

Chapter 2: BASICS

This chapter introduces the foundations for the rest of the book. First, we discuss object-oriented programming since the DEVS-Scheme environment is built upon this paradigm. Then we introduce the concepts of modular, hierarchical models and the system entity structure which we elaborate in the rest of the book.

1 Object-Oriented Programming Concepts

Object-oriented programming is a paradigm in which a software system is decomposed into subsystems based on objects. Computation is done by objects exchanging messages among themselves. The paradigm enhances software maintainability, extensibility and reusability.

Conventional software systems tend to consist of collections of subprograms based on functional decomposition techniques that concentrate on algorithmic abstractions. In conventional programming languages such as LISP or Pascal, we tend to write programs around a main routine which calls on other routines to work at appropriate times. Decision making power is concentrated in the main routine. The other routines play supporting roles, coming alive only when the flow of control passes through them. In contrast, object oriented program encourages a much more decentralized style of decision making by creating objects whose existence may continue throughout the life of the program. We can make such objects act as experts in their own task assignments by providing them with the appropriate knowledge. Such distribution and localization of knowledge simplifies the main routine and relieves it of much of its decision making burden.

Indeed in object oriented programming, we abandon the concept of main routine. Instead we design an interface to the user which sends messages to the objects directing them to carry out desired tasks. This interface exists at a higher level of control than the objects in the sense that it co-ordinates their activities. One of the main advantages of object oriented programming is that the interface need not change as more and more specialized objects are introduced. This greatly simplifies program design and enhances *evolvability*. Program evolvability is the ability to easily modify a program as we learn more about the problem domain.

1.1 Objects

As just indicated, an object-oriented program contains components called *objects*. Each object has its own variables and procedures to manipulate these variables called *methods*. Only the methods owned by the object can access and change the values of its variables. The values originally assigned to variables of an object will persist indefinitely throughout its lifetime — unless changed by some method. They will stay this way until a subsequent change is brought about by some —the same this way until a subsequent change is brought about by some —the same or another— method. Thus the variables collectively constitute the state of the object. In these terms, only the methods constitute the state of the object. In these terms, only the methods of an object can alter its state.

Objects can communicate with each other, and with higher levels of control, to cause changes in their states, by a process called message passing. The general form of a message is:

to object: o

apply method: m

with arguments: a1,...,an

This represents a message sent to object O telling it to apply its method named m with the argument values a1,..., an. Carrying out the orders of this message may result in the method changing the state of the object and/or producing an output in response to the message.

One of the most useful concepts afforded by the object oriented paradigm is that different objects can have variables or methods having the same name. Such methods may exhibit a basic similarity in purpose despite a difference in the detailed manner in which this purpose is achieved. We will see why this is important soon.

As an example, consider objects representing various barn yard animals. Each such object will have a method called *make-sound*, but the result produced by the method will be different for different objects. For example,

```
send cat make-sound(soft) → returns “meow”
send cat make-sound(loud) → returns “meow meow meow”
send dog make-sound(soft) → returns “woof”
send dog make-sound(loud) → returns “woof woof woof”
```

A higher level control procedure might be the following:

Procedure: Make-racket

```
For each object in the barn yard list,
    send object make-sound(loud) and
    collect these in a list called racket.
```

Note that the procedure *make-racket* does not know what sound is generated by an object. Nor does it have to know how the individual objects do this generation. It relies only on the fact that each object has a method called *make-sound*, and leaves the details of this method to the designer of the object. The process of shielding the user of an object from its details is called *abstraction*. Such abstraction has an important consequence, called *extensibility*, the ability to extend a software system by adding in new definitions without modifying earlier ones. In our example, the procedure *make-racket* does not have to be modified when new objects are added to the barn yard list — provided that each new object is given a method called *make-sound*.

1.2 Object Classes

In object orient programming systems, objects are usually not defined individually. Instead, a class definition provides a template for generating any number of instances, each one an identical copy of a basic prototype. Thus in our example above, we might define a class called barn-yard-animals, from which the various instances, dog, cat, pig, horse, etc., would be generated. The basic form for such a class definition is given in Figure 2.1.

```

Define class: name      (name to be used for the class)

class variables:      (variables describing the class per se)

instance variables:  (variables owned by each instance)

methods:             (methods owned by each instance)

constructor:         (procedure for creating instances)

destructor:         (method for destroying instance)

inheritance:        (other classes from which to inherit definitions)

```

Figure 1: Basic form of Class Definition.

In our example, we might provide the following definition:

```

Define class: barn-yard-animals
  class variables: barn-yard-list
  instance variables: name, sound
  methods:
    (set-sound(sound-string) sound:= sound-string)
    (get-sound() return sound )
    (get-name() return name)
    (make-sound(strength)
      (if (strength = loud)
        return: concatenate (sound ,sound,sound)
        else if (strength = soft) return: sound
        else return: wrong input argument ))
  constructor:
    (make-barn-yard-animal(name-string)
      -- make an instance with name, name-string,
      and place it on the barn-yard-list -- )
  destructor:

```

```
(destroy() -- remove this instance
  from the barn-yard-list -- )
inheritance: none
```

This definition specifies a class variable called `barn-yard-list`, which will be used to keep track of the instances of the class that are currently in existence. When an instance is created using the defined constructor, it is placed on the `barn-yard-list`. To remove it from this list, we send it a message to apply its method `destroy`. Note that each instance has a destructor method. However, to construct an instance, we cannot employ one of its methods — the instance does not yet exist, so we cannot send a message to it.

