Chapter 7: SYSTEM ENTITY STRUCTURES

This chapter discusses the use of the *Systems Entity Structure (SES)* to specify hierarchical models and to organize them for reuse from an archival model base. We illustrate these concepts in modeling and simulating simple computer architectures. We begin with a review of SES concepts.

Recall that a knowledge representation scheme for managing a family of models must support the following three relationships: *decomposition, taxonomy*, and *coupling*. Knowledge about *decomposition* means that there are schemes for representing the manner in which an object is decomposed into components. The schemes are hierarchical since components themselves may be decomposed into subcomponents, and so on, to a depth determined by the modeller's objectives.

The requirement for *taxonomic* knowledge means that there must be a method of organizing the different kinds of objects, i.e., how they can be categorized and subclassified. For example, a scheme could "know" that automobile transmissions are automatic or manual, and that the latter can be of the four-speed or five-speed variety.

The requirement for *coupling* knowledge means that there must be a way of representing how models are coupled together and what constraints apply to component combinations.

1 SYSTEM ENTITY STRUCTURE DEFINITIONS AND AXIOMS

The *System Entity Structure* (SES) is defined as labeled tree with attached variable types which satisfies the following axioms:

1. *uniformity*: Any two nodes which have the same labels have identical attached variable types and isomorphic subtrees.
2. *strict hierarchy*: No label appears more than once down any path of the tree.
3. *alternating mode*: Each node has a mode which is either *entity*, *aspect*, or *specialization*; if the mode of a node is *entity* then the modes of its successors are *aspect* or *specialization*, if the mode of a node is *aspect* or *specialization*, then the modes of its children are *entity*. The mode of the root is *entity*.
4. *valid brothers*: No two brothers have the same label.
5. *attached variables*: No two variable types attached to the same item have the same name.
6. *inheritance*: every *entity* in a *specialization* inherits all the variables, *aspects* and *specializations* from the parent of the *specialization*

The SES is completely characterized by its axioms (Zeigler, 1984; Zhang and Zeigler, 1989). However, the interpretation of the axioms cannot be specified and thus is open to the user. When constructing a SES it may seem difficult initially to decide how to represent concepts of the real

world. How to choose between *entity*, *aspect* or *specialization*? To help make this decision the following points should be kept in mind:

An *entity* represents a real world object that either can be independently identified or postulated as a component of a decomposition of another real world object.

An *aspect* represents one decomposition out of many possible of an entity. The children of an aspect are entities representing components in a decomposition of its parent.

A *specialization* is a mode of classifying entities and is used to express alternative choices for components in the system being modeled. The children of a specialization are entities representing variants of its parent. For example, in an SES for a computer system, the entity *printer* could have such specializations as: *size*, *typeface*, and *interface-type*. The children of interface-type might be *parallel interface* and *serial interface*. These are variants for the interface of printer. That printers also come in various sizes is represented in the specialization *size*.

The entities of an aspect represent distinct components of a decomposition. A model can be constructed by connecting together some or all of these components. The aspects of an entity do not necessarily represent disjoint decompositions. A new aspect can be constructed by selecting from existing aspects as desired.

The properties of a SES are illustrated in a computer example (Figure 7.1). The root entity is COMPUTER and it has a specialization, shown by two vertical lines, called CLASS-SPECIALIZATION with entities ANALOG, DIGITAL, and HYBRID. In such a specialization relation, COMPUTER is referred to as a generic type relative to the entities, ANALOG, DIGITAL, and HYBRID, which are called special types. Besides having their own distinctive attributes, HYBRID, ANALOG, and DIGITAL inherit all of the attributes (variables and substructures) possessed by COMPUTER. To make this true, we must be sure to assign to COMPUTER only those attributes that are common to all its variants.

HYBRID is shown as having a decomposition into ANALOG and DIGITAL, i.e., it is a system built from two component systems. By the uniformity axiom, the DIGITAL part of a HYBRID computer has the same PHYSICAL-DECOMPOSITION shown under the occurrence of DIGITAL as a special type of COMPUTER. In other words, when a DIGITAL_COMPUTER is combined with an ANALOG_COMPUTER, its internal structure is the same as if it were free standing.

The SES makes it possible to represent, and distinguish between, two types of property transfer related to multiple inheritance in object-oriented programming (Chapter 2). The first kind is illustrated above in the fact that HYBRID, being decomposed into DIGITAL and ANALOG components, "inherits" properties from both through the uniformity axiom. In this case, the sources of the "inheritance" are represented as distinct components and it is possible to take account of their interaction through coupling specification associated with the parent decomposition. This contrasts with object-oriented inheritance in which attributes are simply accumulated.

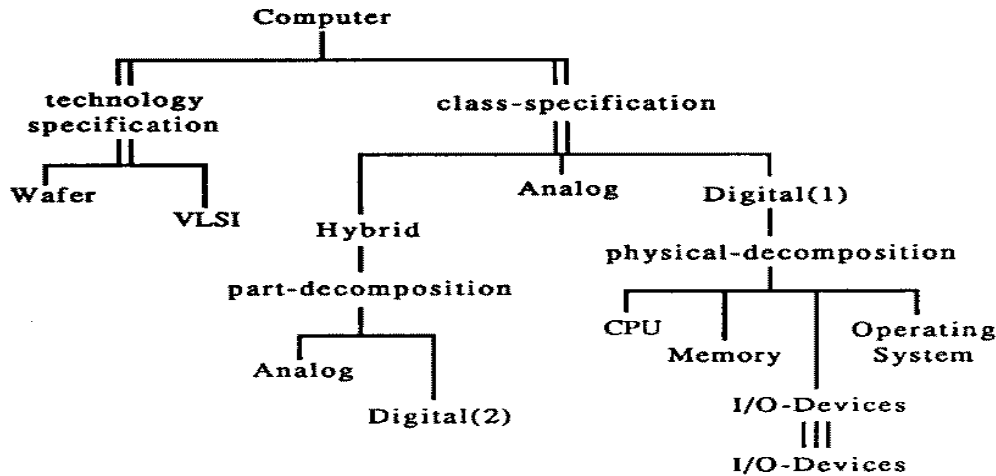


Figure 1: System entity structure for electronic computers

The second kind of property transfer is that directly corresponding to multiple inheritance in object oriented systems. This occurs when a succession of selections is made from a set of specializations under the same entity. For example, first select DIGITAL from CLASS-SPECIALIZATION under COMPUTER. Then the TECHNOLOGY-SPECIALIZATION of COMPUTER is inherited by DIGITAL. Selecting VLSI from it, we have a result VLSI_DIGITAL_COMPUTER which has inherited from both DIGITAL and VLSI. Later (Chapter 9), we shall see that to be meaningful for models, this kind of accumulation of properties assumes a underlying superposition principle.

