Reasoning Tools

to Support

Systems Analysis and Design

by

James Daniel Paulson

B.Sc.(Eng.), The University of Saskatchewan, 1977
M.Sc.(Physics), The University of Saskatchewan, 1981
B.Ed., The University of Saskatchewan, 1982

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

(Faculty of Commerce)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
February, 1989

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Commerce and Business Administration

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date 06 Feb 89

Abstract

Some parts of the systems analysis and design process are not well structured and rely heavily on human judgement and experience. This is particularly true for decomposition and the validation of system specifications. Decomposition has long been considered a fundamental part of systems analysis and design. However, ensuring that a decomposition is optimal is nearly impossible.

Ensuring that a system specification is complete and consistent is an inherently difficult task. Most existing systems analysis and design methodologies allow only the use of techniques such as code walk-throughs and post-implementation testing. Analysis errors discovered at such late stages can be quite expensive to correct. Existing methodologies cannot support automated completeness and consistency testing because they lack the degree of formalism required to allow automation.

The primary objective of this research was to increase understanding of system decomposition. To aid in achieving this objective a formalism for representing a system specification, and a set of computer-based specifications analysis tools were developed. The tools support decomposition and provide completeness and consistency testing of a system specification.

An existing system modelling formalism was extended to provide the basis for the specification formalism. This extended formalism will allow an analyst to describe a system with the degree of precision necessary for automated testing and decomposition. The ability to create a complete and consistent system model facilitated the development of a general theory of system decomposition. A system model created using the specifications analysis tools can be analyzed using a decomposition algorithm based on this theory. The algorithm incorporates a number of commonsense software design rules and decomposition heuristics drawn from the literature, and has been included in the specifications analysis tools. Experience has shown that the specifications analysis tools may suggest system decompositions not previously considered by the analyst. Alternative decompositions may arise in two situations:

1. The system has a valid alternative structure which may not have been considered by the analyst. This alternative structure may be superior to the original structure envisioned by the analyst when the system model was constructed.

2. The system specification does not contain enough information to rule out certain unreasonable decompositions. The missing information should be explicitly included in the specification to avoid problems of interpretation later in the system development life cycle.

Analysis of several test systems (including the IFIP Working Conference system often used as a standard problem in the systems analysis literature) using the specifications analysis tools has proven the feasibility of automated consistency and completeness testing and decomposition. Further research is required in two areas:

1. Enhancement of the specifications analysis tools. The tools are not user friendly. An analyst will require extensive training to use them effectively. As well, the computational speed of the tools must be improved. Automated decomposition is too slow to allow easy interaction between the analyst and the tools.

2. A hierarchical analysis technique must be developed to support application of the specification formalism and the theory of decomposition to larger systems.

Table of Contents

Acknowledgements

I would like to thank all the members of my dissertation committee for their advice and support during the preparation of this thesis. In particular, I would like to express sincere gratitude to my research supervisor, Dr. Yair Wand, for his patience and countless hours of valuable discussion, and Dr. Richard Mattessich for his many helpful criticisms and suggestions with respect to the ontological foundations of this research.

I would also like to thank my wife, Kim, for her endless encouragement and tolerance during a tremendously stressful period in both of our lives.

Chapter 1: Introduction

1.1. General

The notion of decomposition is central to most methodologies for systems analysis and design. For example, Structured Analysis (DeMarco, 1978), Warnier-Orr Diagrams (Warnier, 1974; Orr, 1977), JSD[1] (Jackson, 1983), and HOS[2] (Hamilton and Zeldin, 1976) all require the analyst to construct a hierarchical structure for a proposed computer-based system. Courtois (1985) notes the importance of decomposition:

> "Decomposition has long been recognized as a powerful tool for the analysis of large and complex systems. The technique of decomposing a system, studying the components, and then studying the interactions of those components has been successfully used in many areas of engineering and science."

Despite this, there exists no theory to guide the process of system decomposition. Decomposition has always been considered a heuristic activity.

An important objective of the research is to increase our understanding of system decomposition. In order to achieve this objective a formalism has been developed for representing an information system based on the states and laws system model developed by Wand and Weber (1988, 1989). Not only does this formalism provide a basis for the development of a theory of system decomposition, but it includes operational definitions for completeness and consistency of system specifications. Tangible results of the research include a set of Prolog-based specification analysis tools. These tools can support formal description of a system model and suggest possible decompositions for that system. The decompositions will reveal the inherent structure of a system specification and can be used to identify deficiencies in the model[3].

---

[1] Jackson System Development

[2] Higher Order Software

[3] In this research, a system model is considered to be part of a full system specification. System models give the parts of the real world to be represented in the implemented information system and their relationships.

It is generally assumed that the structure and behaviour of an implemented system should closely mirror that of the specification. In the Structured Design literature, Myers (1978) states

> "...the program structure should closely model the problem structure." (p. 73)

JSD (Jackson, 1983, p. 5) defines system modelling to consist of two activities:

a. first, making an abstract description of the real world, and
b. second, making a realization, in the computer, of that abstract description.

Therefore, it seems reasonable to assume that the structure of the specification can provide a basis for the further design of a system implementation. A suggested decomposition will be "good" in the sense that it will support modular construction and restrict the effects of system maintenance to easily identifiable segments of the overall system. Since the decompositions suggested by the tools describe the structure of the specification, they will aid in both the understanding of a complex real system and in system design.