There are two instance variables, `name`, and `sound`, respectively, to hold the name of the instance and a string for the sound it makes. As indicated above, the only way to change values of an object's variables is to apply one of its methods. The only exception to this rule pertains at the birth of an instance. The constructor can assign initial values to the instance variables individually, or such values can be specified in the class definition as default values assigned to each instance upon creation. In our example, the constructor assigns the name to the newly created instance and a method, `set-sound`, can be employed to initially assign a sound to the instance.

Note that the sound can be changed at any time by applying the `set-sound` method. However, the `name` cannot be changed, since no method has been defined to do this. (There is a method to obtain the name). In general, we may define set- and get- methods for those instance variables whose values we wish to modify and access, respectively.

In our example, we can create the instances mentioned previously and assign them appropriate values as follows:

```
make-barn-yard-animal(cat)
send dog set-sound "meow"
make-barn-yard-animal(dog)
send dog set-sound "woof"
```

Having done so, calling the procedure, `make-racket`, will return the list

```
racket: "woof woof woof" "meow meow meow"
       "oink oink oink" "neigh neigh neigh"
```

(remember that `barn-yard-list` employed by `make-racket` contains the created instances). The following would destroy horse and create cow:

```
send horse destroy
make-barn-yard-animal(cow)
send cow set-sound "moo"
```

```

;;;class definition of animals

(define-class animals
  (classvars
    (barnyard-list '()) ;;the list of instances, initially empty
  )
  (instvars
    name           ;;name of animal, a symbol
    self           ;;this object itself
    (sound "")     ;;the sound it makes, a string
  )
  (options
    settable-variables ;;each instance variable has a corresponding
                      ;;set method
    gettable-variables ;;each instance variable has a corresponding
                      ;;get method
    inittable-variables) ;;each instance variable can be initialized in
                      ;;the general constructor, make-instance
  )

  (compile-class animals) ;;needed to make instances of the class
                          ;;not needed if class is only used as a source
                          ;;hereditary material

;;;:::methods for animals

(define-method (animals make-sound)(strength)
  (if (equal? strength 'loud)
      (string-append sound sound sound)
      sound
  )
)

(define-method (animals destroy)()
  (delete! self barnyard-list)
)

;;;::: example of construction of an instance
;;;::: uses set methods existing due to class definition

(define cat (make-instance animals))
(send cat set-sound "meow ") ;;uses set method for sound
(send cat set-name 'cat)    ;;name is symbol, 'cat
(send cat set-self cat)     ;;self is object, cat

;;;::: another example of instance construction
;;;::: uses inittable capability defined in class definition

(define dog (make-instance animals 'sound "woof " 'name 'dog))
(send dog set-self dog)
(setcv animals barnyard-list (cons dog (getcv animals barnyard-list)))

```

Figure 2: Class Definition of Animals.

Make-racket would now return a correspondingly modified list of sounds. Once again, note that such a procedure need not be modified whenever we create new instances or destroy existing ones. What's more it does not have to be changed when new classes are added to the system as we next consider doing.

For comparison, Figure 2.2 shows a class definition of animals as it is done in SCOOPS (Scheme Object-oriented Programming System). It is beyond the scope of this book to delve into the details of object-oriented programming systems in Scheme (see Texas Instruments (1986) PC-Scheme Users Manual; see Smith (1989) or Eisenberg (1988) for introductions to Scheme programming). However, from time to time, we shall touch upon some facets of the implementation of DEVS-Scheme in SCOOPS when it is of special interest to do so.

```
#include <stream.h>

class animal{

protected:

char * name;
animal * self;
char * sound;

public:

animal(char * Name,char * Sound){
    name = new char[strlen(Name) + 1];
    name = Name;
    sound = new char[strlen(Sound) + 1];
    sound = Sound;
    self = this;
    //self not needed: this provided by C++
}

char * make_sound(char * strength){
char * output = new char[3*strlen(sound) + 1];
strcpy(output,sound);
if (strength == "loud"){
    strcat(output,sound);
    strcat(output,sound);
}
return output;
}

~animal(){
delete [] name;
```

```

delete [] sound;
self = 0;
}
};

main(){
animal cat("cat","meow ");
cout << cat.make_sound("loud") <<endl;

animal * dog = new animal("dog","woof ");
cout << dog->make_sound("quiet") <<endl;
delete dog;
}

```

Figure 2.2. Class definition of animals.

obtain all their definitions. This saves having to copy or rewrite such common variables and methods and helps maintain consistency in definitions when code modifications are made.