The triple vertical bars connecting I/O_DEVICES and I/O_DEVICE in Figure 7.1 represent a special type of decomposition called a *multiple decomposition*. A *multiple decomposition* is used to represent entities whose number in a system may vary. For example a digital computer may have 0, 1, 2, or more I/O_DEVICEs.

As you may recall the coupling relationship defines how the entities (models) communicate with each other. Since the aspects define the decompositions of the system, the coupling relationships must be associated with their respective aspects.

2 USING THE SYSTEM ENTITY STRUCTURE IN DEVS-SCHEME

In the first place, the SES provides an alternative to the digraph model approach for specifying coupled models. In DEVS-Scheme this is a high level of specification which is then transformed into an equivalent coupled model, similar to the way in which a procedural language is compiled into assembly language. Using a higher level specification considerably simplifies the model descrip-

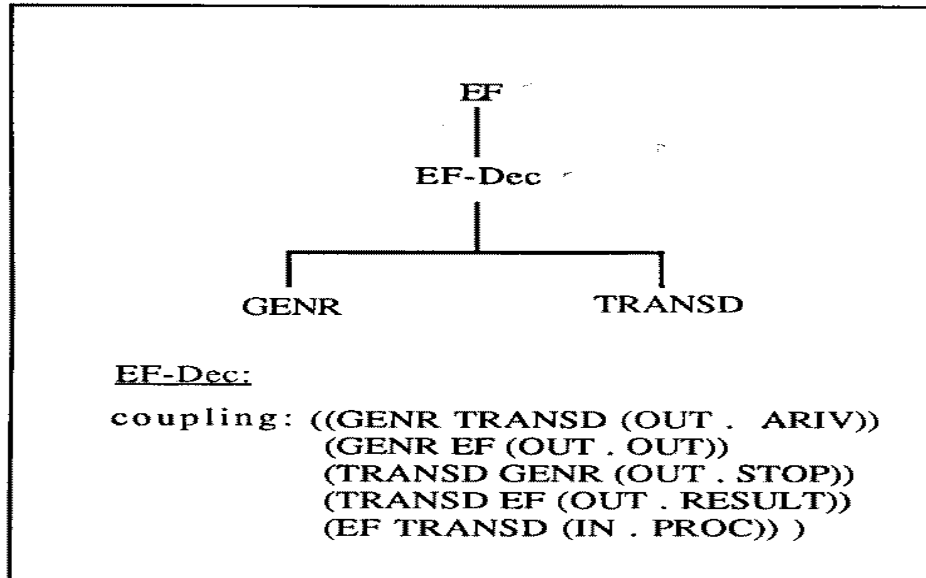


Figure 2: Entity structure for experimental frame component

tion task in the same way that writing a program in a procedural language simplifies writing the equivalent code in machine language. However, we shall see that the primary feature of the SES is that it serves as a knowledge representation scheme to organize a family of models in a model base.

Figure 7.2 illustrates an entity structure for the experimental frame component discussed earlier. The SES asserts that an entity EF, the root of the structure, is decomposed into entities TRANSD and GENR using aspect EF-DEC. The coupling specification which is employed to synthesize EF from TRANSD and GENR is associated with the aspect EF-DEC. This coupling is presented as a list of elements of the form (comp1 comp2 (port1 . port2)) in Figure 7.2. The type of coupling (external-input, external-output, or internal) is recognizable from the component sequence: comp1 comp2. For example, (GENR TRANSD (OUT . ARIV)) specifies the internal coupling from GENR's outport 'out to TRANSD's input port 'ariv. (GENR EF (OUT . OUT)) specifies external-output coupling from GENR to the enclosing digraph model, EF. Note that the digraph model's instance variables, *composition-tree* and *influence-digraph*, can be inferred from the decomposition and coupling information in the SES.

C++:

```
//ef_ses.h

#include "ses.h"

class ef_ses:public ses{
```

```
public:

ef:_ses():ses("ef"){
ent * ef = make_ent("ef");
set_root(ef);

asp * ef_dec = make_asp("ef_dec");
add_asp_to_ent(ef_dec,ef);

ent * transd = make_ent("transd");
ent * genr  = make_ent("genr");
add_ent_to_asp(transd, ef_dec);
add_ent_to_asp(genr, ef_dec);

add_couple(ef,transd,"in","solved");
add_couple(transd,ef,"out","resolved");
add_couple(transd,genr, "out","stop");
add_couple(genr,ef,"out","out");
add_couple(genr,transd,"out","ariv");
}
};
```

Figure 7.3 shows how the SES of Figure 7.2 is specified. First the SES object E:EF, is constructed using *make-entstr*. In creating E:EF, *make-entstr* makes a root entity with name EF. Then the aspect, EF-DEC is added to the root, EF using the operation, *add-item*. To add entities to this aspect, we must first move the current-item pointer from EF, where it starts to EF-DEC. In general, all operations are performed with respect to the current-item (i.e., the item designated by the last *set-current-item* operation). Using *add-item* again, the entities GENR and TRANSD, are added to the EF-DEC aspect. Without moving the *current-item*, the couplings to be associated with EF-DEC are added using the operation *add-couple*. The *save-en* command compiles the information defining E:EF in a file, ef.e. Subsequently, E:EF is known to the entity structure manager and it can be quickly loaded with the command(*load-entstr e:ef*).

