CMU-CS-80-117

Well-Formed System Compositions

A. Nico Habermann

Dewayne E. Perry

March 1980

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-80-117

Well-Formed System Compositions

A. Nico Habermann

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Dewayne E. Perry

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
and
Pegasus Systems
125 Beechwood Road
Summit, NJ 07901

March 1980

Abstract

A Major part of an integrated programming and system development environment such as Gandalf is that which is concerned with describing systems and controlling versions of those systems. This paper investigates the foundations of systems' description and control. It presents a system description language, discusses the issue of composing system descriptions from component specifications, addresses the questions of consistency, ambiguity and completeness, delineates the basic rules for system compositions, and presents an algorithm to generate all viable system compositions. It concludes with a demonstration that the algorithm has the desired properties and a discussion of its theoretical and practical complexity.

This work is sponsored by the Software Enginneering Division of CENTACS/CORADCOM, Fort Monmouth, NJ.

1. Introduction.

This paper discusses the issue of composing viable system versions out of component specifications. It addresses questions of consistency, ambiguity and completeness of system descriptions. We are primarily concerned with functional aspects of system composition. For a discussion of the implementation aspects of system composition, see [Tichy 80] and [Cooprider 79]. The usefulness of this paper is in the application of its result to a software construction and maintenance system. Such a system is typically part of an integrated program and system development environment (such as Gandalf [Gandalf 79]). The programming environment can enforce some basic system composition rules and assist the user in generating system descriptions. Using the results of this paper, the environment will guarantee that users create only well-formed system compositions.

System development is a matter of two very different activities -- the conventional task of writing programs and the construction of system versions out of separate system components. Central to both is a precise specification of the facilities provided by a system component. For implementors, component specifications serve as a concise task description and state the desired goal which they must realize by choosing appropriate data representations and by writing efficient subprograms. For designers, component specifications define a unique user interface which shows how component facilities can be used and which components can be put together into working systems.

The important role played by system component specification is reflected in the Ada language. It makes a clear distinction between the *definition* of a module and its *body*. The definition part specifies the facilities that a module provides to its users. These facilities are definitions of data types, data objects or constant values, subprograms and modules. Completely separate from the definition part is the module body which determines the representation of data objects and the implementation of subprograms (see [Ada 79], Chapter 7].

In the Ada language each module has a unique implementation which is given in its body.

Hence, there is no need to name a particular implementation in a definition part in Ada, because there is only one single implementation. Such an arrangement is not realistic for systems because modifications, testing and various usage give rise to a multitude of versions of a single system. Hence we need descriptors that allow for the specification of multiple versions of components and systems.

In the first section we start out with a discussion and definition of system and component descriptors. In the next section we formulate and simplify a small set of rules which express some basic properties of system descriptions. We will demonstrate that the two sets of rules are equivalent. In the next section we prove some useful properties of system descriptions which can be derived from the basic rules. We introduce the notions of *proper* compositions, *conflict-free* compositions and *well-formed* compositions. The next section presents an algorithm that generates all minimal well-formed compositions for a given system description. The last section shows that the complexity of the algorithm is on the order of the number of subsets of the set of components but that this can often be substantially reduced.

We use the term "version" in a very general sense and not in a specific technical sense. We use distinct terms to indicate various kinds of versions of software components: e.g., source modules exist in a number of *implementations* and each implementation consists of a sequence of *revisions*, ordered by date. A system version is constructed by selecting subsets of its components. Such a subset is called a *composition*. Descriptions of compositions contain enough information for the programming environment to generate working versions of a system.

2. System Descriptions

Systems are described in terms of their components and the facilities which they provide and require. The provided facilities correspond to the facilities (types, objects, operations and modules) exported from the component. An example of the provided facilities can be found in the visible interfaces of Ada modules. The required facilities correspond to those facilities (types, objects, operations and modules) that must be imported by the component.

2.1. System Description Elements

There are two types of components from which systems are constructed -- *modules* and *systems*. At the system description level, the module is considered atomic and its provided and required facilities are *given* in the module descriptor by the system designer. However, at the programming level, these facilities are derived from the provided and required facilities of submodule components: each type definition implicitly provides the type name defined and requires the types used in the definition; each variable declaration implicitly provides the variable name and requires the type of the variable; subprograms implicitly provide the operation defined by the subprogram and require, again implicitly, the types of the formal parameters; additionally, a subprogram may explicitly require other facilities it needs. Because these subcomponent descriptions occur below the level of granularity desired in constructing systems, the system designer must integrate them into the module descriptor by hand by delineating their provisions and requirements.

Systems, on the other hand, are described in terms of module and system components. The description of the system is constructed so that the provide list is a subset of the facilities provided by the components while the require list is derived from the requirements of the components. Systems that require no facilities are complete stand-alone systems whereas systems that require facilities to be supplied externally are simply subsystems.

In both types of components, the provided facilities must be considered indivisible. If two components provide identical facilities, they conflict with each other and only one of the components can provide facilities required by a third component. Consider the following example. Components A and B both provide facilities f and g. If component C requires facilities f and g, one cannot select f from A and g from B to satisfy the requirements of C. Either A or B can provide the required facilities, but not both.

There are two aspects of system building that are important -- the sharing of components and the combining of components. Often different systems may have components in common. Just as it is safer to have one source of a component that is shared, it is good practice to

have one description of the component in the environment along with some means of designating its use within several system descriptions.

The *use* statement makes explicit the sharing of a component and corresponds in function to an import statement: it specifies that a particular component is to be made available within a system description and may then be used as a component in system compositions. In contrast, a facility required by a system is analogous to a formal parameter: the particular actual parameter will be supplied by other components in a composition and may be satisfied in any way that the system designer desires.

We provide two mechanisms for combining components. The first is the *system* descriptor. Components of a system are specified by module descriptors, system descriptors or use statements. From this set of system components, compositions can be constructed that satisfy the provide list of the system specification. The system descriptor, with its compositions as version specifications, welds the components into a single indivisible unit. Its use in constructing other systems is identical to that of the module descriptor.

The second mechanism for combining components is the *box* descriptor. It is similar to the system descriptor in that it provides an encapsulation mechanism that allows both the hiding and the sharing of components within a well-bounded and restricted scope. It is distinct from a system descriptor in that it does not provide facilities but exports components that should be viewed as (possibly mutually exclusive) alternative solutions. One or more of the exported components may be selected in the construction of systems. Whether they are mutually exclusive alternatives depends upon whether their individual sets of provided facilities conflict.

In short, a system provides facilities that must be selected together while a box exports components that may be selected alternatively as desired.

2.2. System Description Grammar

The grammar presented below contains four statements that provide the capabilities that we need to describe systems -- the *box*, *sys*, *mod* and *use* statements.

In the grammar descriptions, normal type is used for syntactic categories, bold face is used for keywords, square brackets indicate selection, and curly brackets indicate an optional field. Some syntactic categories are considered self-explanatory (e.g., name). Elipses have their usual meaning.

The box descriptor enables one to declare a box name, to define the system and module components that are exported and to describe the inner structure of the box in terms of boxes, systems and modules.

```
box ::= box name
            export [sysname, modname], . . . ;
            [box, sys, mod, use]; . . . ;
        end
```

For example,

```
box DB systems
      export DES, INS;
      sys DES  . . .  end
      sys INS  . . .  end
end
```

In the sys construct the name of the system is declared, the provide list selected as a subset of all the facilities provided by the components and the require list constructed from the requirements of the components not satisfied by provided facilities. The internal structure of the system is described by component descriptions. Compositions are then specified using these components.

```
sys ::= sys name
            prov fac, . . . ;
            {req fac, . . . ;}
            [sys, mod, use]; . . . ;
            composition; . . . ;
```

```
        end
```

For example,

```
sys DES
      prov RE, RU, RR, RT, RP;
      req Exec;
      mod RMGT  . . .  end
      . . .
end
```

The mod descriptor defines the module name, the facilities provided and required by the module, and the implementations of the module.

```
mod ::= mod name
           prov fac, . . . ;
           {req fac, . . . ;}
           implementation; . . . ;
        end
```

For example,

```
mod RFA
      prov getreport, putreport
      req RFIO, SVCs
      . . .
end
```

The use statement specifies that a particular system or module known in the environment is to be used in this system description.

```
use ::= use [sysname, modname]
           prov fac, . . . ;
           {req fac, . . . ;}
```

For example,

```
use RFA  . . . ;
use DES  . . . ;
```

Compositions specify collections of components that form usable system versions which satisfy the provide clause of the system specification and contain no requirements other than those required by the system. The programming environment will insure that the composition

is well-formed. A composition is named and defines the components to be included in the composition.

composition ::= **comp** name = component name, . . . ;

For example,

comp A = RFA, VIDEO.BCRT, EXEC.MM, . . . ;
comp B = RTH, INQUIRY, . . . ;

Composition element names are either qualified or unqualified. In the first case, the qualification is in the form of an implementation name for a module or a composition name for a system. In the second case, some standard default determination must be used (e.g., the most recently released system composition or module implementation).