The contributions of this research include the following:

a. Integration of existing decomposition heuristics.
b. Development of a theory-based modelling technique for system specification and decomposition.
c. Development and implementation of a set of computerized tools for constructing a complete and consistent model of a system.
d. A formal theory of system decomposition.
e. Development and implementation of an algorithm capable of decomposing a system without human participation.

This chapter presents the results of a literature search for material related to decomposition in the fields of general system theory and computer science. The second chapter presents a system modelling formalism suitable for

---

System specifications may also include requirements such as minimum response times, required throughput, cost, etc..

use as the basis for a general theory of system decomposition. This formalism is an extension of the works of Bunge (1977, 1979) and Wand and Weber (1988, 1989), and supports automated testing of completeness and consistency. The theory of decomposition is developed in Chapter 3 and later extended in Chapter 5. Chapter 4 includes a discussion of various measures of system complexity. One of these measures is shown to be useful for discriminating between competing decompositions of a given system, and is adopted for use in this research. To illustrate their usefulness, the system modelling formalism and techniques stemming from the theory of decomposition are applied to the IFIP Working Conference problem (Olle, 1982, pp. 8-9) in Chapter 6. Definitions for some terms of general importance to systems analysis and design are suggested in Chapter 7, prior to a presentation of the conclusions reached in this research.

The next section describes previous research on system decomposition. Ideas from both system theory and software design are considered. The decomposition rules of Myers (1978), Yourdon and Constantine (1979), Hamilton and Zeldin (1976), and Mili et al. (1986) are described in detail.

1.2. Background

1.2.1. System Theory

Outside the systems analysis and design literature, the concept of system decomposition is viewed from two different, but complementary, perspectives. Simon and Ando (1961) and Courtois (1985) suggest the use of decomposition as an aid in analyzing an existing system. Alexander (1967) and Simon (1981) argue that a suitable decomposition could provide the basis for design of a new system.

From a systems analysis perspective, Simon and Ando consider the aggregation of variables in dynamic systems where short-run dynamics are separable from long-run dynamics. Their necessary criteria for a system to be decomposable are as follows (from Courtois, 1985):

a. In a short-term period, as a result of stronger internal bonds, subsystems tend to reach an internal equilibrium "approximately" independently of one another.

b. In a long-term period, when a whole structure evolves toward a global equilibrium state under the influence of weak interactions among subsystems, the internal equilibriums reached at the end of the short-term period are approximately maintained in relative value.

Simon and Ando illustrate these rules by considering thermal equilibrium in an office building (p. 117). The building is divided into a number of rooms. The rooms are separated from each other by walls which are good, but not perfect, insulators. The rooms are further divided into offices by poorly insulating partitions. Suppose that initially there is a large variation in the temperatures of the offices. After a relatively short time, the temperatures of each office in a particular room will be approximately equal. After a much longer time, the temperatures of each room will approach some common value. The thermal behaviour of individual rooms is a relatively short-term phenomenon. The rooms may be treated as independent subsystems with respect to this behaviour. The problem of how different short-term and long-term periods need be was addressed by Courtois (1985). He presented intuitively derived mathematical criteria for the decomposition of queuing networks. In Chapter 3, it will be suggested that Simon and Ando's criteria approximate more general rules governing "good" decomposition.

From a system design perspective, Alexander suggests that a good decomposition will lead to a design which exhibits a "good fit" with its environment. His examples are architectural, but most of his arguments are applicable to system design in general. He also defines a mathematical method for clustering system variables such that information transfer is minimized between clusters of modules.

Simon argues that a large proportion of naturally occurring systems in the world exhibit hierarchical structures, and that "On theoretical grounds we could expect complex systems to be hierarchies in a world in which complexity had to evolve from simplicity" (Simon, 1981, p. 229). Thus, a hierarchical structure is seen not only as a useful tool, but as a fundamental feature of the universe. Simon uses a parable of two watchmakers, named Hora and Tempus, to illustrate this point. Both men constructed watches consisting of 1,000 parts. Tempus constructed his watches in such a way that if he was interrupted and had to put it down, it immediately fell to pieces and assembly had to begin again. Hora's watches performed precisely the same functions as Tempus', but he

designed his to have stable subassemblies of 10 parts each. Ten of these subassemblies could be put together in another stable assembly, and ten of these final assemblies could be put together to form a completed watch. If Hora was interrupted, previously completed subassemblies would not be affected. If the probability of being interrupted while adding a part to a watch is 0.01, a simple calculation (see Appendix A) shows that it will take Tempus on average 4000 times as long to complete a watch as Hora.

1.2.2. Computer Software

A decomposed or "modular" computer program is seen to be superior to a monolithic program. Yourdon (1975, p. 97) outlines the arguments in favour of modularity as follows[4]:

a. A modular program is easier to write and debug. Functional components can be written and debugged separately.

b. A modular program is easier to maintain and change. Functional components can be changed, rewritten, or replaced without affecting other parts of the program.

c. A modular program is easier for a manager to control. For example, more difficult modules can be given to the better programmers.