1.3 Class Inheritance

Often objects can be organized into a family of homogeneous classes, which are mutually distinct, but do share certain fundamental properties in common. The object oriented paradigm provides a natural way to exploit such situations to afford an economy of definition as well as the extensibility just referred to. As in Figure 2.1, a class definition can specify from which classes the new class will inherit, i.e., automatically obtain all their definitions. This saves having to copy or rewrite such common variables and methods and helps maintain consistency in definitions when code modifications are made.

In the most straightforward case, a class inherits only from at most one other class, called its parent. In this case, the classes form a tree structure, called a specialization hierarchy under a root class. The root class is the most general class. Its children inherit all its features and are more specialized, in that they may have additional ones as well. Their children may be even more specialized, inheriting from the parents (and hence from the grandparent). For example, we could define specialized classes fowl and cattle of barn-yard-animals. Each class definition would have barn-yard-animals in its inheritance slot. Objects in such classes would automatically be given all the instance variables and methods defined for barn-yard animals. In addition, we could add new class and instance variables and methods to fowl that are particular to chicken, geese, etc. as opposed to cattle, such as cows, oxen, etc. Figure 2.3 shows how this may be done in SCOOPS.

```

class fowl:public animal{

protected:

```

```

static int price_of_eggs; //class variable
int egg_laying_frequency;

public:
foul(char * Name, int Freq):animal(Name,"cluck"){
egg_laying_frequency = Freq;
}

int market_value(){
return egg_laying_frequency * price_of_eggs;
}

};

int foul::price_of_eggs = 10;
    //class variable must be initialized outside class declaration

main(){
foul * chick1 = new foul("chick1",2);
cout << chick1->market_value()<<endl;
foul * duck1 = new foul("duck1",1);
cout << duck1->market_value()<<endl;
delete chick1;
delete duck1;
}

```

Figure 2.3. Specialized classes of animals.

The root class generally provides general definitions for methods needed to interface with higher level procedures. The more specialized classes may override such general definitions by providing methods of their own with the same name. For example, in Figure 2.3, *foul* and *cattle* each have methods with the same name but with different behavior. The term *polymorphism* is used to refer to such multivalued definition: object-oriented programming systems obviously must interpret the meaning of a method name in its proper context, a process called *dynamic binding*¹. Polymorphism and dynamic binding are important features which distinguish message passing from ordinary procedure calls.

The system may evolve by adding more and more specialized classes, while the higher level procedures need not change so long as the newly introduced specialized methods are compatible (same input and output) as the general ones. Thus extensibility and ease of program evolution are inherent in the object-oriented programming approach.

The more general form of organization in which classes may inherit from several classes, called multiple inheritance, provides additional flexibility in certain situations but is harder to keep track of. Layer 1 of the DEVS-Scheme environment uses mainly the hierarchical organization but Layer

2 provides extensive support of multiple inheritance by means of the system entity structure.¹

2 The System Entity Structure/Model Base

While object-oriented programming provides the means for implementing knowledge-based simulation environments, the The System Entity Structure/Model Base (SES/MB) framework provides the ends to which these means are applied. Here we provide a general overview of these concepts so as to provide a context for later detailed discussion.

DEVS-C++ is implemented over the container library described in “Objects and Systems”, by the same author (Springer-Verlag, 1995). The classes are roughly characterized as follows:

- containers – the base class, provides basic services for the derived classes
- bags – counts numbers of object occurrences
- sets – only one occurrence of any object is allowed in.
- relations – are sets of key-value pairs, used in dictionary fashion
- functions – are relations in which only one occurrence of any key allowed
- orders – maintains items in given order
- queues – maintains items in FIFO order
- stacks – maintains items in LIFO order
- lists – maintains items in order determined by insertion index

2.1 Modularity and Model Base Concepts

Hierarchical synthesis and reuse of models are greatly facilitated if the objects in the model base are modular in the sense we now describe. As in Figure 2.4, suppose that we have models A and B in the model base. If such model descriptions are in the proper modular form, then we can create a new model by specifying how the input and output ports of A and B are to be connected to each other and to external ports, an operation called *coupling*.

Oren (1980,1984) was the first to introduce this concept into simulation languages. The resulting model, AB, called a coupled model is once again in modular form. As can be seen, the term *modularity*, as used here, means the description of a model in such a way that it has recognized input and output ports through which all interaction with the external world is mediated. Once placed into the model base, AB can itself be employed to construct yet larger models in the same manner used with A and B. This property, called closure under coupling enables hierarchical construction of models.

A coupling specification has three parts:

¹The recent standardization of object oriented approaches in CLOS (Common Lisp Object System; Keene, 1988) takes things one step further. It supports a very flexible method based on the classes of its arguments. Ordinary inheritance is then the simple case in which only one argument determines how a generic method is to be interpreted.

- external input coupling tells how the input ports of the coupled model are identified with the input ports of the components.
- external output coupling tells how the output ports of the coupled model are identified with the output ports of the components.
- internal coupling specifies how the components inside the coupled model are interconnected by telling how the output ports of components are connected to input ports of others.