Implementations name particular implementation versions and provide suitable information to derive or to find the appropriate module. For our purposes here, we require only that implementations be named.

implementation ::= **impl** name = . . . ;

For more complete details about compositions and implementations, see [Cooprider, Tichy].

2.3. Scope Rules

We impose the following scope rules and conventions upon the description of systems.

1. Several systems may not use each other recursively, e.g., " **use** A . . . ;" occuring inside system B and "**use** B . . . ;" occuring inside system A leads to the undesirable consequence of a system using itself.

2. Compositions may be constructed only with components known within the system description through mod or sys, or use statements.

3. Facilities may be referenced individually by the name of a particular facility or collectively as a set of provided facilities by the component name.

3. Basic System Construction Rules

Let the set of components of a system S be $\{C_1, \ldots, C_m\}$ and let $M = \{1, \ldots, m\}$ be the set of component indices. The set of facilities provided by a component C_i is denoted by p_i and that provided by the system as a whole by p(S). Sets of required facilities are similarly denoted by r_i and r(S). Any one of these, or all, may be empty.

In the preceding section, module descriptors were introduced as the atomic building blocks of system descriptions. It is therefore impossible to derive the lists of provided and required facilities of a module automatically. However, the provide and require lists for each module descriptor can be checked for contradictions. A contradiction arises if a facility mentioned in the provide list of a module also occurs in the require list of that module. We require that all module descriptors are *free of contradictions*. This is expressed by the basic rule:

$$p(\text{module}) \cap r(\text{module}) = \emptyset \qquad (1)$$

The purpose of including a particular component in a system is that it provides some facility that is required by another component or one that is exported by the system through its provide list. We require that the set of components of a system be *complete* in the sense that all facilities provided by a system are provided by at least one of its components. This is expressed by the rule:

$$p(S) \subseteq \bigcup_{i=1}^{m} p_i \qquad (2)$$

This rule states that a system cannot magically provide some facility from nowhere. However, it permits information hiding because it is not necessary that *all* facilities provided by the components of a system be included in its provide list.

The facilities required by a module must be explicitly indicated by the system designer. The require list of a system, on the other hand, can be derived automatically from the provide and require lists of its components. It is obvious that a system requires exactly those facilities which are required by one or more of its components but are not provided by any

of its components. This is expressed by the rule:

$$r(S) = \bigcup_{i=1}^{m} r_i - \bigcup_{i=1}^{m} p_i \tag{3}$$

This rule expresses the *external dependency* of a system on modules and systems other than its components.

Rule 1 is also valid for systems.

Theorem 1: No facility provided by a system is also required by that system,

or

$$p(S) \cap r(S) = \phi$$

Proof

If $r(S) = \phi$, the statement is true. Otherwise, let $f \in r(S)$. Rule 3 implies $f \notin \bigcup_{i=1}^{m} p_i$ which, with rule 2, implies $f \notin p(S)$. Thus, $r(S) \cap p(S) = \phi$.

*****

Corollary: $r_i \cap p_i = \phi$ for every component C_i because a component is either a module or a system.

It seems that rule 3 is somewhat too strong. We may refine it by saying that a facility f required by a component C_i is included in $r(S)$ if none of the components other than C_i provide that facility. Let $\bar{p}_j = \bigcup_{i=1}^{m} (p_i - p_j)$ be the complement of p_j. Facility $f \in r_j$ is included in $r(S)$ if $f \notin \bar{p}_j$. This refinement leads to a rule which replaces rule 3.

$$r(S) = \bigcup_{i=1}^{m} (r_i - \bar{p}_i) \tag{4}$$

Theorem 1 can also be derived from the basic rules if rule 3 is replaced by rule 4.

Alternative Proof of Theorem 1

If $r(S) = \phi$ or $p(S) = \phi$ the statement is true. Otherwise, let $f \in r(S)$; use(4): $(\exists i \in M)\ f \in r_i$. Suppose the components are reordered such that

$$f \in r_i \text{ for } r \in \{1, \ldots, k\},\ f \notin r_i \text{ for } i \in \{k+1, \ldots, m\} \tag{5}$$

where $1 \le k \le m$. If $k = m$, f is an element of all r_i.

Suppose $f \in p(S)$; use (2): $(\exists j \in M)\ f \in p_j$.

<u>case 1</u>: $j \in \{k+1, \ldots, m\}$.

Define $R := \bigcup_{i=1}^{k} (r_i - \bar{p}_i)$ and $R' := \bigcup_{i=k+1}^{m} (r_i - \bar{p}_i)$.

$(i \in \{1, \ldots, k\})\ p_j \in \bar{p}_i$, because $\bar{p}_i = \bigcup_{l=1}^{m} p_l - p_i$ and $i \le k < j$.

$$\text{Hence, } (i \in \{1, \ldots, k\})\ f \in p_i, \text{ so } f \notin R. \tag{6}$$

Also, $f \notin R'$ because of (5). This implies that $f \notin r(S)$, because $r(S) = R \cup R'$ (see 4).

<u>case 2</u>: $j \in \{1, \ldots, k\}$.

In this case there is a component C_j which has a facility f in both its provide and require list. Component C_j cannot be a module because of (1). Thus, if all components are modules, it is not possible that $f \in p(S)$ and $f \in r(S)$ are both true. If C_j is a system, we apply the proof recursively to that system. The nesting rules exclude circularity in system definitions. Since the total number of module and system descriptors is finite, the proof is eventually applied to a system that has only modules for components.

***

Using theorem 1 we can easily show the equivalence of rules 3 and 4. Since $p_i \cap r_i = \phi$ is true for all i, it follows that

$$r_i - \bar{p}_i = (r_i - p_i) - \bar{p}_i = r_i - p_i \cup \bar{p}_i = r_i - \bigcup_{j=1}^{m} p_j$$

Thus,

$$\begin{aligned} r(S) &= \bigcup_{i=1}^{m} (r_i - \bar{p}_i) \\ &= \bigcup_{i=1}^{m} (r_i - \bigcup_{j=1}^{m} p_j) \\ &= \bigcup_{i=1}^{m} r_i - \bigcup_{i=1}^{m} p_i \end{aligned}$$

Theorem 2: The facilities required by a component are either provided by another component or are required by the system. That is,

$$f \in r_i \Rightarrow f \in p_i \ \underline{\text{xor}}\ f \in r(S)$$

Proof

A. $f \in p_i$ @ $f \in \bigcup_{i=1}^{m} p_i$ @ f ß $r(S)$ (see rule 3). This means that no facility is both provided by some component and required by the system as a whole.

B. Let $f \in r_i$ $(i \in M)$ and suppose there is an index $j \in M$ such that $f \in p_j$. $j \neq i$, because $f \in r_i$ and $f \in p_i$ is in contradiction to theorem 1. If $j \neq i$, $f \in p_j$ implies $f \in p_i$.

C. Let $f \in r_i$ $(i \in M)$ and suppose for all indices $j \in M$ $f \notin p_j$. In this case $f \notin \bigcup_{j=1}^{m} p_j$, but $f \in \bigcup_{i=1}^{m} r_i$. This implies $f \in r(S)$ (see 3).

*****

The relationship between facilities provided by a system and its components is not as strict as that between required resources. Rule 2 allows that some component provides a facility that is not also provided by the system and rules 3 or 4 do not preclude that a component provides a facility that is required by no other component. All possible situations are shown in the following example.

Example

```
sys S prov a , b , x req z

    mod A prov a , x , y req v . . . end
    mod B prov b , v req x , z . . . end

    comp dflt = (A , B)
end
```

$p(S) = \{a , b , x\}$ $\qquad \bigcup_{i=1}^{m} p_i = \{a , b , x , y , v\}$

$r(S) = \{z\}$ $\qquad \bigcup_{i=1}^{m} r_i = \{v , x , z\}$

Facilities 'a' and 'b' are examples of ones that are in p(S), but are required by no component. Facility 'v' is one that is not in p(S) but is required by some component. Facility 'y' is an example of a superfluous facility: it is not included in p(S) and it is required by no component.