Most authors advocate the use of decompositions which exhibit high cohesion within modules and low coupling between modules. Methodologies for achieving this goal vary greatly in both scope and degree of rigor. Several authors suggest "rules of thumb" for decomposing computer programs (Stevens, Myers and Constantine, 1974; Myers, 1978; Yourdon and Constantine, 1979) and some define rules for ensuring that a given decomposition is consistent (Hamilton and Zeldin, 1976). Since a computer program is a system, some

---

[4] Yourdon (pp. 97-99) also describes some performance related arguments against the use of modular programs. Subroutine calls consume CPU time. Working storage allocations for each module may cause a modular program to require more space than an equivalent monolithic program. In virtual memory systems where only some modules may be in physical memory at one time, some time may be wasted while waiting for the operating system to retrieve required modules from disk.

insights into a general theory of system decomposition might be gained through a close examination of this body of knowledge.

1.2.2.1. Characteristics of Good Decomposition

It is generally acknowledged that large computer software systems tend to be difficult to maintain or change[5] (Bubenko, 1986, p. 292). Parnas (1972, p. 1058) writes that when designing a program "one begins with a list of ... design decisions which are likely to change". Myers (1975) suggests examining the impact of future maintenance in order to determine the "best" structure for software. This impact is determined by the number of related changes to the system made necessary by coupling between its modules. However, modules must be defined before such an analysis can be performed. Myers' search for the best decomposition is conducted by trial and error. The relationship between decomposition and software maintenance is examined in the light of a general theory of decomposition in Chapter 4.

Several suggested properties of good computer program decompositions will be discussed in this section. Techniques for producing decompositions which possess these properties will be described in the next section.

1.2.2.1.1. Myers

Myers (1975) appears to have been the first to propose a framework for analyzing coupling and cohesion within an existing program. He identifies five forms of coupling and seven forms of cohesion. They are defined in order of decreasing desirability and have been included as Appendix B. The ranking is Myers' and was derived from experience.

"Measurement" of coupling and cohesion is largely arbitrary. Myers (1975) develops a quantitative measure of the independence of two modules based on the type of coupling between them. This measure of decomposition quality depends on a subjective assignment of weights to the various forms of coupling and cohesion. He uses a matrix of probabilities to express the overall modular independence of a decomposition. These probabilities represent the likelihood of a change to one module forcing a change to another module. The matrix could

---

[5] The term "maintenance" shall refer to both the correction of implementation errors and the modification or enhancement of software.

be used to generate scores permitting quantitative comparison of two decompositions. A designer could use these scores to guide system decomposition in a post-hoc manner. Ideally, a design methodology should force the first design of a system to be correct. Of course, this would require a definition of decomposition correctness.

The levels of coupling and cohesion within a program are related. Page-Jones (1980) claims that lower coupling tends to result in higher cohesion within modules.

1.2.2.1.2. Parnas

Parnas (1972, p. 1056) introduces the concept of information hiding. Ensuring that higher-level modules do not have unnecessary knowledge about the internal workings of lower-level modules is an important step in the reduction of coupling. He also compares alternative decompositions by examining the impact of future modifications. "Good" decomposition results from minimizing this impact.

1.2.2.1.3. Yourdon and Constantine

Yourdon and Constantine (1979, chapter 9) suggest a number of decomposition heuristics. Rules affecting module size, span of control, fan-in, scope of effect, and scope of control are suggested as a basis for judging the quality of a decomposition. All of these are described below.

Module Size

Module size has been discussed extensively in the literature. One early suggestion for module size comes from Baker (1972). He suggested that modules should be no longer than 50 statements, so that they could be shown on a single page of a printer listing. Weinberg (1970) showed that programmer comprehension of a module is reduced if the size exceeds about 30 lines. Yourdon (1975, pp. 94-95) mentions a number of other module size recommendations proposed by other researchers and practitioners. Many of these recommendations are incompatible.

a. Modules should fit into 4096 bytes, or 512 words, or 1024 words, or 2048 words of memory, etc.

b. Modules are anything that can be written and debugged by one programmer in one month.

c. Modules should be no more than 100-200 statements in length. (This suggestion is attributed to Larry L. Constantine.)

d. Modules should consist of no more than 20 high-level language statements.

e. Modules should be no longer than 500 COBOL statements.

When discussing the construction of hierarchical program structures, Steward (1987, p. 98) suggests that no limb have more than 5 to 7 branches off of it[6]. The lowest level of Steward's structures correspond to program code. Therefore, he is suggesting a maximum module length of 5 to 7 statements. He cites Miller's Principle (Miller, 1956) which asserts an upper limit to the number of concepts a human being may consider simultaneously. However, it is not immediately obvious why a programmer should be expected to consider all the statements of a module simultaneously.

Steward also claims that high cohesion is indicated "by whether what is done within the module can be given a short description" (p. 98). Myers (1978) defines a module which has functional cohesion (his most desirable form of cohesion) as a module which performs a single specific function. Similarly, a common rule for module size suggests that a module should consist of a single functional idea. Unfortunately, the phrase "single functional idea" is difficult to define. While this rule is superior to any of the size maximums mentioned earlier, it still contains an undesirable degree of arbitrariness. Any module size rule based on function is bound to be language dependent. Alexander (1967, p. 205) suggests a module to deal with the acoustical requirements of a system to illustrate this problem. It could be argued that the term acoustics "is not arbitrary but corresponds to a clearly objective collection of requirements -- namely those which deal with auditory phenomena.

---

[6] Steward exempts CASE structures from this rule.

But this only serves to emphasize its arbitrariness. After all, what has the fact that we happen to have ears got to do with the problem's causal structure?".

If anything is clear from the above suggestions it must be that there is no consensus as to either the optimal or maximal size of a module.