An entity structure for the simple processor/experimental frame pair EF-P is shown in Figure 7.4. The coupling associated with the aspect is the same as that represented in the digraph model EF-P described earlier. The same approach to specifying this SES is used as just discussed, except that now we use the operation:

(*add-priority e:ef-p '(p ef)*)

to specify that the select function will be based on the priority scheme in which P takes priority over EF.

Later we shall see how the transformation procedure conveniently handles the situation where a leaf entity, such as EF in Figure 7.4, is itself a coupled model, specified by another SES. For now

```
;-------------------------------------------------------------------
; This file contains the experimental frame for all architectures
;-------------------------------------------------------------------

;; make an entity structure with root EF

(make-entstr 'ef)

;; add an aspect for decomposition

(add-item e:ef asp 'ef-dec)

;; experimental frame consists of generator and transducer

(set-current-item e:ef 'ef-dec)
(add-item e:ef ent 'transd)
(add-item e:ef ent 'genr)

;-------- coupling -----------------------------------

(add-couple e:ef 'ef 'transd 'in 'solved)
(add-couple e:ef 'transd 'ef 'out 'result)
(add-couple e:ef 'transd 'genr 'out 'stop)
(add-couple e:ef 'genr 'ef 'out 'out)
(add-couple e:ef 'genr 'transd 'out 'ariv)

; save e:ef in a file ef.e

 (save-en e:ef)
```

Figure 3: Specification of SES for EF

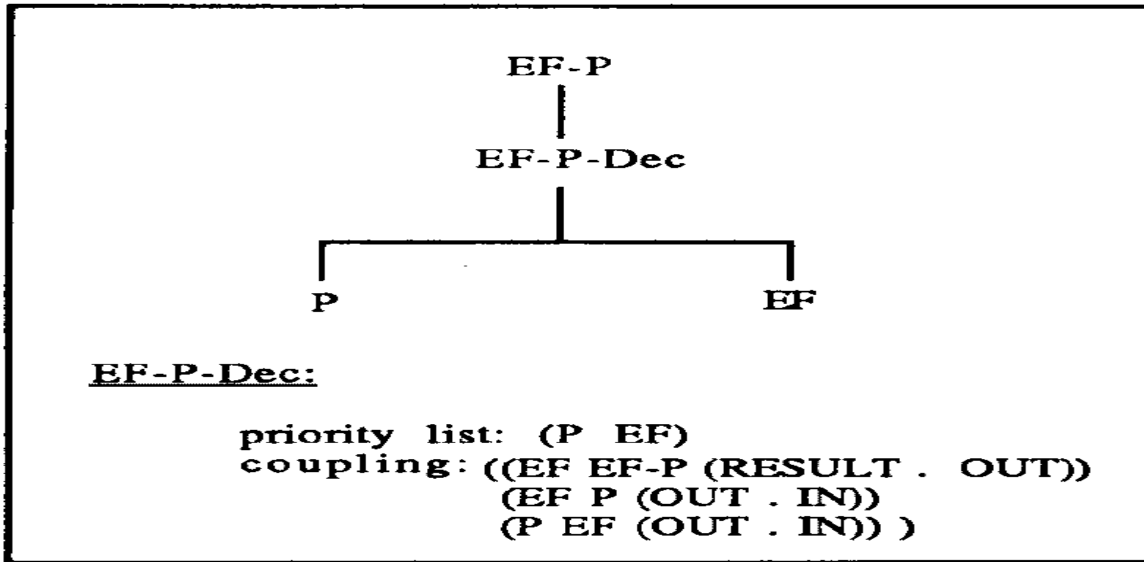


Figure 4: SES for processor/frame pair

we extend the SES of Figure 7.4 to include the specification for entity EF as shown in Figure 7.5.

2.1 Simulating directly from an entity structure

The SES, E:EF-P is said to be *pure*, i.e., it has no specializations and at most one aspect under each entity. Such an SES can be directly transformed into a hierarchical model and simulated. The transformation procedure expects that models for the leaves of the SES will be available: having files p.m, genr.m and transd.m in the model base directory will satisfy this requirement. After synthesizing the digraph model EF-P, the transformation procedure makes a root-co-ordinator R:EF-P and initializes it to the co-ordinator C:EF-P, it has also made. Thus, as shown below, after transforming, the model is ready to be simulated and awaits the command (*restart r:ef-p*) to do so.

```
;; load the entity structure
(load-entstr e:ef-p)
;; transform it into a hierarchical model and initialize it
;; with a root-co-ordinator r:ef-p
(transform e:ef-p)
;; start a simulation run
(restart r:ef-p)
```

Comparing the SES manner of constructing EF-P with the direct digraph-model approach of Chapter 5 we see that