4. Basic Composition Rules

A system composition specifies a subset of the components of that system. Each system composition describes a different way to realize that system. Different compositions may incorporate different versions of the same component or different components that provide similar facilities. For simplicity, we denote a system S as a set of components $\{C_1, C_2, \ldots, C_m\}$ and we denote a composition as a set of indices $\{i_1, i_2, \ldots, i_k\}$, where $i_j \in M$. We use p(set) as an abbreviation for $\bigcup_{i \in set} p_i$. COMPOS(S) denotes the set of indices of the components in the composition of S.

We first require that a composition be made from components included in the system description.

$$(j \in \text{COMPOS}(S))\ j = k \text{ for some } C_k \in S \tag{7}$$

It is natural to find several components within a system that provide the same facilities. These components may be included because they present different algorithms or different representations for these common facilities. However, components providing the same facilities should not be used together in a composition because this gives rise to a conflict similar to that of declaring the same variable name twice in one scope. We require, therefore, that compositions be *conflict-free*.

$$(i,j \in \text{COMPOS})\ i \neq j \Rightarrow p_i \cap p_j = \phi \tag{8}$$

Since a composition contains a subset of the set of components of a system, we want to guarantee that the facilities provided by the system are also provided by the composition. A composition satisfying this property is *self-sufficient*.

$$\text{A composition is self-sufficient} \Longleftrightarrow p(S) \subseteq p(\text{COMPOS}) \tag{9}$$

We also wish to guarantee that the facilities required by the composition are satisfied (i.e., provided) either by the composition itself or are facilities required by the system. Such a composition is considered *self-contained.*

A composition is self-contained $\Longleftrightarrow$ (10)
$r(COMPOS) - r(S) \subseteq p(COMPOS)$

A composition that is both self-sufficient and self-contained is a *proper* composition. If it lacks either of these two properties, it is an *improper* composition.

A composition is proper $\Longleftrightarrow$ (11)
it is both self-sufficient and self-contained

From these rules we can derive the following theorem: the set of facilities required by a proper composition and provided by the system are included in the set of facilities provided by that proper composition and required by the system.

Theorem 3: A composition is proper $\Longleftrightarrow r(COMPOS) \cup p(S) \subseteq p(COMPOS) \cup r(S)$

Proof

$\Leftarrow$ Assume the antecedant to show that COMPOS is proper

1. We know that $p(S) \cap r(S) = \phi$ by theorem 1. Hence, p(S) must be included in p(COMPOS) since it cannot be included in r(S). Thus, COMPOS is self-sufficient.

2. By subtracting r(S) from both sides of the assumption, we have $(r(COMPOS) \cup p(S)) - r(S) \subseteq p(COMPOS)$. Since, by theorem 1, r(S) does not subtract any element from p(S), we know that $r() - r(S) \subseteq (r(COMPOS) - r(S)) \cup p(S)$. Thus, $r(COMPOS) - r(S) \subseteq p(COMPOS)$ and COMPOS is self-contained (by rule 10).

3. By 11, COMPOS is proper.

$\Rightarrow$ Assume that COMPOS is proper and that $f \in r(COMPOS)$ or $f \in p(S)$.

1. Assume that $f \in r(COMPOS)$. Since COMPOS is proper, it is self-contained (by rule 4). By rule 4, $r(COMPOS) - r(S) \subseteq p(COMPOS)$. By adding r(S) to both sets, we have $r(COMPOS) \subseteq p(COMPOS) \cup r(S)$. And hence, by the assumption, $f \in p(COMPOS) \cup r(S)$.

2. Assume $f \in p(S)$. By rule 4, since COMPOS is proper, COMPOS is self-sufficient. By rule 4, $p(S) \subseteq p(COMPOS)$ and by the assumption, $f \in p(COMPOS)$. Thus, $f \in p(COMPOS) \cup r(S)$.

*****

We have shown the properties that are required of a composition to be proper. We would like to know whether a proper composition exists for any system.

Theorem 4: Every system has a proper composition.

Proof

Let COMPOS = (1,2, . . . ,m)

1. $p(S) \subseteq \bigcup_i p_i = p(COMPOS)$ by rule 2. Hence, by rule 9, COMPOS is self-sufficient.

2. $r(COMPOS) \subseteq p(COMPOS) \cup r(S)$ by theorem 2. If we subtract r(S) from each set, then $r(COMPOS) - r(S) \subseteq p(COMPOS)$. Hence, COMPOS is self-contained.

3. Thus, by rule 11, COMPOS is proper.

*****

The proper composition that is always constructable from a system description may not be very interesting. In fact, it may not even be usable as a "real" composition because it may have conflicting components. However, we can show constructively that given a composition of S that is proper but contains conflicts, it is possible to derive a system S' such that its "standard" proper composition is also conflict-free.

Theorem 5: If a composition CS of system S has a conflict, we can derive a system S' from S by removing the conflicts from the components so that CS' is conflict-free.

Proof

Let the composition CS = {1,2, . . . ,k}, where $k \leq m$ and m is the number of components in S, have conflicts. Take the first facility provided by the first element of the composition, $f_{11} \in p_1$; replace all C_j for which $f_{11} \in p_j$ by C'_j such that $p_{j'} = p_j - \{f_{11}\}$. Do this for each facility of each element in the composition. S' consists of the components from which all conflicting facilities have been removed. We note that no facility has been removed from p(S) or from p(CS). Therefore, if CS was a proper composition with conflicts, then CS' is also a proper composition but is conflict-free.

*****

Compositions that are both conflict-free and proper are *well-formed*. A system that has only conflict-free compositions is considered a conflict-free system; a system that has only well-formed compositions is considered a well-formed system.

In the complete set of well-formed compositions for a given system S, there exists a (possibly empty) subset of compositions that contain useless elements, i.e., elements that neither provide any required facilities of other elements in the composition nor provide facilities provided by the system. We are not interested in these superflous compositions. We are interested only in those compositions that contain the minimum number of elements to provide facilities that satisfy the provide list of the system and the requirements (less the requirements of the system) of the elements in the composition. These useful system compositions are *minimal well-formed* (MWF) compositions.

A composition C is minimal well-formed $\Longleftrightarrow$ C is well-formed and $(S \subseteq C)$ C - S is not well-formed. (12)

5. Finding All Minimal Well-Formed Compositions

There is a simple algorithm for generating all minimal well-formed compositions of a given system description. This section contains a program for that algorithm, an explanation and a correctness proof. We show that the program generates minimal compositions, that it generates all distinct solutions, that it generates only one of all possible permutations of a solution, and that all solutions it generates are well-formed.

5.1. The Algorithm

At the heart of the algorithm is **procedure** 'expand'.

```
procedure expand(COMPOS,REM) =
        local need = p(S) ∪ r(COMPOS)
        L0: if need ⊆ p(COMPOS) then printsolution(COMPOS); return fi
        L1: for i in REM while need ⊆ p(COMPOS . REM) if need ∩ pi ≠ ∅ do
              L2: REM := REM - {i}
              L3: if pi ∩ p(COMPOS) = ∅ then L4: expand(COMPOS ∩ {i}, REM) fi
              od
end expand
```

A program for generating all distinct minimal well-formed compositions is

```
declare COMPOS := ∅, REM := {1, . . . ,m}
begin expand(COMPOS,REM) end
```

The program described here operates on stand-alone systems. A stand-alone system is characterized by $r(S) = \phi$, implying $\bigcup_{i \in M} r_i \subseteq \bigcup_{j \in M} p_j$ (see rule 1). This says that all facilities required by its components are collectively provided by its components. General purpose operating systems are examples of stand-alone systems.

The algorithm can easily be extended to apply to arbitrary systems. The only change is in the definition of constant 'need'. A composition does not need any facilities included in r(S), because those will be supplied by external modules and systems. The definition of 'need' must be modified to read

local need = (r(COMPOS) ∩ p(S)) - r(S).

In order to avoid verbosity in the explanation and the correctness proof, we restrict our discussion from now on to stand-alone systems.