Span of Control

Span of control refers to the number of immediate subordinates[7] to a module. Yourdon suggests that spans of control of two or less or more than ten should be carefully reconsidered. He claims that abnormally small or large spans of control are usually indicators of poor design. Small spans of control correspond to either insufficient decomposition at the subordinate level or too much decomposition at the superordinate level. Large spans of control are usually the result of a failure to define intermediate levels in the decomposition. No theory or empirical evidence is presented to support this heuristic.

Fan-in

Fan-in refers to the use of modules at more than one point of the program's structure. The use of these common modules reduces the amount of programming effort required. There is clearly a trade-off between module simplicity and generality. For example, consider a point of sale (POS) inventory system. A single module could be written to handle all forms of input to the system. This module could be called from any point in the overall structure. However, some designers might argue that such a module would be unnecessarily complicated. Input from a POS terminal and keyboard input from, say, the receiving dock could be sufficiently different to warrant separate modules.

---

[7] A module X is subordinate to module Y if Y controls the activation of X. Activation may be accomplished by a subroutine CALL statement, for example.

Scope of Effect and Scope of Control

Scope of effect refers to the location of decision events in the program's structure. The scope of effect of a module is the collection of all modules containing any processing that is conditional upon the processing in that module. The scope of control of a module is the module itself and all of its subordinates. Yourdon and Constantine (1979, p. 178) state "for any given decision, the scope of effect should be a subset of the scope of control of the module in which the decision is located". In other words, any modules that are affected by a decision should be subordinate to the module which makes that decision. Again, no theory or empirical evidence is presented to support this heuristic.

1.2.2.1.4. Cluster Analysis

Hutchens and Basili (1985) have proposed the use of cluster analysis to analyze the structure of an existing computer program. All cluster analysis algorithms require the definition of a similarity or difference measure. This measure represents the "distance" between modules and is the basis for decisions to group modules together. Hutchens and Basili suggest several such measures based on data bindings[8]. There is no theoretical reason for the selection of one measure over another. Clustering algorithms are also sensitive to the "black hole" effect. As more and more modules are combined into a single cluster, the number of linkages to other not yet clustered modules increases. This means that modules and small clusters are more likely to be grouped with growing super-clusters than with each other. The fine structure of the system may be obscured. Weighting schemes can be used to reduce this effect, but a suitable assignment of weights must usually be found by trial and error. It is also interesting to note that, in order to perform cluster analysis, it is necessary to remove common modules as they cause disparate subroutines to cluster at low levels in the hierarchy.

---

[8] A data binding exists when two modules reference the same variable.

1.2.2.2. Decomposition Techniques

Several computer program decomposition techniques have been proposed. The Structured Design (Myers, 1975, 1979; Yourdon and Constantine, 1979) literature describes several decomposition heuristics. Myers defines source/transform/sink (STS) decomposition, transactional decomposition, and functional decomposition. Yourdon and Constantine define transform analysis and transaction analysis. Hamilton and Zeldin (1976) have developed a methodology based on an analysis of the inputs and outputs of a module. Their decomposition rules are embodied in constructs called JOIN, INCLUDE and OR. These constructs are referred to as "primitive control structures". In addition, Mili, Desharnais, and Gagné (1986) have formally defined the process of program decomposition as performed by programmers.

1.2.2.2.1. Structured Design Decomposition Techniques

1.2.2.2.1.1. STS Decomposition and Transform Analysis

STS decomposition is Myers principal decomposition technique. Transform analysis, as defined by Yourdon and Constantine (1979) is essentially identical[9]. The steps for applying STS decomposition to a high-level module are as follows:

a. Outline the structure of a module.

b. In this module structure, identify the major stream of input data and the major stream of output data.

c. Identify the point in the module structure where the input data stream last exists as a logical entity and the point where the output data stream first exists as a logical entity.

---

[9] This decomposition technique was first outlined in Stevens, Myers and Constantine (1974).

d. Using these points as the dividing points in the module structure, describe each division of the problem as a single function. These become the functions of the immediate subordinate modules.

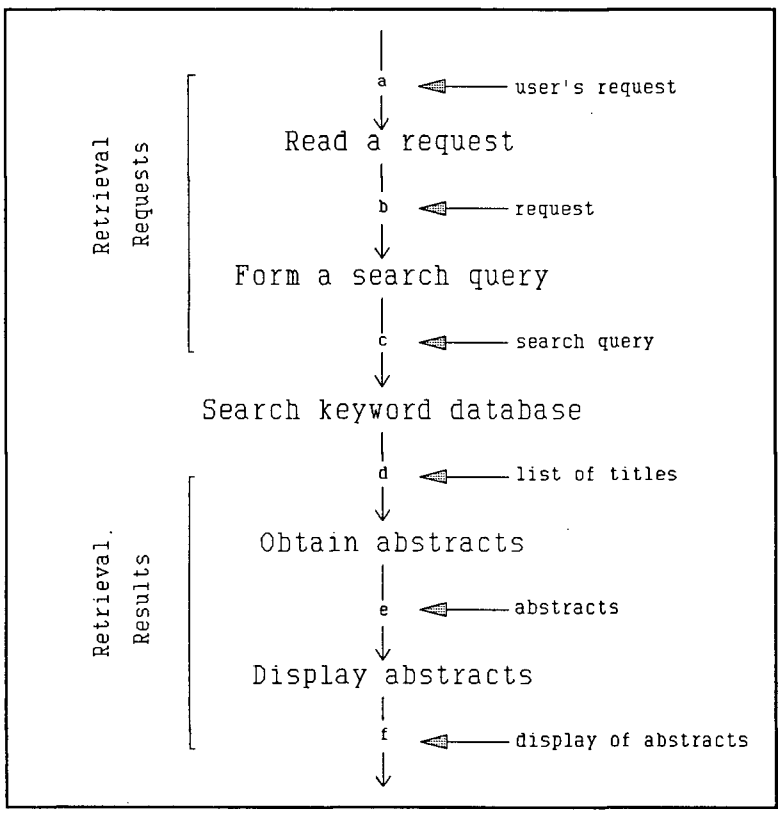


Figure 1: A program structure for illustrating Myers' STS decomposition.