It is important to remember that the resulting coupled model cannot be used in further model construction unless external coupling is specified. However, internal coupling can be omitted: a non-interacting, parallel composition of components results in this case.

3 Independent Testability

An important benefit of such modular construction is that any model in the model base can be readily, and independently, tested by injecting the proper test message sequences and comparing the results with those expected. The ability to do such testing at each stage of a hierarchical construction facilitates reliable and efficient verification of large simulation models. Such systematic testing is not possible in standard simulation languages which do not support modular and hierarchical model development. Test modules for models can be developed in a systematic manner using the concept of experimental frame, which specifies the input, control and output variables and constraints desired of the experimentation.

Models that have been validated in experimental frames can be put into the model base for reliable reuse as components in larger models. A caveat: such validation is relative only to the tests that have been done. In reusing a model we should be aware that the behavior elicited in its new context may not have been covered in earlier validation experiments.

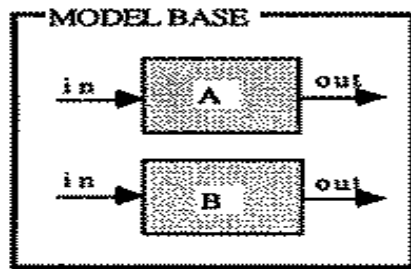
3.1 Hierarchical Construction: The Composition Tree

A *hierarchical* model is inductively defined:

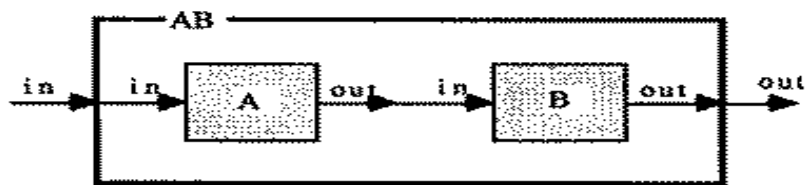
- an atomic model is a hierarchical model
- a coupled model whose components are hierarchical models is a hierarchical model
- nothing else is a hierarchical model. ²

The structure of a hierarchical model is exhibited in a *composition tree* such as that in Figure 2.5. The components of the outermost coupled model, or root model, are shown as its children

²The last clause is not superfluous: it says the only way to get hierarchical models is by following the inductive process defined in the first two clauses

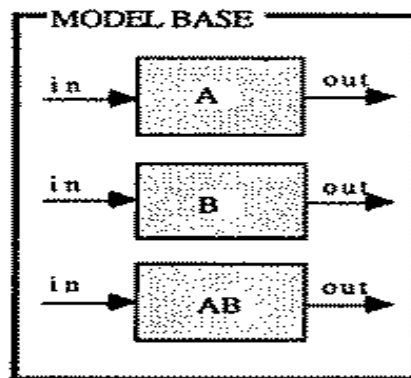


(a)



Coupling: external input: AB.in \rightarrow A.in
 external output B.out \rightarrow AB.out
 internal: A.out \rightarrow B.in

(b)



(c)

Figure 3: Modularity and Model base concepts.