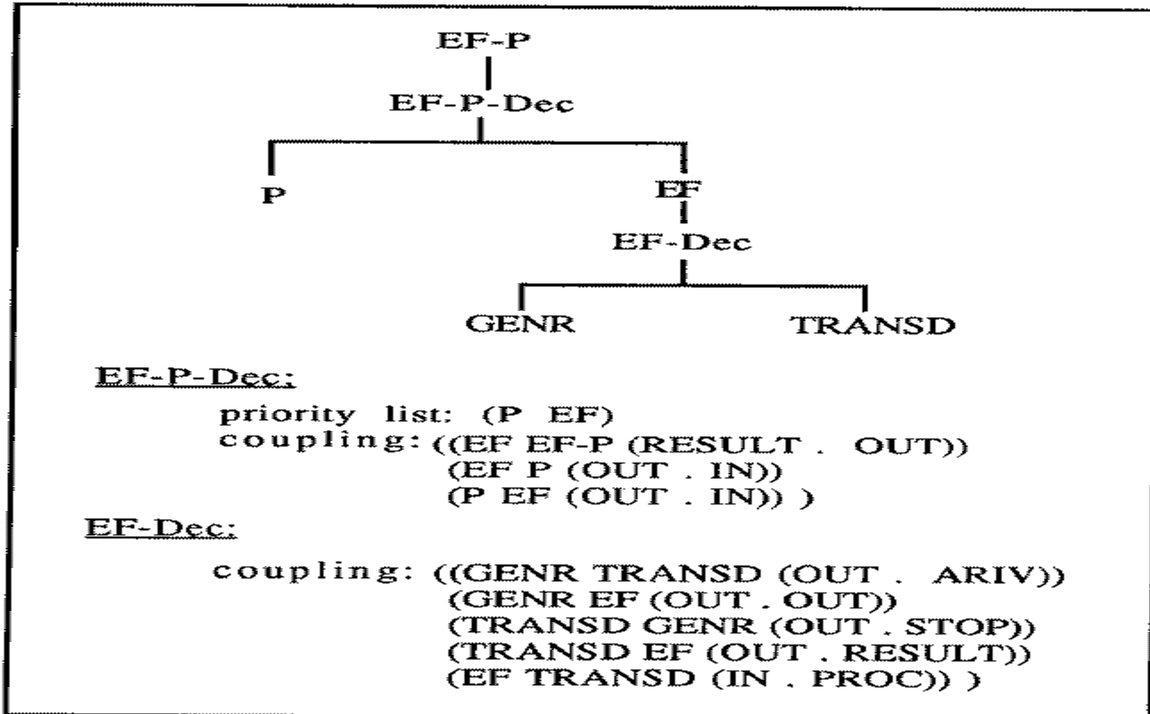


Figure 5: Extended SES for processor/frame pair

- it the SES is declarative, while the digraph-model specification is procedural, in character, and
- the SES relieves the modeller of specifying many of the details required by the digraph-model that can be inferred from other information(c.f., the coupling specification).
- the transformation process, besides synthesizing the model, also can relieve the user of associated actions (namely, creating, naming and initializing a root-co-ordinator.)

However, the SES provides much more power to specify and organize models as we shall now show.

3 SYSTEM ENTITY STRUCTURE ORGANIZATION of MODEL BASES

The various computer architecture models and model/frame pairs synthesized earlier in Chapter 6, are all built from the atomic-models shown in Figure 7.6. Files describing these models are the only ones that actually must be present in the model base directory for the system entity structure to direct synthesis of all of the hierarchical models.

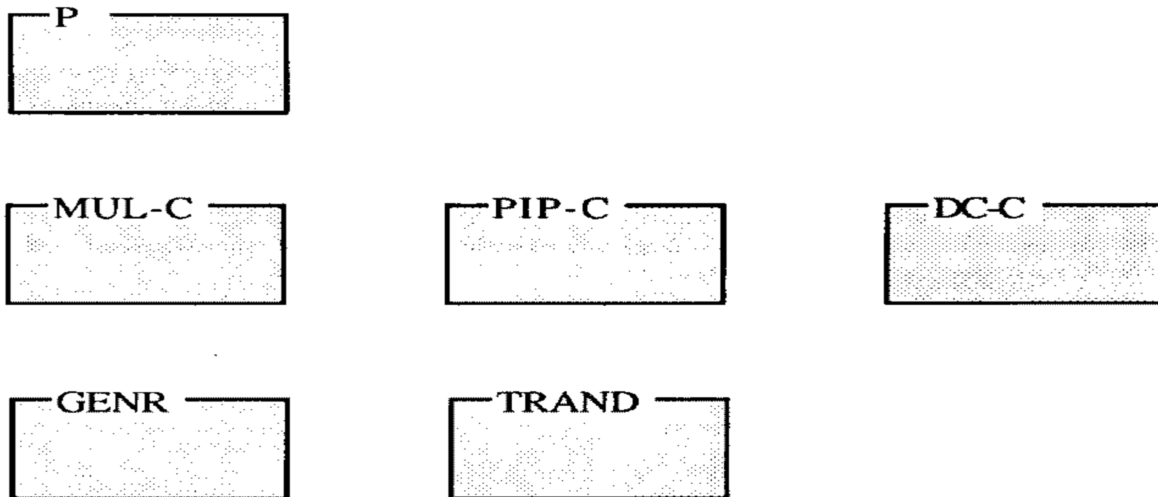


Figure 6: Figure 7.6: Model base for simple architecture study

In general, files for all the atomic entities in the SES must exist in the model base with the following exception:

- entities with names ending in numbers or having a "&" in the name.

The transform procedure treats names ending in numbers or having an "&" as instance names for an underlying "class" corresponding to the first part of the name. For example, the entities P1, P2, P3, P&DIV, and P&CMPL are all transformed into models which are copies of the model P (including its initial state). In this example, P is called the *base-name* and the file *base-name.m* should be in the model base. Alternatively, the base-name model must be constructable by the transformation procedure to be discussed soon. Note however that no explicit class definition corresponding to base-name model is required.

The organization of the directories for a model domain takes the following form:

scheme directory:	\scheme
DEVS-Scheme directory:	\scheme\devs
domain directory:	\scheme\devs\simparc
model base directory:	\scheme\devs\simparc\mbase
entity structure directory:	\scheme\devs\simparc\enbase

Each of the top three directories contains a scheme initialization file (scheme.ini) which respectively, loads the PCScheme system, loads the DEVS-Scheme system, and informs DEVS-Scheme of the model domain directory and its subdirectories. The model and entity structure directories are assumed to be subdirectories of the domain directory with names, *mbase* and *enbase*, respectively. These directories can be changed with the DEVS-Scheme command (*change-dir*). Global variables

model_base-directory and *entstr_base-directory* hold the path information to the respective subdirectories.

To start up ESP-Scheme, the entity structure layer of DEVS-Scheme, the user has only to set the current directory to the domain directory and call the extended or expanded version of PC-Scheme. For example:

⟩cd \scheme\devs\simparc
⟩pcsext

When loading has finished, the variable, *model_base-directory* is bound to the string “\\scheme\\devs\\simparc\\m
and the variable, *entstr_base-directory* is bound to the string “\\scheme\\devs\\simparc\\enbase\\”(note: / may be used instead of \\).