For example, consider a module which accepts a request to search an abstracts database by keyword and then displays selected abstracts. The structure of the module might be as illustrated in Figure 1. The major (and only) input stream consists of user requests "a". The major (and only) output stream consists of the retrieval results "f". Point "c" is the last point where the input stream exists as a distinct entity. At point "d" there exists a list of abstract titles retrieved from the keyword database. There is a one-to-one correspondence between the final retrieval results and this list of titles. Therefore, Myers claims that point "d" is where the output stream first exists as a distinct entity. The module would then be broken into three submodules. The "source" submodule would read a request and form the search query. The "transform" submodule would search the keyword database. The "sink" submodule would obtain and display the selected abstracts. Yourdon and Constantine's transform analysis appears to be identical to STS decomposition. They refer to locating point "c" as identifying the "afferent data elements" and locating point "d" as identifying the "efferent data elements". Afferent data elements are defined as follows:

> Afferent data elements are those high-level elements of data that are furthest removed from physical input, yet still constitute inputs to the system.

Efferent data elements are similarly defined, but for outputs. Clearly, the points at which the original module is to be split are subject to some degree of ambiguity. For example, an argument could be made for splitting the above example at points "c" and "e" as it is only at point "e" that the final result is clearly seen. Yourdon and Constantine (1979, p. 194) are aware of this problem, but claim that experienced designers will not differ by more than one or two transforms (i.e. functions) when identifying afferent and efferent data elements.

It should be noted that all decomposition techniques discussed in this section, including STS decomposition, are intended to be applied recursively to the newly created submodules. This recursion is to be carried out until the lowest-level modules may be easily converted into code.

1.2.2.2.1.2. Transactional Decomposition and Transaction Analysis

Myers' transactional decomposition is applied when a module takes the form of a selection process. If a module receives different types of transactions, and the processing which follows is dependent on the type of transaction received, the module is a candidate for transactional decomposition. For example, a module whose purpose is to process a merchandise transaction might be decomposed as shown in Figure 2.

Transactional decomposition is similar to Yourdon and Constantine's transactional analysis. However, they introduce the concept of a "transaction center". A transaction center must be able to

a. obtain transactions in raw form,

b. analyze each transaction to determine its type,

c. dispatch each type of transaction, and

d. complete the processing of each transaction.

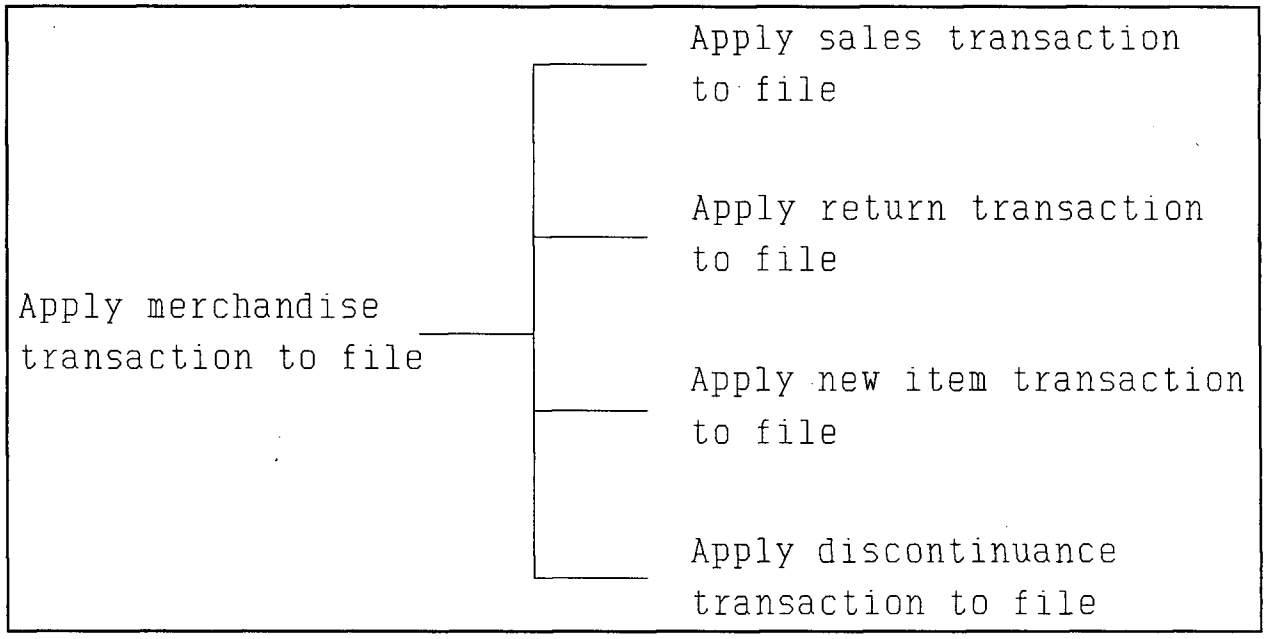


Figure 2: An example of Myers' transactional decomposition.