5.2. Explanation

The basic purpose of the procedure is to expand COMPOS by transferring an element from REM to COMPOS. Selection of an element in REM depends on the facilities that are needed by COMPOS. Initially, COMPOS is empty and REM contains all indices (representing all components). The facilities initially needed by COMPOS are those in p(S) because these must eventually be provided by COMPOS. When COMPOS is extended, additional facilities may be needed depending on the required facilities of the added element. For this reason the total need of COMPOS is set to p(S) $\cup$ r(COMPOS), where r(COMPOS) is short for $\{ \bigcup r_i \mid i \in$ COMPOS $\}$.

A composition provides all the necessary facilities if need $\subseteq$ p(COMPOS). In the next subsection we will show that the solutions printed in statement L_0 are MWF compositions. If 'need' is not included in p(COMPOS), all elements of REM will be tried in succession (see statement L_1). However, the construction is short circuited when the needed facilities are not included in p(COMPOS $\cup$ REM), because in that case transferring elements from REM to COMPOS will never lead to a COMPOS that provides all needed facilities.

If an element is encountered in REM that provides none of the needed facilities (see if clause in statement L_1), this element is skipped and left in REM because it may be used later if additional facilities are needed. An element that provides some of the needed facilities is used right away: it is removed from REM (see statement L_2). If it causes no conflict with the elements already in COMPOS (see if clause in statement L_3), the removed element is added to COMPOS and 'expand' is called recursively (see statement L_4). It is clear that the program terminates because the **for** statement calls 'expand' a finite number of times and the number of elements in REM decreases with each recursive call. The procedure is not called when REM = ϕ.

5.3. The Correctness Proof

Observe that every permutation of a MWF composition is itself a MWF composition. Such compositions are not distinct because they provide the required facilities by the same subset of components. We will show that the algorithm generates only distinct and interesting solutions without backtracking or duplication. To be precise, we will show that

1. every generated solution is well-formed,
2. no solution is generated more than once,
3. every generated solution is minimal, and
4. all distinct solutions are generated.

Theorem 6: Every solution printed by the algorithm is well-formed.

Proof

A. Every printed solution is *proper*, because the if clause in statement L_0 is exactly the condition for being *proper*.

B. We prove by induction on the size of COMPOS that COMPOS is always ***conflict-free.***

B1. COMPOS is conflict-free for size(COMPOS) = 0, because $COMPOS_0 = \emptyset$.

B2. Suppose we proved that COMPOS is conflict-free for $0 \le$ size(COMPOS) $< k$. All COMPOS of size k are generated in statement L_4.

$$\text{COMPOS conflict-free: } (i,j \in \text{COMPOS})\ i \neq j \Rightarrow p_i \cap p_j = \phi \qquad (13)$$

The if clause of statement L_3 assures that for the chosen index i

$$(j \in \text{COMPOS})\ p_j \cap p_i = \phi \qquad (14)$$

Combination of (13) and (14) results in

$$(j,l \in \text{COMPOS} \cup \{i\})\ j \neq l \Rightarrow p_j \cap p_l = \phi$$

Thus, if $COMPOS_{k-1}$ is conflict-free, then $COMPOS_k$ is also conflict-free. The induction

principle establishes the correctness of B.

*****

Theorem 7: The algorithm generates no value of COMPOS more than once.

Proof

Suppose COMPOS = $\{i_1, \ldots, i_k\}$ is generated more than once.

Case 1: COMPOS' = $\{i_1, \ldots, i_{k-1}\}$ is generated once and COMPOS is generated twice from COMPOS'. This is not possible, because as soon as COMPOS is generated from COMPOS', element i_k is removed from REM and is therefore never chosen again in an extension of COMPOS' (see statement L_2).

Case 2: COMPOS' is generated more than once. This cannot be the case, since by applying the reasoning recursively we find eventually that $COMPOS_0 = \emptyset$ must have been generated more than once. This is not the case, because $COMPOS_0 = \emptyset$ is generated only once in the main program. For all recursive calls of 'expand', size(COMPOS) > 0 (see statement L_4).

*****

Theorem 8: The algorithm generates only one of the collection of all permutations of a given COMPOS.

Proof

Let COMPOS' and COMPOS" be permutations of one another and let $COMPOS_k = \{i_1, \ldots, i_k\}$ be the common left part of these permutations ($COMPOS_k$ may be empty). COMPOS' and COMPOS" both are extensions of $COMPOS_k$: COMPOS' = $\{i_1, \ldots, i_k, x \ldots y \ldots\}$ and COMPOS" = $\{i_1, \ldots, i_k, y \ldots x \ldots\}$. x and y are the leftmost elements that differ in COMPOS' and COMPOS". Each one occurs in the remainder of the other, because COMPOS" = perm(COMPOS').

Both x and y are in REM_k and are apparently eligible as extensions of $COMPOS_k$. Suppose x occurs before y in REM_k. Statement L_1 shows that x is chosen as extension before y. However, once x is chosen, it is removed from REM_k. This means that $x \notin REM_k$ when y is chosen as extension of $COMPOS_k$. Thus, if COMPOS' exists then COMPOS" cannot exist.

*****

Theorem 9: All compositions generated by the algorithm are minimal.

Proof

Let COMPOS' = $\{i_1, \ldots, i_k\}$ be a solution generated by the algorithm. Suppose COMPOS' is not minimal. Then there exists a COMPOS" $\subset$ COMPOS' which is a proper composition. Let i_x be the leftmost element that occurs in COMPOS' but not in COMPOS" and let COMPOS := $\{i_1, \ldots, i_{x-1}\}$ be the common root of COMPOS' and COMPOS".

We take it for granted that the algorithm generates COMPOS at some point before it generates COMPOS', which is an extension of COMPOS. It is not possible that COMPOS is a solution, because this would imply that no further extensions of COMPOS are generated (see statement L_0), including COMPOS'.

Index i_x is selected as extension of COMPOS in statement L_1. This implies that $p(i_x) \cap$ need(COMPOS) $\neq \phi$. Let f be an element of that intersection. We proved in theorem 6 that COMPOS' is conflict-free. Thus

$$f \notin p_j \text{ for all } j \in (\text{COMPOS'} - \{i_x\}) \tag{15}$$

Since $i_x \notin$ COMPOS",

$$\vee \text{ COMPOS"} \subseteq \text{COMPOS'} - \{i_x\} \tag{16}$$

Combining (15) and (16) yields

$$(j \in \text{COMPOS"})\ f \notin p_j \Rightarrow f \notin p(\text{COMPOS"})$$

However, f $\in$ need(COMPOS) implies f $\in$ need(COMPOS"). Thus, need(COMPOS") is not included in p(COMPOS"). This implies that COMPOS" is not a proper composition. It follows that COMPOS' is minimal, because none of its subsets is a proper composition.

*****

Theorem 10: The algorithm generates all minimal well-formed compositions modulo permutations.

Proof

Let $W = \{i_1, \ldots, i_k\}$ be a minimal well-formed composition. Assume that W is sorted in ascending order. We will show that the algorithm generates either W itself or a permutation of W.

W is proper: $p(S) \cup r(W) \subseteq p(W)$ (17)
conflict-free: $(i,j \in W)\ i \neq j \Rightarrow p_i \cap p_j = \phi$ (18)
minimal: $(X \subset W)\ p(S) \cup r(X) - p(X) \neq \phi$ (19)

Suppose we show that the algorithm calls 'expand' with a pair (COMPOS,REM) satisfying

$COMPOS \cap REM = \phi$ (20)
$COMPOS \subseteq W$ (21)
$V = W - COMPOS \subseteq REM$ (22)

We want to show that the algorithm either prints COMPOS as a solution if $V = \phi$, or selects an element $j \in REM$ that is also in W and calls 'expand' with a pair (COMPOS',REM') satisfying (20), (21) and (22), where $COMPOS' = COMPOS \cup \{j\}$ and $REM' \subseteq REM - \{j\}$.

The algorithm defines need := $p(S) \cup r(COMPOS)$.

Case 1: $V = \phi$. Use (22): $W \subseteq COMPOS$. Combined with (21), we find W = COMPOS. Use (17):

$$p(S) \cup r(COMPOS) = p(S) \cup r(W) \subseteq p(W) = p(COMPOS).$$

Thus, need $\subseteq$ p(COMPOS). In this case the algorithm prints COMPOS as a solution (see the <u>if</u> clause of statement L_0).