3.1 SES/Model Base for Simple Computer Architectures

Files corresponding to the models of Figure 7.6 are placed in the model base directory:

model base directory: \scheme\devs\simparc\mbase

files:

p.m
mul-c.m
pip-c.m
dc-c.m
genr.m
transd.m

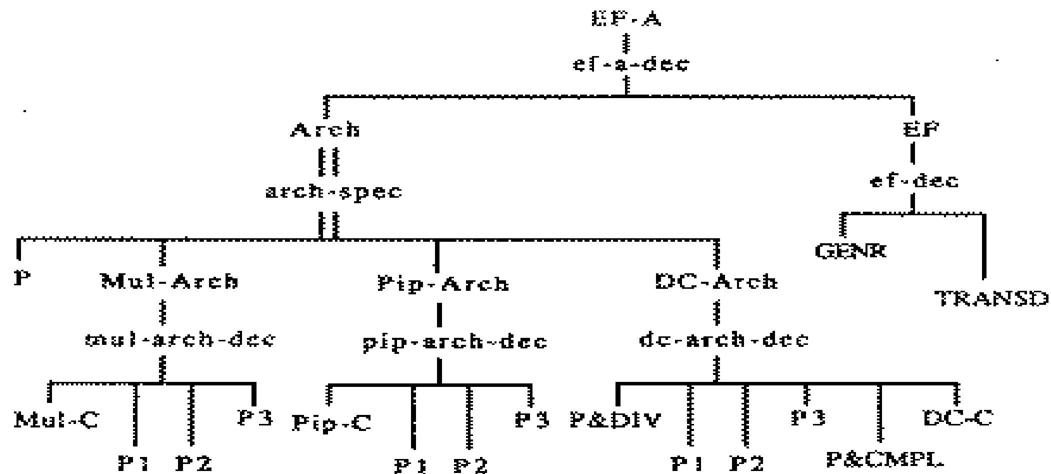
An SES for this model domain is shown in Figure 7.7 and specified in file ef-a.s which is contained in the entity structure base. After ef-a.s is loaded the first time, the compiled form of the SES it specifies, is stored in the entity structure directory as ef-a.e. Hence, we have:

entity structure directory: \scheme\devs\simparc\enbase

files:

ef-a.s
ef-a.e

The SES of Figure 7.7 makes it clear that the same experimental frame is to be used with each of the four architectures under study. To see this, notice that the model/frame combination is represented by the root entity, EF-A. The components of this combination are specified through an aspect, EF-A-DEC, which has as components the frame, EF and an entity, ARCH. ARCH generically represents the four architectures since it has a specialization, ARCH-SPEC which contains the architectures as variants.



EF-A-DEC: priority list: (EF ARCH)
coupling: ((EF EF-A (RESULT . OUT)) (EF ARCH (OUT . IN))
(ARCH EF (OUT . IN)))

EF-DEC: coupling: ((GENR TRANSD (OUT . ARIV)) (GENR EF (OUT . OUT))
(TRANSD GENR (OUT . STOP)) (TRANSD EF (OUT . RESULT))
(EF TRANSD (IN . PROC)))

MUL-ARCH-DEC:
priority list: (P1 P2 P3 MUL-C)
coupling: ((P3 MUL-C (OUT . Y3)) (P2 MUL-C (OUT . Y2))
(P1 MUL-C (OUT . Y1)) (MUL-C P3 (X3 . IN))
(MUL-C P2 (X2 . IN)) (MUL-C P1 (X1 . IN))
(MUL-C MUL-ARCH (OUT . OUT)) (MUL-ARCH MUL-C (IN . IN)))

PIP-ARCH-DEC:
priority list: (P3 P2 P1 PIP-C)
coupling: ((P3 PIP-C (OUT . Y3)) (P2 PIP-C (OUT . Y2))
(P1 PIP-C (OUT . Y1)) (PIP-C P3 (X3 . IN))
(PIP-C P2 (X2 . IN)) (PIP-C P1 (X1 . IN))
(PIP-C PIP-ARCH (OUT . OUT)) (PIP-ARCH PIP-C (IN . IN)))

DC-ARCH-DEC:
priority list: (P&CMPL P3 P2 P1 P&DIV DC-C)
coupling: ((P&CMPL DC-C (OUT . CY)) (P&DIV DC-C (OUT . PY))
(DC-C P&CMPL (CX . IN)) (DC-C P&DIV (PX . IN))
(P3 DC-C (OUT . Y3)) (P2 DC-C (OUT . Y2))
(P1 DC-C (OUT . Y1)) (DC-C P3 (X3 . IN))
(DC-C P2 (X2 . IN)) (DC-C P1 (X1 . IN))
(DC-C DC-ARCH (OUT . OUT)) (DC-ARCH DC-C (IN . IN)))

Figure 7: SES for simple architectures study

The coupling between ARCH and EF is attached to the aspect, EF-A-DEC. In pruning, when a particular specialized entity is selected from ARCH-SPEC, this particular entity replaces all references to ARCH in the coupling specification. Thus, the general entity, ARCH, can be viewed as a kind of place holder for any of its specialized versions in the coupling specification. For example, if P is selected as the specialized entity from ARCH-SPEC, the coupling between EF and P becomes:

coupling → ((EF EF-A (RESULT . OUT))
(EF P (OUT . IN))
(P EF (OUT . IN))).

We see that this is the same as the coupling established earlier when P was coupled to EF in the digraph model EF-P.

Clearly, for ARCH to validly represent its specialized versions, the coupling between each of the versions and the experimental frame must be isomorphic, i.e., essentially the same except for replacement of one variant for another. Fortunately, this is the case here. Indeed, the frame was designed so that it could consistently be coupled with any job processing model to measure average turnaround time and throughput.

The priority order attached to an aspect is treated similarly to the attached coupling by the pruning procedure. Thus when P is selected for ARCH from ARCH-SPEC, the priority list attached to EF-A-DEC becomes

priority-list: (P EF).