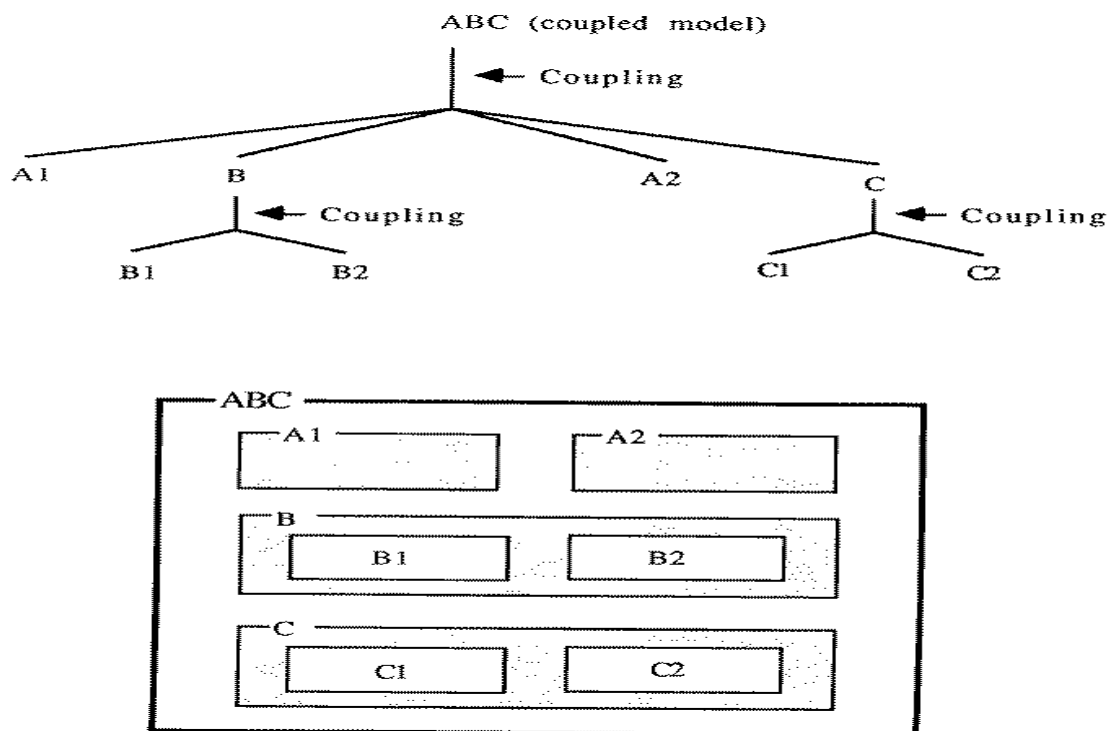


Figure 4: Composition tree.

(e.g., A1, A2, B and C are the children of ABC). A child, which is also a coupled model, has its components descending from it as children (e.g., B and C). Children which are atomic models become leaves in the tree (e.g., B1, B2, C1, and C2). The coupling specification needed to construct a coupled model is attached to vertical line descending from the parent to its children. In other words, the coupling specification is associated with a decomposition of the parent into its children. The system entity structure, to come next, generalizes this concept. In it, an entity may have several decompositions, called aspects, associated with it.

Each such aspect carries with it the coupling specifications needed to construct the parent by coupling together the children of that aspect.

Thus a composition tree represents the information needed to construct a particular hierarchical model. The SES represents the possible construction of a whole family of hierarchical models.

3.2 System Entity Structure

The *system entity structure* (SES) directs the synthesis of models from components in the model base (Figure 2.6). The SES is a knowledge representation scheme that combines the decomposition, taxonomic, and coupling relationships. Associated with an SES is a model base which contains

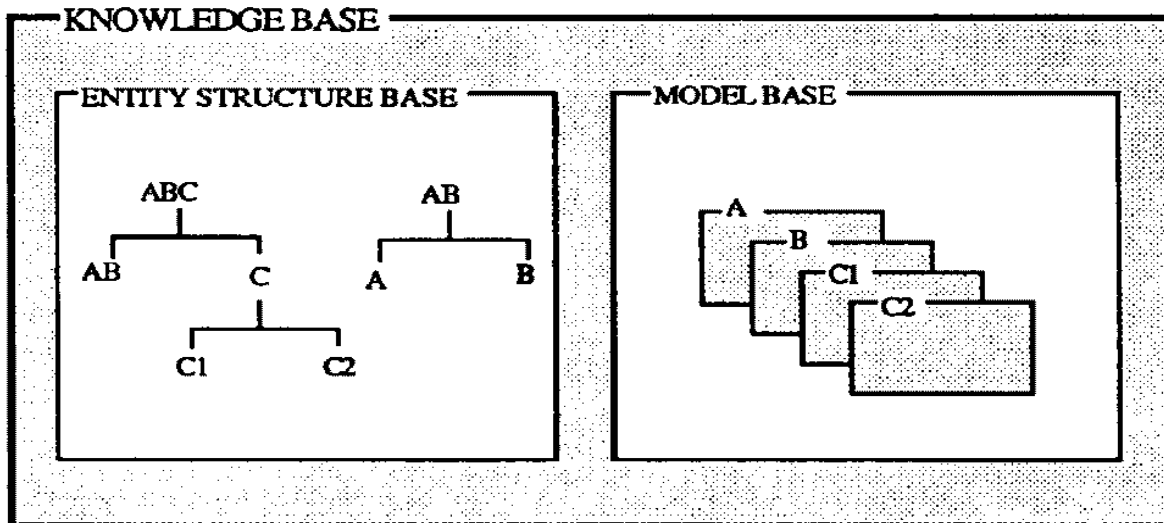


Figure 5: The System Entity Structure/Model Base (SES/MB) framework.

models which may be expressed in any of dynamical formalisms mentioned earlier. The *entities* of the SES refer to conceptual components of reality for which models may reside in the *model base*. Also associated with entities (as with other object types we shall discuss) are slots for attribute knowledge representation. An entity may have several *aspects*, each denoting a decomposition and therefore having several entities. An entity may also have several *specializations*, each representing a classification of the possible variants of the entity.

Whereas the entity-aspect relation conveys decomposition knowledge, the entity-specialization relation represents taxonomic knowledge. Specializations (classification schemes) also have several entities. Specializations, can be thought of as partitions; the product of two partitions forms a finer partition whose blocks are the intersections of the originals. In pruning, when one entity from a specialization is selected, it *inherits* the substructure (slots, aspects and remaining specializations) of its parent. The selected entity also replaces the parent in any coupling specifications involving the latter. Specializations may form a hierarchy analogous to the object class hierarchy. It may happen that not all combinations of specialization choices represent actual possibilities. In this case, *constraints* can be added to the specializations to express the allowable combinations.

A *multiple entity* represents the set of all members of an entity class. Such a multiple entity always has an aspect which is its multiple decomposition into the individual entities of the class. Class variables, carrying aggregation and distribution information, are associated with the multiple entity, whereas instance variables, belonging to each instance of the class are associated with the entity.