Myers would apply STS decomposition prior to transactional decomposition in order to identify the modules concerned with getting the transaction and determining its type.

Yourdon and Constantine also provide an operational definition of a transaction.

> A transaction is any element of data, control, signal, event, or change of state that causes, triggers, or indicates some action or sequence of actions.

This definition makes it apparent that transactional decomposition may be applied in cases where there is no "traditional" transaction evident as there was in the above example.

1.2.2.2.1.3. Functional Decomposition

Myers describes functional decomposition as "an ad hoc process of pulling single subfunctions from a module to achieve certain purposes". He suggests two possible purposes:

a. Isolating common functions, and

b. Isolating functions within informational cohesion modules.

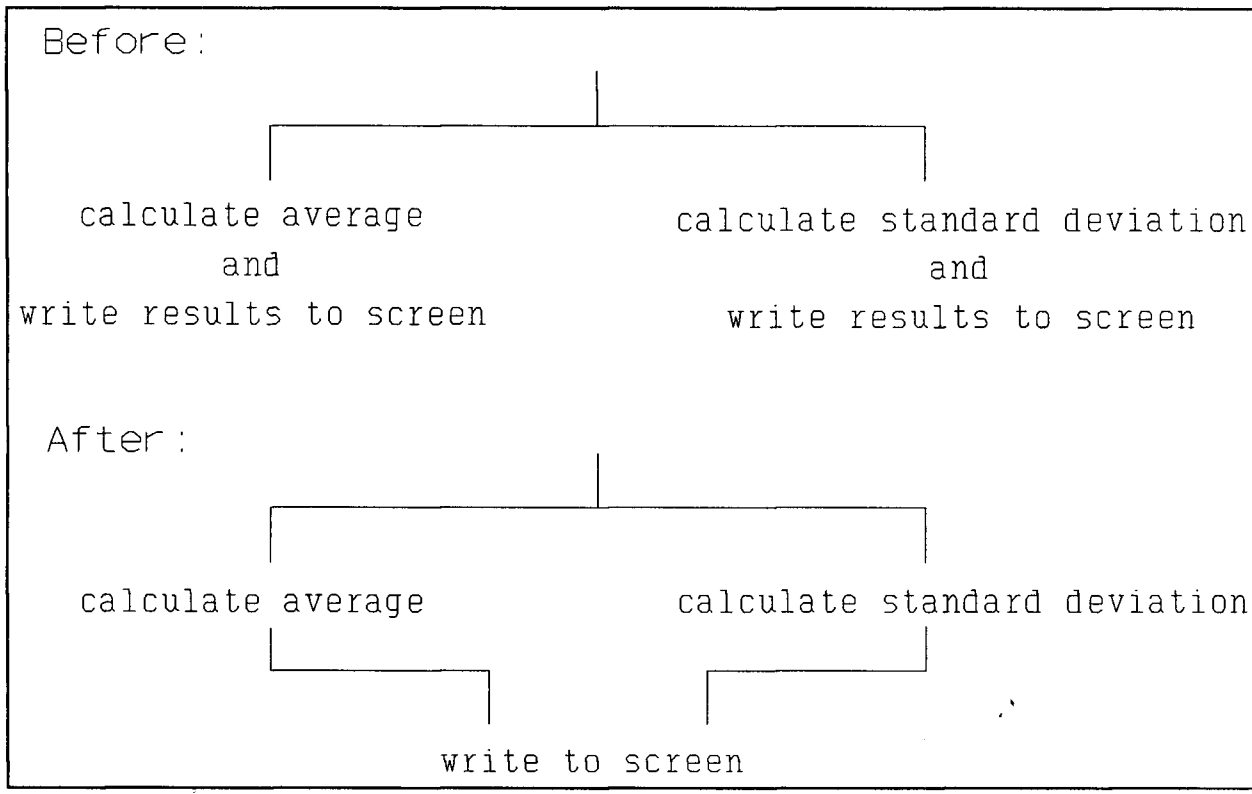


Figure 3: Myer's functional decomposition: Isolating common functions.

The first purpose reflects the desirability of removing a subfunction which is contained in a number of larger modules, and making it into a separate module referenced by each. For example, the modules shown in the "before" part of Figure 3 might be restructured as shown.

The second purpose refers to splitting a function which references a number of data structures into modules which reference only one data structure each. An informational cohesion module is one that hides "some concept, data structure, or resource" (Myers, 1978, p. 37). Modules with informational cohesion are considered as desirable as ones with functional cohesion. Myers' provides the following example of a situation where this splitting is desirable. Suppose there exists a module whose function is "build table of underpaid employees". The module sequentially examines the personnel file, and if an employee meets the underpaid criteria, it places the employee's name in the table. The

structure of the module might be as shown in the "before" part of Figure 4. Myers would not have applied STS decomposition to this module "because its logic is easily visualized". Easy visualization is Myers' decomposition stopping criteria. The first and the last subfunctions refer to separate data structures: the personnel file and the output table. Functional decomposition of the above would lead to structure shown in the "after" part of Figure 4. The two newly created modules could be combined with other modules referencing the data structures, thus either adding to or creating informational cohesion modules. Testing to determine whether a given employee is underpaid would be performed in the "Build table of underpaid employees" module.