C++

```
\\efa_ses.h

#include "ses.h"

class efa_ses:public ses{

public:

efa_ses():ses("efa"){
ent * efa = make_ent("efa");
set_root(efa);

asp * efa_dec = make_asp("efa_dec");
add_asp_to_ent(efa_dec,efa);

ent * ef = make_ent("ef");
ent * arch  = make_ent("arch");
```

```
add_ent_to_asp(ef, efa_dec);
add_ent_to_asp(arch, efa_dec);

add_couple(ef,efa,"result","out");
add_couple(ef,arch,"out","in");
add_couple(arch,ef, "out","in");

spec * arch_spec = make_spec("arch_spec");
add_spec_to_ent(arch_spec,arch);

ent * p = make_ent("p");
ent * mul_arch = make_ent("mul_arch");
ent * pip_arch = make_ent("pip_arch");
ent * dc_arch = make_ent("dc_arch");

add_ent_to_spec(p,arch_spec);
add_ent_to_spec(mul_arch,arch_spec);
add_ent_to_spec(pip_arch,arch_spec);
add_ent_to_spec(dc_arch,arch_spec);

}
};
```

3.2 Pruning the System Entity Structure

Recall that a pure entity structure is one having no specializations and at most one aspect hanging from every entity. *Pruning* is required to create a such a pure SES. The result of pruning is a *Pruned Entity Structure (PES)* which contains fewer aspects and specializations than the original and therefore specifies a smaller family of alternative models than the latter. Ultimately, pruning terminates in a pure entity structure which specifies the synthesis of a particular hierarchical model. We employ the pruner by issuing the command (*prune es*) for a desired entity structure *es*.

For example, let us prune the entity structure of Figure 7.7 to construct a model/frame pair suitable for experimenting with the divide & conquer architecture. Then we issue the command:

(*prune* e:ef-a),

and follow the interaction shown in Figure 7.8.

C++:

```
//efa_pes@dc.C

#include "efa_ses.h"
```

```
1)   give extension for pruned-entstr name :
**  dc
2)   select starting entity from the following: (GENR TRANSD P&CMPL
     P&DIV DC-C MUL-C P3 P2 P1 PIP-C P DC-ARCH MUL-ARCH PIP-ARCH EF
     ARCH EF-A)
**   ef-a
     working from entity EF-A
3)   make this a leaf? (y/n)
**   n
4)  select an aspect from the following: (EF-A-DEC)
    aspect EF-A-DEC selected
    working from entity ARCH
5)  select a specialization from the following: (ARCH-SPEC)
    specialization ARCH-SPEC is selected
6)  select an entity from the following: (P PIP-ARCH
    MUL-ARCH DC-ARCH)
**   dc-arch
   entity DC-ARCH from specialization ARCH-SPEC selected
7)   select an aspect from the following: (DC-ARCH-DEC)
   aspect DC-ARCH-DEC selected
   working from entity DC-C
   working from entity P1
   working from entity P2
   working from entity P3
   working from entity P&DIV
   working from entity P&COM

   working from entity EF
8)   make this a leaf? (y/n)
**   n
   select an aspect from the following: (EF-DEC)
   aspect EF-DEC selected
   working from entity TRANSD
   working from entity GENR

9) save this entity-structure? (y/n)
** y
  Pruned entstr P:ef-a-dc made.
```

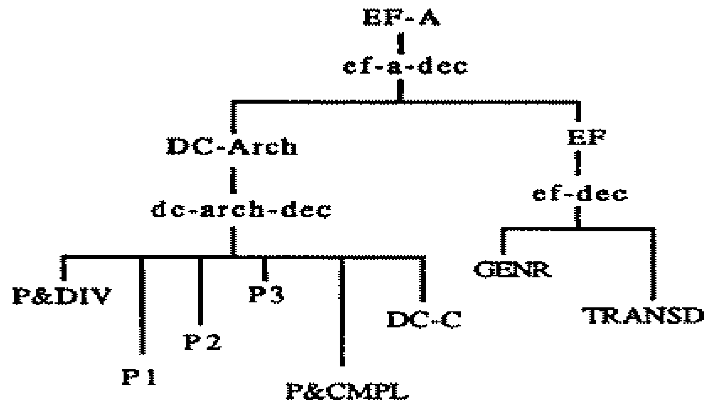
Figure 8: Transcript of a pruning interaction to construct divide-and-conquer model

```
main(){
efa_ses * pes = new efa_ses();
cout << "BEFORE PRUNING" << endl;
pes ->print();

pes ->prune_spec("arch","arch_spec","dc_arch");
cout << "AFTER PRUNING" << endl;

pes->print();
pes->transform("efa_pes@dc");
}

g++ efa_pes@dc.C   -lcon;a.out
//efa_pes@dc_comp.C

#include <stream.h>
#include <devshead.h>

main()
{
efa * efa1 = new efa("efa1");
dc_arch * dc_arch1 = new dc_arch1("dc_arch1");
ef * ef1 = new ef("ef1");
efa1->add(dc_arch1);
efa1->add(ef1);
}

g++  efa_pes@dc_comp.C  -lcon -o efa_pes@dc;
```

First the pruner requests an extension to distinguish this particular pruning from others (line 1). The starting entity for pruning is then requested (line 2). The pruned entity structure (PES) that results will have as its root the chosen starting entity. This PES will have the name of the starting entity with the above extension suffixed to it. Note that since the starting entity can be chosen by the user, models of components or subsystems of the the overall system, may be constructed by pruning.

The next choice (line 3, and later line 8) is whether the current entity is to be made a leaf or not. Making an entity a leaf has the effect of terminating subsequent pruning of its subtree and requires that a model for the entity be accessible to the transform procedure. This is appropriate for example, if we want to employ a simplified model for the current entity rather than the more elaborate one that would result from synthesis directed by the subtree.