One application of the SES/MB framework is to the design of systems. Here the SES serves as means of organizing and generating the possible configurations of a system to be designed. Al-

though we will discuss the SES formalism in greater depth later, we discuss an example at this point so as to facilitate the reader's visualization of the concept.

4 Artificial Worlds Example

There is increasing interest in design and construction of "artificial" worlds, in the sense of large scale enclosures supporting human, animal and plant habitation placed in environments, such as the moon, which would otherwise not support life. Modern materials technology has made it possible to build and isolate such systems from interaction with external environments. Modern high technology has supplied the information processing hardware needed to "wire" the system for the necessary observation and control. Modern design methodology must be invoked to marshal these powerful means toward achieving desired objectives. Proper design entails intensive application of techniques of modelling and simulation and of knowledge representation. Artificial worlds engineering is a prime example of the need for multifaceted modelling methodology to deal in a coherent manner with the multiplicity of facets involved in large scale systems.

A biosphere *biosphere* is a stable, complex, evolving system containing life, composed of various ecosystems operating in a synergetic equilibrium, completely closed to material input or output, and open to energy and information exchanges. The earth ecosystem is the only such biosphere currently known. Biosphere II (Hayes, 1989) is a unique, non-governmentally sponsored project to create an analog of the earth ecosystem (Biosphere I) in an encapsulated structure spanning 2.5 acres located 20 miles north of Tucson Arizona. The knowledge gained in experimenting with Biosphere II, a faster responding replica of the earth, could be invaluable in solving the latter's ecosystemic problems. Such a biosphere would provide a potential means of establishing permanent manned stations in space, or on other planets, as research and observation bases and eventually colonization.

Complete material closure implies recycling of wastes and regeneration of all consumable resources: a major set of problems for which Biosphere II may demonstrate solutions. However, complete material closure is not a necessity in many life support environments such as projected space stations, antarctic colonies, manned expeditions below the earth's surface, radiation impregnable habitats, etc. In such circumstances, although recycling of wastes and regeneration of vital resources would be advantageous, disposal of some wastes and resupply of some consumables might prove more feasible than complete closure. The point to be made is that there are a great variety of alternatives in the continuum between partial closure and complete closure to be explored in the engineering of such artificial worlds.

4.1 SES for Life support systems

Figure 2.7 is a system entity structure for designing life support systems in artificial worlds. The figure shows that a LSS (life support system), the top-most, or root, entity, in its broadest context consists of number of subsystems including: (reading from left to right) a radiation protection subsystem, an expendables resupply subsystem, an energy supply system, etc. These subsystems are

identified as entities falling under an aspect called LSS-DEC, which represents the decomposition of the LSS into the just mentioned components. Generally, entities may have one or more ways of being decomposed, each represented by an aspect (labelled decomposition). For example, the food supply system has a decomposition, FOOD-SUP-DEC which breaks it into food production and food storage subsystems. The food production subsystem is further decomposed, under food-prod-dec, into several entities, the growth medium, the harvesting subsystem, nutrient supply subsystem, and plant waste recycling system.

Also ubiquitous in Figure 2.7a) is a third organizational concept (besides entity and aspect), namely specialization. Specialization is a means of representing the possible variants or forms that an entity might assume. For example, the LSS might be biologically based, physical/chemical based, or a hybrid of both. This is represented in the specialization *lss-spec*, shown with a double line, as are all specializations to visually distinguish them from aspects. A biologically based LSS has a specialization, BIO-SPEC, which has as one variant an earth biome-based bioregenerative LSS such as Biosphere II, and other biologically based alternatives that do not attempt to replicate the function of the earth's biomes.