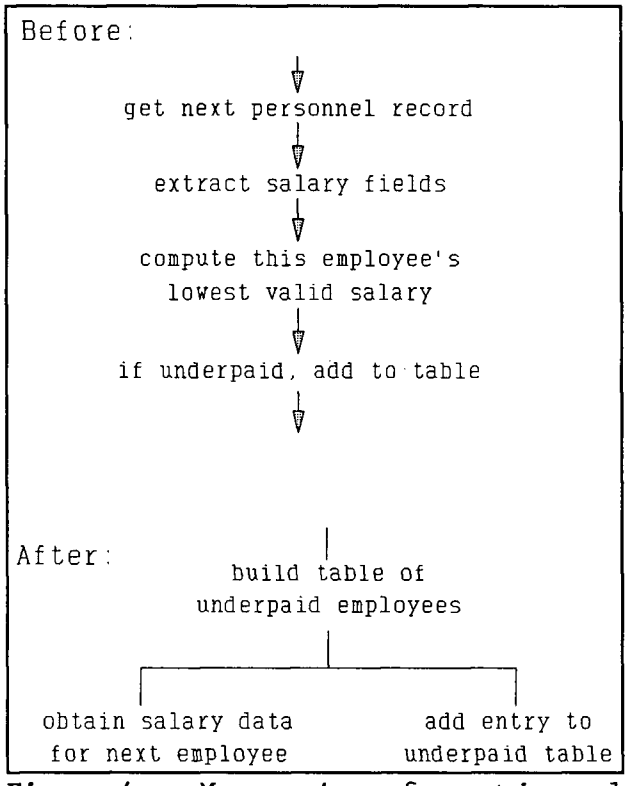


Figure 4: Myers' functional decomposition: Creating informational strength modules.

Yourdon and Constantine do not refer to any decomposition method which is analogous to Myers' functional decomposition.

1.2.2.2.2. HOS Decomposition

The HOS design methodology developed by Hamilton and Zeldin (1976) is capable of generating computer code through the use of "mathematically provable constructs". Specifically, three primitive control structures are identified: JOIN, INCLUDE, and OR. The HOS methodology does not provide specific techniques for actually performing system decomposition. The methodology provides formal tools for ensuring that a given decomposition is consistent with certain axioms governing the relationships between modules. Therefore, the HOS methodology can provide some insights into the nature of "good" decomposition, but cannot add to the decomposition heuristics found in the Structured Design literature.

JOIN is used to support the decomposition of a function into two sequentially executed subfunctions. The outputs of one module must be inputs

to the other. For example, if the function of the original module was "make a stool", it might be decomposed to "make parts" and "assemble parts". HOS uses a functional notation combined with a binary tree to represent decompositions as shown in Figure 5a. Inputs to the system are TOPWOOD and LEGWOOD and the system's output is STOOL. The output from the first, or right most, module is TOP and LEGS. TOP and LEGS form the inputs to the second module. JOIN is analogous to STS decomposition when no transform module is identified. The original module will be broken into only two submodules.

If some set of desired outputs can be obtained in more than one way, OR is used to separate the methods. For example, if LEGS can be constructed from either HARD wood by turning or SOFT wood by carving, a "make legs" function could be decomposed as shown in Figure 5b. OR is similar to transactional decomposition. Its use implies that one and only one of the identified subsystems may be activated by a single transaction.

INCLUDE is used to separate independent subfunctions. For example if the functions "make legs" and "make top" were independent of one another, the function "make parts" could be decomposed as shown in Figure 5c. This sort of decomposition is neither STS nor transactional. Nor does it fall under either of the circumstances Myers suggests for functional decomposition. Perhaps, Myers and Yourdon and Constantine considered this form of decomposition too obvious to mention. That is, if a module consists of subfunctions which do not interact with each other in any way, separate them.

The HOS methodology has been lucidly described by Martin (1985). He claims that

> "The technique has been automated so that bug-free systems can be designed by persons with no knowledge of either mathematics or programming. The software automatically generates bug-free program code. Whereas most mathematical techniques have been applied only to small programs. Hamilton and Zeldin's technique has been used successfully with highly complex systems. The technique is used not only for program design but, perhaps more important, for high-level specification of systems. The design is extended all the way from the highest-level statement of system functions down to the automatic generation of code." (pp. 39-40)