A single aspect must be chosen for the current entity from those available (line 4). Since, in this case, only one aspect, EF-A-DEC, is available, the pruning procedure automatically selects it.




```
EF-A-DEC:  priority list: (EF DC-ARCH)
           coupling:  ( (EF EF-A (RESULT . OUT)) (EF DC-ARCH (OUT . IN))
                        (DC-ARCH EF (OUT . IN)) )

EF-DEC: coupling:  ( (GENR TRANSD (OUT . ARIV)) (GENR EF (OUT . OUT))
                     (TRANSD GENR (OUT . STOP)) (TRANSD EF (OUT . RESULT))
                     (EF TRANSD (IN . PROC)))

DC-ARCH-DEC:
     priority list: (P&CMPL P3 P2 P1 P&DIV DC-C)
     coupling:  ( (P&CMPL DC-C (OUT . CY)) (P&DIV DC-C (OUT . PY))
                  (DC-C P&CMPL (CX . IN)) (DC-C P&DIV (PX . IN))
                  (P3 DC-C (OUT . Y3)) (P2 DC-C (OUT . Y2))
                  (P1 DC-C (OUT . Y1)) (DC-C P3 (X3 . IN))
                  (DC-C P2 (X2 . IN)) (DC-C P1 (X1 . IN))
                  (DC-C DC-ARCH (OUT . OUT)) (DC-ARCH DC-C (IN . IN)) )
```

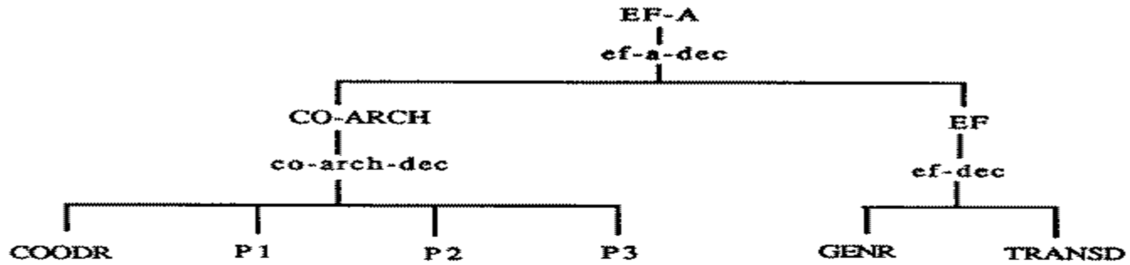
Figure 9: PES resulting from pruning interaction of Figure 7.8

Pruning proceeds in a depth-first traversal of the SES. For example, the first child of aspect EF-A-DEC, ARCH, is visited next. If more than one specializations are hanging under the current entity, the user is queried to select one (line 5). If only one exists, we proceed to make a selection from it (line 6).

Pruning in this manner continues until all leaf entities (which remain after subtree abortion) have been visited.

Figure 7.9 shows the PES resulting from the above pruning. When transformed, this SES will result in a digraph model which couples the divide and conquer architecture with the experimental frame. Notice that the coupling associated with the aspect ef-a-dec has automatically been modified to reflect the replacement of the generic entity ARCH by the specialized choice DC-ARCH. In general, replacement of a generic entity by its selected specialized entity is made in all contexts in which the generic entity appears.

To construct a simulation model ready for execution we need only issue the command (*transform p:ef-a@dc*). Should we desire to construct the same model in subsequent sessions, we need




```
EF-A-DEC:  priority list: (EF  CO-ARCH)
           coupling:  ( (EF EF-A (RESULT . OUT))
                        (EF CO-ARCH (OUT . IN))
                        (CO-ARCH EF (OUT . IN)) )

CO-ARCH-DEC: priority list: (P1  P2  P3  COORD)
             coupling:  ( (P3 COORD (OUT . Y3))
                          (P2 COORD (OUT . Y2))
                          (P1 COORD (OUT . Y1))
                          (COORD P3 (X3 . IN))
                          (COORD P2 (X2 . IN))
                          (COORD P1 (X1 . IN))
                          (COORD CO-ARCH (OUT . OUT))
                          (CO-ARCH COORD (IN . IN)) )
```

Figure 10: An alternative SES for the simple architecture

only load the PES P:EF-A@DC and issue the same transform command.

3.3 Alternative SES for Simple Architectures Domain

An alternate SES for the simple architectures model domain is shown in Figure 7.10. Notice that in this case the four architectures of interest are not listed individually under the ARCH-SPEC specialization as they are in Figure 7.7. Instead, a common structure for the co-ordinated architectures is described under the specialized entity, CO-ARCH. The decomposition CO-ARCH-DEC specifies this common structure consisting of a co-ordinator, CO-ORD and three subordinate processors, P1, P2, and P3. The fact that the co-ordinator is different in each case is reflected in the specialization, COORD-SPEC which contains the three types: MUL-C, PIP-C, and DC-COORD.

This SES is superior to the original one in Figure 7.7 in two ways:

1. The specification is more compact, requiring less information to be provided
2. It makes explicit the common features of the co-ordinated architectures, leading to better comprehension.

The coupling between the general entity COORD and the subordinate processors is attached to the enclosing aspect CO-ARCH-DEC. When COORD is replaced by any of its specialized entities, the coupling is specialized accordingly. Thus, although each of the co-ordinator alternatives manages the subordinate processors differently, the coupling between co-ordinators and subordinates is essentially the same (isomorphic) in each case.[1]

The divide and conquer co-ordinator is this case is actually a digraph model containing the co-ordinator DC-C proper and its helpers, the partitioner and compiler: P&DIV and P&CMPL. An entity structure for DC-COORD is shown in Figure 7.11. It is shown as a pruned entity structure P:DC-COORD in order to make it available to the transform procedure, as described at the end of Chapter 2. (This PES was derived from the original DC-ARCH model by a process called deepening which we will discuss later.) The entity structure directory in this case contains the file dc-coord.p for the model DC-COORD as well as the file ef-a.e for the overall SES. Thus we have:

entity structure directory: \scheme\devs\simparc\enbase

files:

ef-a.s
ef-a.e
dc-coord.p

In general, *.e files describe system entity structures (SES) while *.p files describe pruned entity structures (PES). When such files are loaded into RAM, the structures they describe are given names prefixed by e: and p:, respectively. Thus after loading, ef-a.e and dc-coord.p, the entity structures E:EF-A and P:DC-COORD exist in RAM.