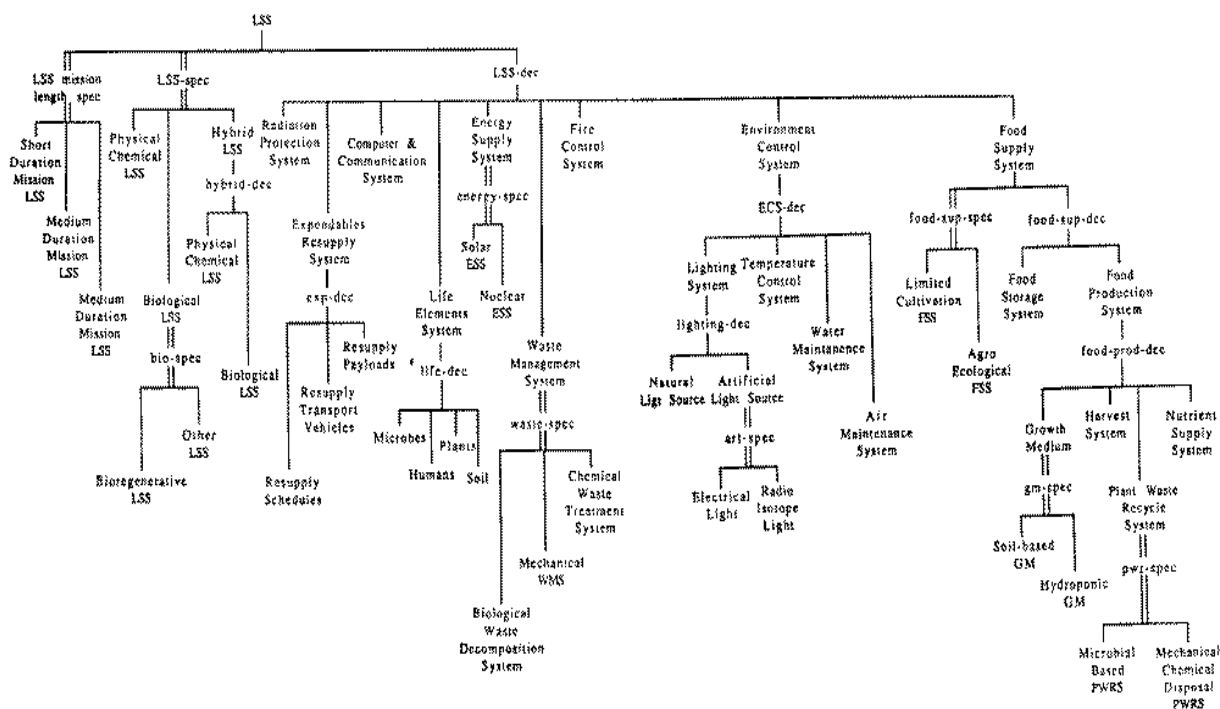
A property of the entity structure formalism is that it automatically brings to bear all the defining features of a component in every context in which it appears. For example, since biological LSS is a component of a hybrid LSS, the alternatives for biological LSS, shown under the *bio-spec* specialization, also present themselves as choices in this context, as well as in the non-hybrid alternative. This means that when considering hybrid life support systems, we should consider the various combinations of alternatives that will be available for both its biological and physical/chemical components. Although many of these combinations may not be feasible, some very good ones may emerge that might not have been considered in a non-systematic approach.

System design is the process of generating candidate designs and ranking them with respect to design objectives. To start generating a candidate we use a process called *pruning* which reduces the SES to a so-called *pruned entity structure* (PES). An example of such a pruned structure is shown in Figures 7b). Note that such structures are derived from the governing structure by a process of selecting from alternatives where ever such choices are presented (we shall have much more to say about pruning later). Not all choices may be selected independently. Once some alternatives are chosen at a high level, some options are closed and others are enabled. Moreover, rules may be associated with the entity structure which further reduce the set of configurations

that must be considered. For example, a long duration bioregenerative mission would not need an expendables resupply system – having selected this type of mission from the specializations under LSS a constraint rule would automatically eliminate expendables-resupply-system from LSS-DEC.

5 SES Pruning and Model Synthesis

As shown in Figure 2.8, pruned entity structures are stored along with the SES in files forming the *entity structure base*. Hierarchical simulation models may be constructed by applying the *trans-*



form function to pruned entity structures in working memory. As it traverses the pruned entity structure, *transform* calls upon a retrieval process to search for a model of the current entity. If one is found, it is used and transformation of the entity subtree is aborted. *Retrieve* looks for a model first in working memory (we shall see that such a model may be an instance of the classes *atomic-models* or *coupled-models*, whose main subclasses are *digraph-models* and *kernel-models*).

If no model is found in working memory, then the *retrieve* procedure searches through model definition files, and finally, provided that the entity is a leaf, in pruned entity structure files. A new incarnation of the *transform* process is spawned to construct the leaf model in the last case. Once this construction is complete, the main *transform* process is resumed.

The result of a transformation process is a model expressed in an underlying simulation language such as DEVS-Scheme which is ready to be simulated and evaluated relative to the modeler's objectives.

The fact that the transform process can look for previously developed pruned entity structures, in addition to basic model files, has an important consequence for reusability – as we shall see in Chapter 13. The reader is certainly allowed to peak ahead to see why. However, before proceeding with the more advanced ideas, let us build up our understanding of DEVS-Scheme:

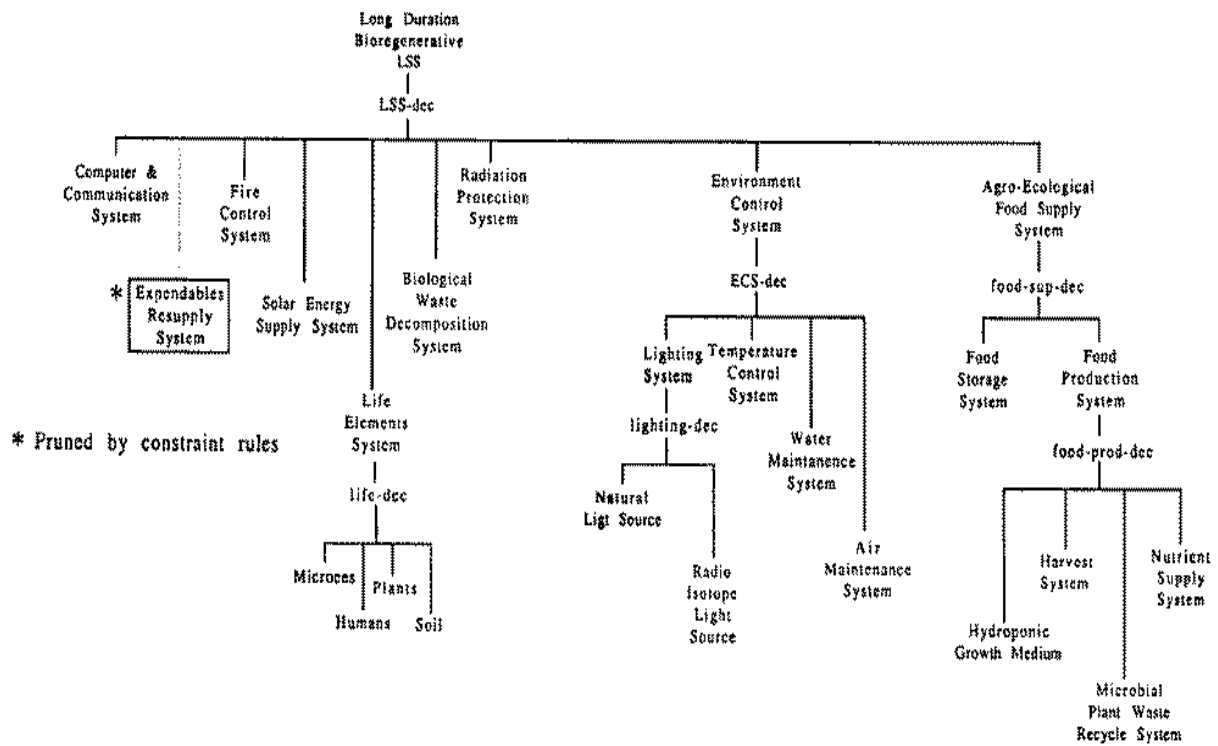


Figure 6: Pruned system entity structure for long duration mission

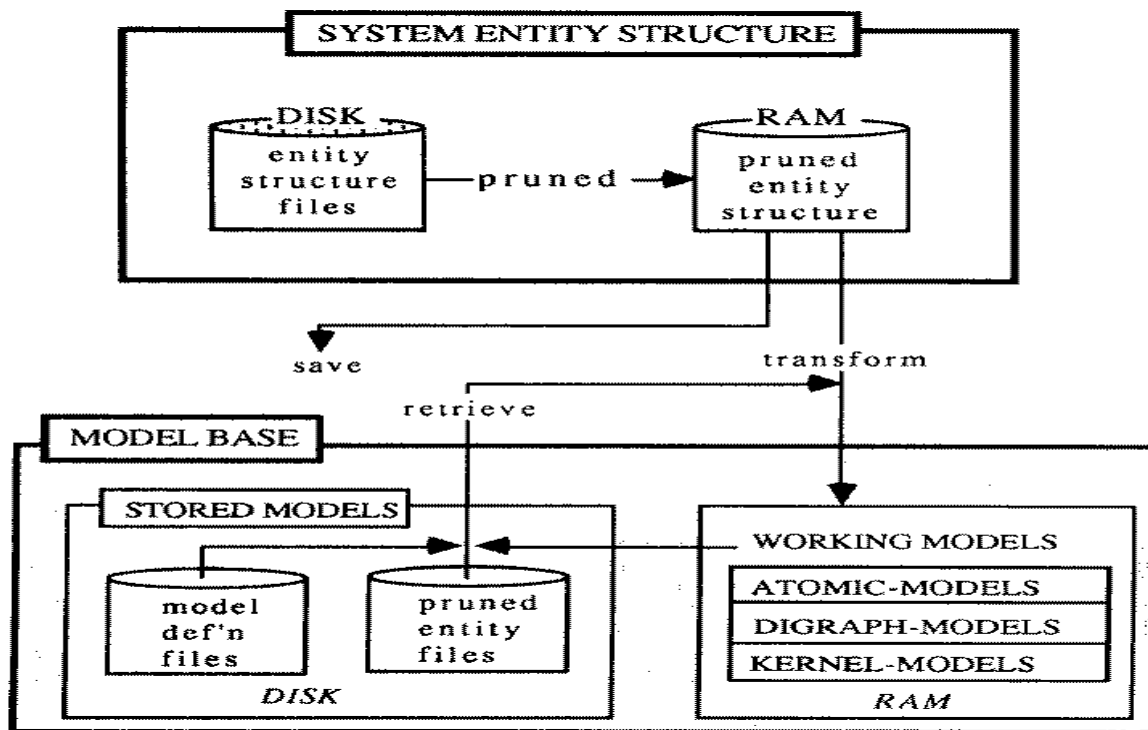


Figure 7: The System Entity Structure/Model Base (SES/MB) Environment.