Case 2: $V \neq \phi$. Use (22): $COMPOS \subset W$ (COMPOS is a true subset of W).

B1: Substitute COMPOS for X in (19). $p(S) \cup r(COMPOS) - p(COMPOS) \neq \phi$ implies 'need' is not included in p(COMPOS). Thus, COMPOS is not printed as solution in statement L_0.

B2: We show that there is an index $j \in V$ such that $p_j \cap need \neq \phi$. Substitute COMPOS for X in (19). It follows that

$$(\exists\, f \in p(S) \cup r(COMPOS))\ f \notin p(COMPOS) \quad (23)$$

Use (21): $f \in p(S) \cup r(W)$
Use (17): $f \in p(W)$
Use (21), (22): $f \in p(V \cup COMPOS) = p(V) \cup p(COMPOS)$
Use (23): $f \in p(V)$
thus, $(\exists\ j \in V)\ p_j \cap need \neq 0$. (24)

B3: Let j be the smallest index in V satisfying (24). Use (22): $j \in REM$. Because of (24), j is an eligible index in loop L_1. We show that the iteration is not aborted before it reaches j. It seems as if it may be aborted by the test need $\subseteq$ p(COMPOS $\cup$ REM).

Let REM* be derived from REM by eliminating some or all indices $i \in REM$ for which $p_i \cap need \neq \phi$ and $i < j$. All indices selected in loop L_1 before j (if any) satisfy these conditions. None of these are in V, because j is the smallest index in V for which $p_j \cap need \neq \phi$. Thus, using (22), we find

$V \subseteq REM*$ (25)
Use (21): $need = p(S) \cup r(COMPOS) \subseteq p(S) \cup r(W)$
Use (17): $p(S) \cup r(W) \subseteq p(W)$
Use (21, 22): $p(W) = p(COMPOS \cup V)$
Use (25): $p(V) \subseteq p(REM*)$
thus, $need \subseteq p(COMPOS) \cup p(REM*)$ (26)

This implies that the **while** clause is true for all iterations before j is reached.

B4: Let COMPOS' = COMPOS $\cup$ {j}, let REM* be derived from REM by eliminating all indices $i < j$ for which $p_i \cap need \neq \emptyset$ and let REM' = REM* - {j}. We show that the pair (COMPOS',REM') satisfies (20), (21) and (22).

Use (20): $COMPOS' \cap REM' = (COMPOS \cup \{j\}) \cap (REM* - \{j\})$
$= COMPOS \cap REM* \subseteq COMPOS \cap REM = \phi$
Use (21): $COMPOS' \subseteq W$, because $COMPOS \subseteq W$ and $j \in V \subseteq W$.
Use (25): $V' = W - COMPOS' = (W - COMPOS) - \{j\} = V - \{j\} \subseteq REM* - \{j\} = REM'$

B5: Now we show that 'expand' is called recursively with pair (COMPOS',REM').

Suppose $p_j \cap p(COMPOS) \neq \phi$. This cannot be true if COMPOS = ϕ. Let $j \in REM'$; $j \notin$ COMPOS (see (20)). If COMPOS is non- empty, there is an index $i \in COMPOS$ and $i \neq j$.

Use (21, 22, 18): $i,j \in W\ i \neq \Rightarrow p_i \cap p_j = \phi$

This is true for all $i \in$ COMPOS, so $p_j \cap p(\text{COMPOS}) = \phi$. This means that the condition of statement L_3 evaluates to true, resulting in the call "expand(COMPOS',REM')".

B6: (20, 21, 22) are initially true, because $\text{COMPOS}_0 = \phi$ and $\text{REM}_0 = \{1, \ldots ,m\}$. Observe that size(V') = size(V) - 1 (see B5). Thus, an inductive proof can be constructed starting with pair (COMPOS_0,REM_0) and using B1 through B5 as proof of the induction step. Induction is on the size of V.

*****

6. The Complexity of the Composition Generation Algorithm

The worst case complexity of our composition generation algorithm is O(n!) where n is the number of components in the system. The worst case space complexity is O(n) if the algorithm is executed on a single processor.

The canonical worst case* is the system specification that provides facilities $f_1 \ldots f_n$ where each component c_i provides facility f_i and requires facility f_{i+1} except for component c_n which provides all the facilities but requires none.

```
sys   S   prov   f1, f2, f3, f4;
      mod A       prov    f1;   req f2; . . . end
      mod B       prov    f2;   req f3; . . . end
      mod C       prov    f3;   req f4; . . . end
      mod D       prov    f1,f2,f3,f4;  . . . end
         . . .
end
```

While this pathological case may exist, it seems that systems such as this one but with a large number of components would exist only as contrived examples.

There are several factors that ameliorate what appears to be an unacceptable level of complexity.

1. Subsystems have (or should have) a small number of components (say two to five).

*We are indebted to Loretta Guarino Reid for this example

2. Within subsystems, components rarely conflict.

3. Systems have (or should have) a small number of compositions.

The subsystems in the appendix exhibit these characteristics: all have two to five components and have only one composition each. As a result, the composition generation process is partitioned into small segments each of which requires only a small amount of work.

A fourth factor is that even when the number of components is relatively large, the components that conflict will reduce the number of paths actually traversed in the execution of the algorithm. The number of actual executions of Expand will generally be far smaller than the number of executions possible.

Consider the following example (abstracted from the last system specification in the appendix). There are 13 components, some of which conflict in different ways: four components provide facility f1; two components provide facility f2; three components provide facility f5.

```
sys   S    prov   s1, s2;
      use  A1     prov   f1;                req f2;
      use  A2     prov   f1;                req f2;
      use  A3     prov   f1;                reg f2;
      use  A4     prov   f1;                req f2;
      use  B1     prov   f2, f3;            req f1, f9;
      use  B2     prov   f2, f4;            req f1, f9;
      use  C1     prov   f5;                req f1, f3;
      use  C2     prov   f5;                req f1, f3;
      mod  C3     prov   f5;                req f1, f4, f13; . . . end
      use  D      prov   f6, f7, f8, f9;    req f1;
      use  E      prov   f10, f11, f12;     req f1,f 8;
      use  F      prov   f13;
      use  G      prov   s1, s2;            req f1, f5, f7, f10, f11, f12;
      . . .
end
```

There are 12 minimal well-formed compositions generated by the algorithm for this example:

G, A1, B1, C1, D, E

G, A1, B1, C2, D, E
G, A1, B2, C3, D, E, F
G, A2, B1, C1, D, E
G, A2, B1, C2, D, E
G, A2, B2, C3, D, E, F
G, A3, B1, C1, D, E
G, A3, B1, C2, D, E
G, A3, B2, C3, D, E, F
G, A4, B1, C1, D, E
G, A4, B1, C2, D, E
G, A4, B2, C3, D, E, F

In addition, there are 12 partial compositions which cannot be completed because required facilities cannot be provided by the remaining components:

G, A1, B1, C3
G, A1, B2, C1
G, A1, B2, C2
G, A2, B1, C3
G, A2, B2, C1
G, A2, B2, C2
G, A3, B1, C3
G, A3, B2, C1
G, A3, B2, C2
G, A4, B1, C3
G, A4, B2, C1
G, A4, B2, C2

In summarizing the computation for this example, it is first worth noting that only a maximum of 7 components of the 13 are needed to form complete compositions - a reduction for the worst case of work from 13! (6,227,020,800) invocations of Expand to 7! (5040) invocations of Expand. Secondly, there are in fact considerably fewer calls to expand than this - 78 in all: 12 of which determine that a minimum well-formed composition has been completed and another 12 of which determine that the remaining components do not contain provisions for the compositions' requirements. Notice also that the textual ordering of the components in this example (i.e., conflicting components are grouped together and the components are listed more or less in the order that provided facilities are needed) promotes an efficient execution of each activation of Expand: the desired components appear early in the list and the execution stops after each useful component has been added to a composition.

Thus, while the algorithm has a worst case complexity that is appalling, the appropriate use of the system description tools will produce a practical complexity that is well within reasonable bounds.

7. Conclusion

The system version description facility discussed in this paper is part of an integrated programming environment. In contrast to the traditional approach of isolated tools such as a debugger or a link-editor, a programming environment knows the properties of the type of objects that are manipulated within its boundaries. The environment is an active participant in the construction of system objects, primarily by enforcing some basic construction rules.