4 OPERATIONS ON HIERARCHICAL MODEL STRUCTURES:

4.1 FLATTENING AND DEEPENING

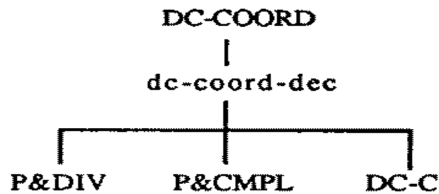
To create a common structure for the co-ordinated architectures we had to group the divide & conquer co-ordinator, DC with its helpers, the partitioner, P&DIV and compiler, P&CMPL to form a new component, DC-COORD. This is an example of a structure modifying operation called *deepening*. A procedure, *deep-devs*, is provided which automatically does the desired restructuring in such a way that behavioral equivalence is preserved. To group the components DC, P&DIV and P&CMPL together, we use the command:

(*deep-devs* dc-arch '(dc p&div p&cmpl) 'dc-coord 'digraph-models)

where *deep-devs* takes as arguments:

- the name of the model some of whose components are to be grouped (dc-arch),

[1]The situation here is reminiscent of writing grammars for natural languages: a syntactic class, such as verbs, can be formed if it expresses a valid generality, namely that all members of that class are treated alike in the construction of sentences.



<u>DC-COORD-DEC:</u>

```
coupling: ( (P&CMPL DC-C (OUT . CY)) (P&DIV DC-C (OUT . PY))
            (DC-C P&CMPL (CX . IN)) (DC-C P&DIV (PX . IN))
            (DC-C P2 (X2 . IN)) (DC-C P1 (X1 . IN))
            (DC-C DC-COORD (OUT . OUT4)) (DC-C DC-COORD (X1 . OUT3))
            (DC-C DC-COORD (X2 . OUT2)) (DC-C DC-COORD (X3 . OUT1))
            (DC-COORD DC-C (IN4 . IN)) (DC-COORD DC-C (IN3 . Y1))
            (DC-COORD DC-C (IN2 . Y2)) (DC-COORD DC-C (IN1 . Y3)) )
```

Figure 11: PES for DC-COORD which includes Co-ordinator proper, partitioner and compiler

- the list of components to be grouped, (dc p&div p&cmpl),
- the name of the model to be constructed by grouping, (dc-coord),
- the class to which the new model will belong (digraph-models or a sub-class of it).

As shown in Figure 7.12, the effect of this operation is to restructure DC-ARCH so that a new level is introduced into its hierarchical structure. The couplings internal to the new digraph-model, DC-COORD are the original ones involving its components (Figure 7.11); new external input and output couplings are added so that the original connections to other components are preserved (Figure 7.10).

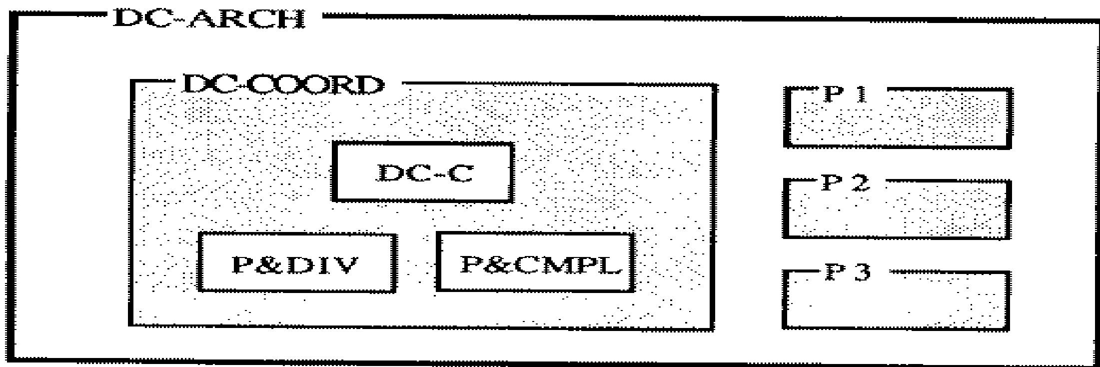
The inverse operation of deepening is *flattening*. Here a digraph model is removed from a larger model and its components (children) are coupled to their grandparent. For example,

(*flat-devs* dc-coord)

will remove DC-COORD from the model in which it is found, DC-ARCH, and reconnect its children to its parent in such a way that the original model, DC-ARCH, is restored. Clearly, the model to be removed cannot be the highest level digraph model of a hierarchical structure

To flatten a hierarchical model down to a single level, we can employ

(*flat-all dc-arch*)



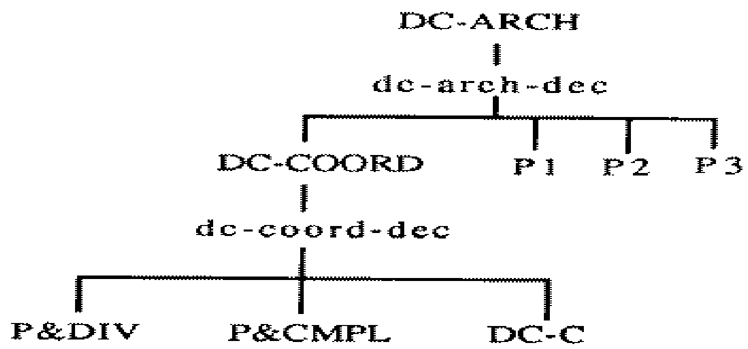


Figure 12: ARCH digraph-model and its equivalent deepened SES (coupling not shown

which repeatedly applies *flat-devs* until a completely flat model (digraph model with atomic children) is obtained.

Flattening and deepening are useful manipulations of a model structure. They produce alternative structures that are all behaviorally equivalent. Such alternative structures may, for example, have better properties in relation to execution speed on either a serial or parallel computer. Or as we have seen, these restructuring operations may be employed to facilitate a better system entity structure representation of the model base.

The command, *inverse-transform*, takes a coupled-model and writes a pure entity structure for it. Thus (*transform* (*inverse-transform* M)) reconstructs M. After flattening or deepening in the model domain, we can use *inverse-transform* to obtain an entity structure representing the result. We obtained the PES for DC-COORD by applying *inverse-transform* to it after deepening DC-ARCH.