ID param ;
defOrUse                  ::= defuse | use ;
defuse                    ::= "DefUse" ID key param ;
use                       ::= "Use" ID ;
key                       ::= ID ;
entier                    ::= ENTIER ;
flottant                  ::= FLOAT ;
string                    ::= STRINGVAL ; //all strings with quotes
```

Appendix D

Some conventions used in OpenMASK

D.1 Use of the configuration file

These a few conventions used :

1. all configuration files should be suffixed `.OpenMASK3`, and files used to describe the parameters of the 3DVis object `.3DVis.OpenMASK3`
2. to describe the original position and orientation of an object, the syntax should ressemble

   ```
   position [x y z]
   orientation [h p r]
   ```

3. a list of 2 names should be used to describe an output or a control parameter. For example, `configurationparameter [nameOfObject, nameOfOutput]`

Appendix E

Cameras and viewpoints in OpenMASK

E.1 Vocabulary

A camera is a construct of the 3DVis simulated object or any equivalent simulated object. For each camera, their is a part of the scene being displayed on a display.

A camera support is a construct that any partner of 3DVis can declare. It describes a position in the scene or relative to an object where it could be interesting to put a camera.

A cameraman is a simulated object whose geometry describes at least a camara support (if no geometry is associated to the cameraman, a camera support is created) and who is designed to be controlled by other simulated objects. When a camera of 3DVis is associated to a camera support of a cameraman, the point of view displayed by 3DVis is controllable by any simulated object.

A casual cameraman is a 3DVis partner with associated camera supports, but whose purpose is not to control the point of view displayed by 3DVis. Rather, the camera support are designed so that a subjective point of view for that partner can be displayed. It would be the case is a camera suppport was associted to a simulated object simulating a plane. Associating the camera to that camera support would enable to see the virtual world as from the plane.

A point of view is the result of the association of a camera to a camera support.

E.2 General notions

Three different objects are involved in camera control when using OpenMASK.

The first is of course the renderer (3DVis), which find in it's configuration parameters the definition of each of the cameras to use, and an initial (casual) cameraman and camera support to put the camera on. Therefore, the configuration parameters define the initial point of view for each display channel (viewport) managed by the renderer. The renderer also sends signals correposnding the the X events that are sent to the window where the point of view is rendered.

The second is the cameraman. A defined in the preceding paragraph, the cameraman is a simulated object that defines at least one camera support and that is controllable by other simulated objects. When a camera is associated to that camera support, the cameraman controls the

rendered point of view. This is done by placing the camera suport under a animated node in the geometric tree (a DCS in SGI Performer terminology), and by updating that node with the appropriate information. If needed, the cameraman can also send events to the renderer to ajust the cameras parameters, such as field of view or for stereovision eye angle and interoccular distance. Nevertheless, sending events requires the cameraman to be linked in a way or an other to one or more renderers. Therefore, the renderer is supposed to send an event to any cameraman (casual or not) when the camera is placed on one of it's camera supports. This enables the cameraman to know the renderes that should be updated with the appropriate rendrering paramters for the camera support.

The third simulated object involved in camera control is the keyboard navigator, which is in charge of interpreting any X event signal sent by a renderer in commands sent to a cameraman or back to the renderer. A keyboard navigator is usualy associated to exactly one renderer.

The relations between these three objects are summarized in figure E.1

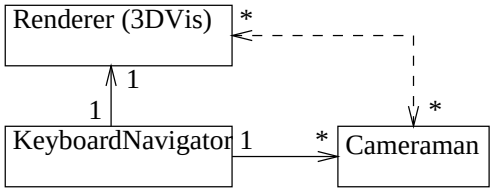

Figure E.1: Relations between simulated objects controlling the point of view

E.3 Coordinate systems of camera supports

A camera is a Performer contruct, and therefore allways is z up.

When the nodes declared as camera supports are declared in an iv file, the loader will automatically correct the camera support so that associating a camera to it will produce the expected result (a transformation is added to the node declared as the camera support to which is associated the effective camera).

A suprising consequence of that is that loading an inventor file or a pfb file produced from that inventor file will not produce the same camera supports for the application.

Bibliography

[1] David Margery. *Environnement logiciel temps-rel distribu pour la simulation sur rseau de PC*. PhD thesis, Universit de Rennes 1, 2001.

[2] S. Donikian, A. Chauffaut, T. Duval, and R. Kulpa. Gasp: from modular programming to distributed execution. In *Computer Animation'98, IEEE, Philadelphia, USA*, pages 79–87, june 1998.

[3] T. Duval and D. Margery. Using gasp for collaborative interactions within 3d virtual worlds. In *Proceedings of the Second International Conference on Virtual Worlds (VW'2000)*, Paris, France, july 2000. Springer LNCS/AI.

Contents