The package concept of the Ada language has proven to be very useful as a basis for system version descriptions. System and module descriptors are extended forms of visible parts of Ada packages. Since the Ada language demands that there be exactly one implementation body for every package descriptor, a small but significant extension was needed in order to include the essential feature of describing a variety of different implementations of a module or different compositions of a system.

Systems are described in terms of basic building blocks, *modules*, P and two composition constructs, *systems* and *boxes*. Modules describe the various concrete implementations of elementary system facilities. System descriptors are used to nest components and to name various ways in which a system version can be put together. The distinction between systems and boxes is basically that of providing a collection of facilties versus a selection of related systems.

There are some basic rules that must be satisfied by system descriptors and that must be applied to compositions (prescriptions for generating system versions). The first set of rules guarantees that the components of a system indeed provide the facilities that the system claims to provide. The programming environment can easily enforce these rules and automatically derive which facilities a particular system needs from external sources. The

second set of rules not only guarantees that a composition provides what the system promises, but also insures that compositions are complete and free of internal conflicts. These properties are expressed by the notion of well-formed compositions.

We showed that while every system description has compositions that provide all system facilities, not every system description has well-formed compositions. It may be that none of the possible compositions is conflict free. We discussed two ways of eliminating such undersirable descriptors. First, we showed that every system that does not have a well-formed composition can be transformed into one that has a well-formed composition. Second, we showed that by enforcing the second set of construction rules the programming environment can prevent the user from writing down such undesirable system descriptors. We strongly favor the second approach over the first, because in the first approach the resulting transformed system descriptors may not reflect the desired partitioning of a system.

The algorithm of section 5 can be used to prove constructively that a given system description a has well-formed composition, or it can be used to generate all possible well-formed compositions. By making the algorithm part of the standard repertoire of the programming environment, the latter can warn a user when he or she introduces a non-well-formed composition.

The complexity of the algorithm is theoretically high. It seems, however, that the algorithm will perform adequately in practice. The example presented in the appendix demonstrates some of the common habits and conventions that programmers are likely to follow when they partition systems into modules.

The essence of the paper has been to show that there are some simple construction rules that can be applied by a programming environment for the purpose of assuring that all described systems are well-formed.

Acknowledgements

We wish to gratefully acknowledge the contributions of Loretta Guarino Reid for her

discussions of the complexity of the composition generation algorithm, David Notkin for his careful readings and constructive criticisms, and Izumi Kimura, Philip Wadler, Gail Kaiser and Raul Medina-Mora for their critical comments.

References

[Ada 79] Ichbiah, J. D., et al.
Preliminary Ada Reference Manual.
SIGPLAN Notices 14(6 Part A), June, 1979.

[Cooprider 79] Cooprider, L. W.
The Representation of Families of Software Systems.
PhD thesis, Carnegie-Mellon University, April, 1979.

[Gandalf 79] Habermann, A. N.
The Gandalf Research Project.
Computer Science Research Review. Carnegie-Mellon University. 1978-79. , 1979.

[Gaudette 75] Gaudette, J. M.
Word Processing at Dun & Bradstreet.
Datamation , November, 1975.

[Tichy 80] Tichy, W. F.
Software Development Control Based on System Structure Description.
PhD thesis, Carnegie-Mellon University, January, 1980.

I. An Example

The system described here is drawn from a "real" existing system -- the Dun & Bradstreet AOS Minicomputer System. We have simplified the system by emphasizing only the basic aspects of the system.

From a pool of programs and subsystems, two basic systems are constructed -- the Data Entry System (DES) and the Inquiry System (INS). For the latter system, two primary versions exist -- the Duns Dial System (DDS) and the Remote Terminal System (RTS).

We proceed first with a short description of each component in order to feel somewhat comfortable with the construction of the system.

The Executive (EXEC) is the operating system and as such provides services (SVCs) for IO, memory management, job management, file management, etc. It requires the program file

(CPS file), a task description table (TDT), and a system initialization task.

The D&B minicomputer systems are concerned primarily with credit reports. These reports are entered, updated, transmitted and stored in a compressed coded form. For an early description of the system, see [Gaudette 75].

Warm Start (WS) provides the system initialization tasks such as opening files, moving data from files to memory, and invoking the initiators of the various processes in the system. To this end it requires task initialization procedures from various components (which ones are required is dependent upon the particular system).

Video Control (VC) provides the basic data entry and screen handling capabilities for the CRT/keyboard terminals and is the interface (through the EXEC) between the user and the dialog control programs (Report Management and Report Inquiry). Data is transmitted between the two in conjunction with screen formats described in the Display File (DF).

The Remote Terminal Handler (RTH) is similar to Video Control, but is used in the Remote Terminal system to interface with various communicating terminals rather than the directly connected CRT/keyboard terminals.

Report Management (RM) controls the interaction of the user with the system. It guides the user in the process of entering (RE), updating (RU), generating (RG), requesting (RR), and sending (RS) reports to the central data bank. Reports entered and updated are validated for correctness of form and, to some extend, content (Data Validation provides this). They are stored in the system by Report File Access (RFA). Transmitted reports are sent and received through the interface provided by the Message Queue Handler. Reports are generated by Report Generation. RM is used in the Data Entry Systems.

Report Inquiry (RI) provides a similar function for the Inquiry systems. The facilities provided to the user, however, are different: Name and Address Look Up (NALU) and Report Display (RD). Required interfaces are Report Generation to generate the reports for Report Display and Message Queue Handler to retrieve name and addresses and reports from the

data bank.

Data Validation (DV) validates that the report has the correct syntax and, where possible, consistent contents. It is essentially a table driven program working from the Data Validation File (DVF) for the appropriate report type. It retrieves the report data from Report File Access.

Report File Access (RFA) provides the capabilities to enter and update reports and to retrieve them from local storage. It uses the basic Report File I/O routines to do this.

The Message Queue Handler (MQH) provides the capabilities to send and retrieve reports from remote storage in the central data bank. It communicates with Communications through the Message Queue (MQ) to send and receive reports.

Communications (COMM) is an independent task that encapsulates the handling of binary synchronous communications. The Message Queue specifies what is to be output. The Message Handler (MH) is invoked to deblock input and to coordinate the storage of input information through the Message Queue. The deblocked data is stored in the Report File through the Report File I/O routines.

Report Generation (RPG) retrieves the specified report from either RFA or MQH, depending on the configuration and the request, and expands it according to the generation blocks kept in the Report Generation File (RGF). Print lines are queued to the Print Line Queue (PLQ) (from which they are removed to be either printed by the Printer Driver (PD) or displayed by Report Inquiry.

Figures 1 and 2 show the basic provide and require dependencies between the components of the two systems. For simplicity, the dependencies of all components upon the Executive are removed as are the initialization requirements of Warm Start. The arrows point in the direction of the components requiring provided facilities.

Rather than just present the entire system description in one piece, we will first give an overview of the entire structure and then proceed a component at a time, elaborating the

Figure 1.

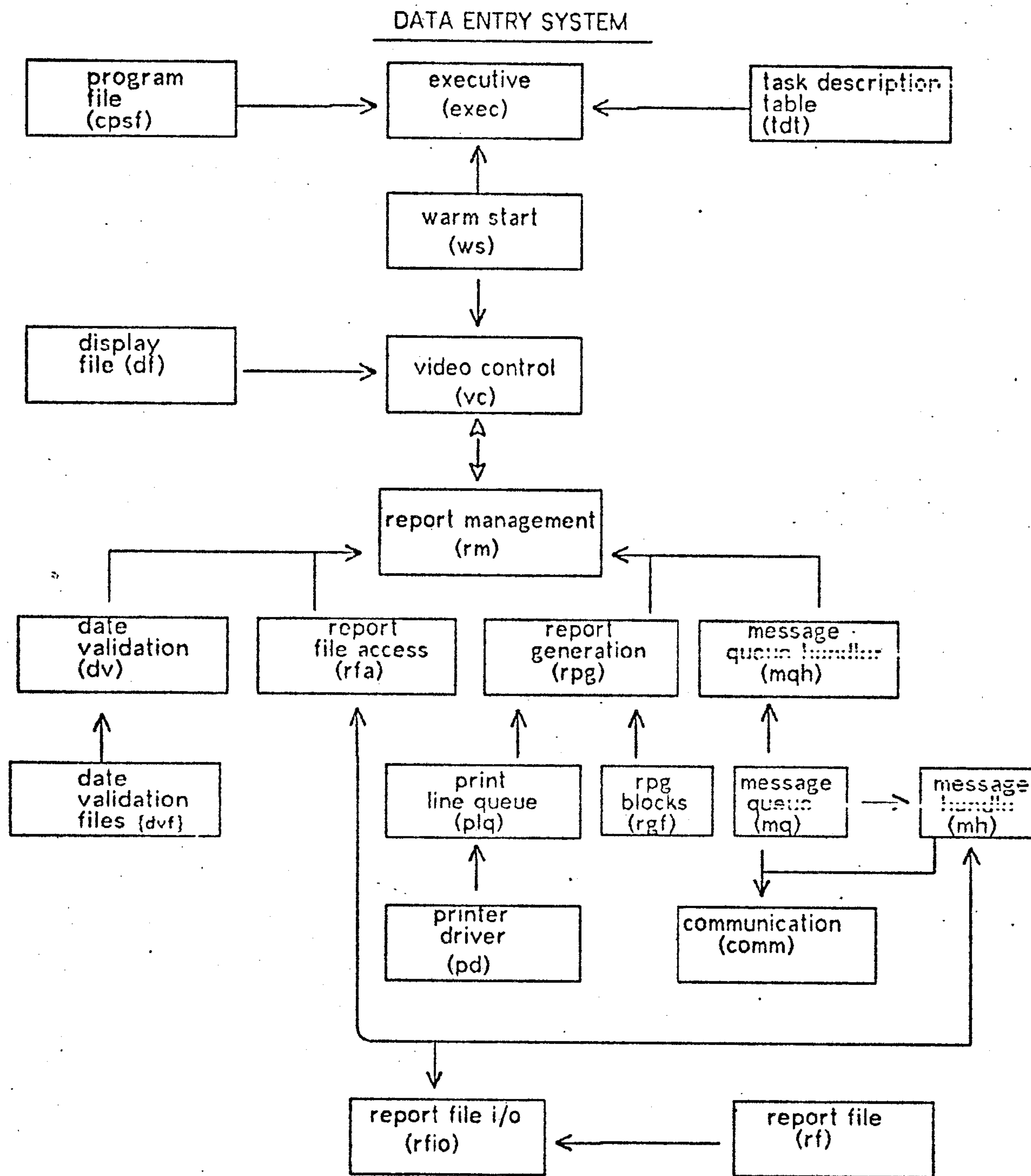
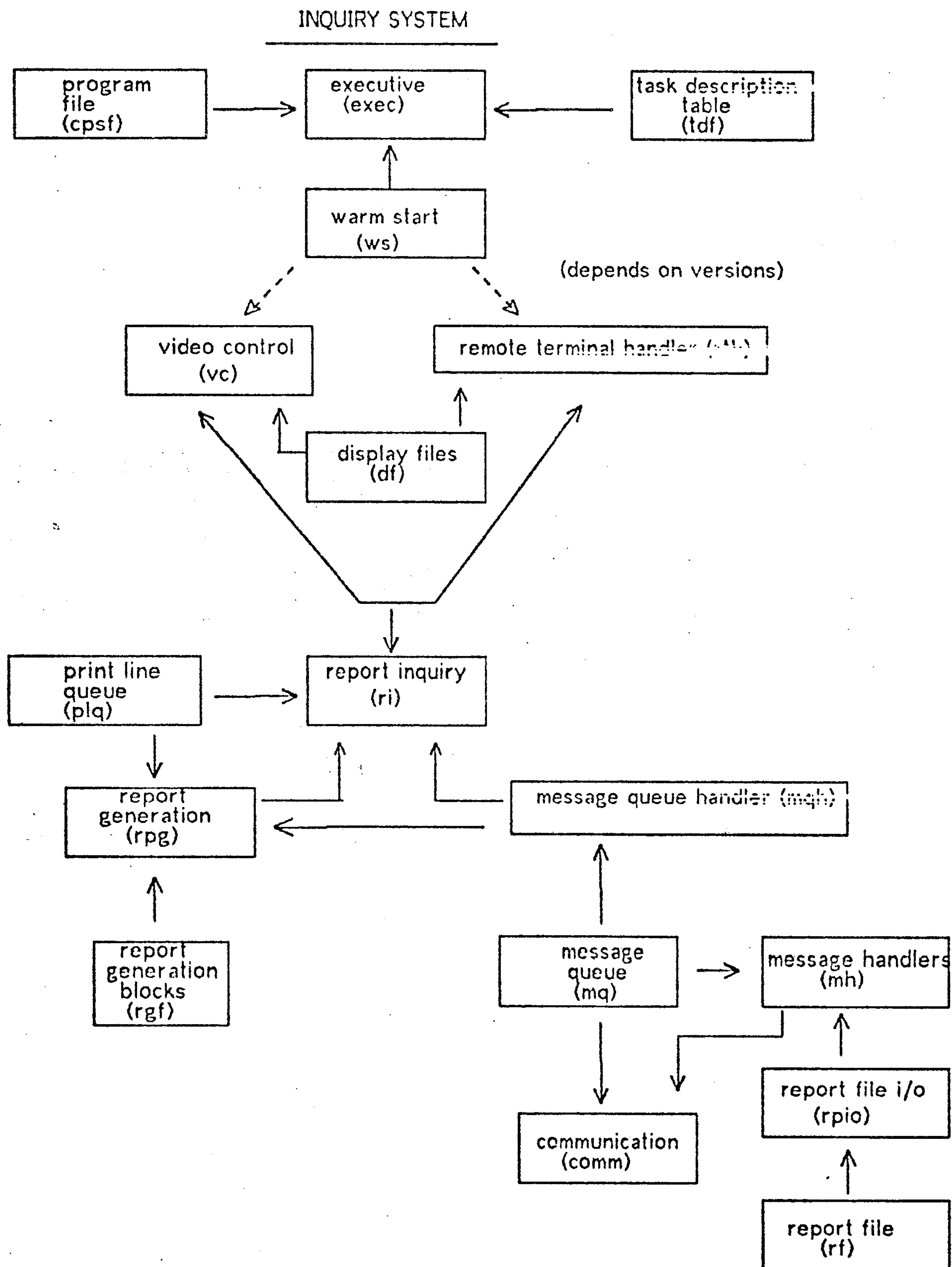


Figure 2.



structure in more detail. We will comment on the description as we proceed.

The entire minicomputer system is encapsulated in a box from which we export two systems, DES and RIS.

```
box DB Systems exports DES, RIS;
        box Executives              . . .  end
        box Warm Start              . . .  end
        box Display Files           . . .  end
        box Video Controls          . . .  end
        box Interface               . . .  end
        box Report File             . . .  end
        box Report Generation       . . .  end
        sys DES                     . . .  end
        sys RIS                     . . .  end
end
```

The dividing line between boxing to export different versions of a component and composing to make different versions is not clear. Clearly, when distinct facilities are provided boxes should be used. The case here is not clear: the facilities provided by the executives remain identical when considered from the standpoint of usage by the system proper. However, there are other facilities that are provided and used that are not apparent from the current system description. Hence our use of boxes.

```
box executives
    export Exec.S,      -- standard
           Exec.DM,     -- disc measures
           Exec.MM,     -- mapped memory
           Exec.RM;     -- resource measures
    mod Execfiles prov CPSF, TDT; . . . end
    sys Exec.S
        prov SVCs
        req systeminitialization( . . . );
        use Execfiles . . . ;
        sys EXEC . . . end;
        comp . . . ;
    end;
    sys Exec.DM . . . end;
    sys Exec.MM . . . end;
    sys Exec.RM . . . end;
end;
```

Because of the possible use of either Video Control or Remote Terminal Handler, we require two versions of Warm Start. This justifies the use of boxes--there are different facilities required by the two versions.

```
box WarmStarts exports WSVC, WSRTH;
        mod WSVC
                prov systeminitialization( . . . ), VCinit( . . . );
                req SVCs, COMMinit( . . . );
                impl 4June79 =  . . . ;
        end
        mod WSRTH
                prov systeminitialization( . . . ), VCinit( . . . );
                req SVCs, COMMinit( . . . );
                impl 17August79 =  . . . ;
        end
end
```

The Video Control component has two distinct versions sufficiently different as to be given separate names and boxed together. The standard video requires the standard video display file. The Buffered CRT Video Control only works in conjuncation with a special display file and special hardware (not specified in the system description). Again, that makes the versions sufficiently distinct to warrant boxing them.

```
box Video Controls exports VCS, VCBCRT;
        sys VCS
                prov Display( . . . );
                req  SVCs, VCinit( . . . );
                use  DFS prov DF;
                mod VC
                        prov Display( . . . );
                        req  SVCs, DF, VCinit( . . . );
                        impl June78 =  . . . ;
                        impl May79  =  . . . ;
                end
                comp new = DFS, VC;
                comp old = DFS, VC.June78;
        end
        sys VCBCRT
                prov Display( . . . );
                req SVCs, VCinit( . . . );
                mod VC
                        prov VCinit( . . . ), Display( . . . );
                        req SVCs, DF;
```

```
                impl ... ;
            end
            use DFBCRT prov DF;
            comp Sept79 = DFBCRT, VC;
        end
end
```

Note that we have exported a system rather than a module and that we have tied the appropriate file to the desired version of Video.

We package the display file as the facility of a module, since that is the lowest level of granularity of system components. As mentioned above there are two different constructions given to the display file which are incompatible.

```
box Displayfiles export DFS, DFBCRT;
        mod DFS prov DF;          -- standard
            impl ... ;
        end;
        mod DFBCRT prov DF;       -- buffered CRT
            impl ... ;
        end
end
```

The report file provides two completely different mechanisms for filing and retrieving reports. The one uses the Report File Access program and the other uses the Message Queue Handler, Message Handler and Communications. The form of the reports in the file is different as well. The RF routines provide merely basic consistency checks as well as consistent access to the reserve and release page mechanisms.

```
box Reportfile exports RFAsys, MQsys;
        sys RF
            prov write( ... ), read( ... ), reserve( ... ), release( ... );
            req SVCs;
            mod RFIO
                prov write( ... ), read( ... ), reserve ( ... ), release ( ... ),
                req SVCs, Rfile
                impl ... ;
            end
            mod RFmod prov Rfile;
                impl ... ;
            end
            comp new = RFmod, RFIO;
```

```
    end
    sys RFAsys
        prov getreport( . . . ), putreport( . . . );
        req SVCs;
        use RF prov write( . . . ), . . . , release( . . . );
        mod RFA
            prov getreport( . . . ), putreport( . . . );
            req SVCs, write( . . . ), . . . , release( . . . );
            impl . . . ;
        end
        comp new = RF, RFA;
    end
    sys MQsys
        prov sendreport( . . . ), receivereport( . . . ),
             obtainreport( . . . ), COMMinit( . . . );
        req SVCs;
        use RF prov write( . . . ), . . . , release( . . . );
        mod MQmod prov MQ, enq( . . . ), deq( . . . ), find( . . . );
            impl . . . ;
        end;
        mod MQH
            prov sendreport( . . . ), receivereport( . . . ),
                 obtainreport( . . . );
            req SVCs, read( . . . ), release( . . . ), MQmod;
            impl . . . ;
        end
        mod MH
            prov putreport( . . . );
            req SVCs, read( . . . ), write( . . . ), reserve( . . . ),
                MQ, find( . . . );
            impl . . . ;
        end
        mod COMM
            prov BSC, COMMinit( . . . );
            req MQ, find( . . . ), putreport( . . . );
            impl . . . ;
        end
        comp new = MQH, RF, MQmod, COMM;
    end
end
```

The Interface box exports the two programs that control the user dialog for system facilities. Their facilities are radically different but their function is the same.

```
box Interface export RM, RI;
    mod RM
        prov RE, RU, RR, RS, RP
        req SVCs, validatereport( . . . ), getreport( . . . ), putreport( . . . ),
            sendreport( . . . ), receivereport( . . . ), generatereport( . . . ),
```

```
                display( . . . );
            impl . . .
        end
        mod RI
            prov NALU, RD
            req SVCs, receivereport( . . . ), generatereport( . . . ),
                display( . . . ), PLQ, PLQ.deq( . . . );
            impl . . .
        end
end
```

We require two distinct versions of report generation because it must interface with both report filing mechanisms.

```
box ReportGenerations export RPGDES, RPGRIS;
        mod RPGFmod prov RPGF;
            impl . . . ;
        end
        mod PLQmod prov PLQ, enq( . . . ), deq( . . . );
            impl . . . ;
        end
        sys RPGDES
            prov generatereport( . . . ), PLQ, PLQ.enq( . . . );
            req getreport( . . . ), obtainreport( . . . ), SVCs;
            use RPGFmod prov RPGF;
            use PLQmod prov PLQ, . . . ;
            mod RPG1
                prov generatereport( . . . );
                req getreport( . . . ), obtainreport( . . . ), SVCs, PLQ,
                    RPGF;
                impl . . . ;
            end
            comp new = RPG1, RPGFmod, PLQmod;
        end
        sys RPGRIS
            prov generatereport( . . . ), PLQ, PLQ.deq( . . . );
            req obtainreport( . . . ), SVCs;
            use RPGFmod prov RPGF;
            use PLQmod prov PLQ , . . . ;
            mod RPG2
                prov generatereport( . . . );
                req obtainreport( . . . ), SVCs, PLQ, PLQ.enq( . . . ), RPGF;
                impl . . . ;
            end
            comp new = RPG2, RPGFmod, PLQmod;
        end
end
```

The two main systems are then described in terms of the components in the environment and components local to the system.

```
sys DES
      prov RE,RU,RR,RT,RP;
      use Exec.S  . . . ;
      use WSVC    . . . ;
      use VCS     . . . ;
      use RFAsys  . . . ;
      use MQsys   . . . ;
      use RU      . . . ;
      use RPGDES  . . . ;
      mod PD
            prov PD;    -- drives printer
            req SVCs, PLQ, PLQ.deq( . . . );
            impl . . . ;
      end
      sys DV
            prov validate( . . . );
            req SVCs, getreport( . . . );
            mod DVFmod prov DVf;
                  impl . . . ;
            end
            mod DV
                  mov validate
                  req SVC, DVF, getreport( . . . );
                  impl . . . ;
            end
            comp new = DV, DVFmod;
      end
      comp new = Exec.S, WSVC, VCS, RFAsys, MQsys, RM,
                 RPGDES, DV, PD
end
```

The Report Inquiry System has a wider range of versions. Two main versions consist of the Duns Dial System and the Remote Terminal System. There exist a number of versions for the Duns Dial System to try out new hardware and do performance measurement and analysis.

```
sys RIS
      prov NALU, RD;
      use Exec.S  . . . ;
      use Exec.DM . . . ;
      use Exec.MM . . . ;
      use Exec.RM . . . ;
```

```
        use WSVC ...;
        use WSRTH ...;
        use VCS ...;
        use VCBCRT ...;
        use MQsys ...;
        use DFS ...;
        use RPGRIS ...;
        use RI;
        mod RTH
               prov display( ... );
               req SVCs, DF, RTHinit( ... );
               impl ... ;
        end
        comp RTS = Exec.S, WSRTH, RTH, MQsys, RI, RPGRIS, DFS;
        comp DDS = Exec.S, WSVC, VCS, MQsys, RI, RPGRIS;
        comp DDS.DM = Exec.DM, WSVC, VCS, MQsys, RI, RPGRIS;
        comp DDS.RM = Exec.RM, WSVC, VCS, MQsys, RT, RPGRIS;
        comp DDS.MM = Exec.MM, WSVC, VCS, MQsys, RI, RPGRIs;
        comp DDS.BCRT = Exec.S, WSVC, VCBCRT, MQsys, RI, RPGRIS;
        comp DDS.BCRT.RM = Exec.RM, WSVC, VCBCRT, MQsys, RI, RPGRIS;
        comp DDS.BCRT.MM = Exec.MM, WSVC, VCBCRT, MQsys, RI, RPGRIS;
end
```

Thus we have completed our systems' descriptions. We have relied heavily on box and use statements to construct our systems. We constructed small systems where practical and useful. However, we still remained within the bounds of useful abstractions in structuring these small systems.

The amount of sharing of components is obvious. Where this is prevalent, the use statement is absolutely necessary.

We have illustrated a wide usage of the boxes and delineated a need for them as opposed to systems. The structuring would not be as clear without them.

In some systems, little emphasis is placed on versions. In system RIS, however, we see an elaborate use of versions to provide distinct systems. The one type of version not depicted here is the earlier released version: versions that depend upon particular versions of the components, usually dated versions. These versions play at least as large a part as the ones we have described.