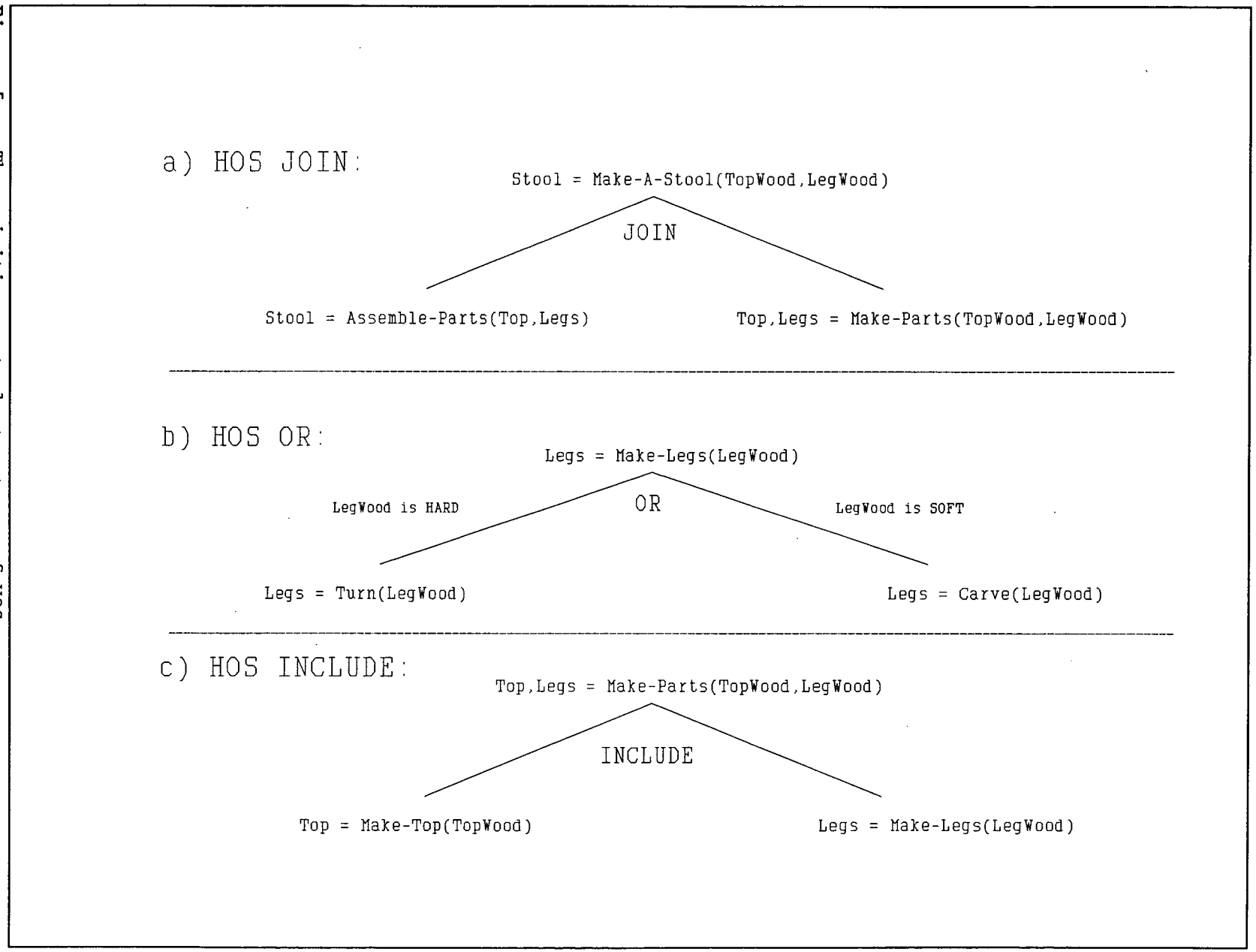


Figure 5: The primitive control structures of HOS.

Although Martin is clearly concerned with "selling" HOS, there is no doubt that it represents a major departure from the relatively informal methodologies of Structured Design.

1.2.2.2.3. Formal Models of Computer Programming

Mili, Desharnais, and Gagné (1986) describe three formal models of the process of computer programming. They present three formalisms for program specifications:

a. As a pair of assertions (p,q), where p, the input assertion, defines the set of admissible input states and q, the output assertion, defines the set of correct output states.

b. As a function mapping admissible input states into correct final states.

c. As a relation containing all the pairs (input/output state) considered to be correct by the specifier.

The third formalism, and its associated relational decomposition, is quite similar to the one developed in the remainder of this document. Mili et al. define a relation R as a subset of S X S, where S is the set of all possible states of a program[10]. That is, R is a subset of all possible pairs (s,s′) where the input state s and the output state s′ are elements of S. A program specification can be described by a relation where each tuple consists of a input/output state pair. Three rules for program decomposition are defined. These rules ensure that an original relation can be reconstructed from a set of simpler[11] relations. It is the programmer's responsibility to find the simpler relations. No procedure for obtaining these simpler relations is described.

---

[10] Program states reflect the values of the program's variables. For example, if a program contains three variables "a", "b" and "c", a state s of the program could be represented by a triplet of values <a(s),b(s),c(s)> where a(s) is the value of variable "a" when the system is in state s, etcetera. If "a" is an integer variable and "b" and "c" are real, possible states of the program might include <3,2.1,3.1>, <1,0.1,0.2>, and <0,0.1,0.2>.

[11] That is, less complex code is required for implementation.

Explanations of the decomposition rules are relatively involved and have been included as Appendix C. The rules are described briefly, and informally, below.

Sequence Statement Rule

A relation R may be decomposed into two relations R_1 and R_2 where:

1) The input states of R_1 are the same as the input states of R.
2) The output states of R are the same as the output states of R_2.
3) The output states of R_1 are the same as the input states of R_2.

Alternation Statement Rule

A relation R may be split into two smaller relations R_1 and R_2 where R_1 consists of those states of R which satisfy some condition, and R_2 consists of those states of R which do not.

The Iteration Statement Rule

Decomposition by iteration involves finding a relation which, when applied to itself recursively using the sequence statement rule, will yield the original relation.

It is difficult to see how the iteration rule qualifies as a form of decomposition. The rule is primarily intended to avoid coding a large number of input/output pairs, by applying a smaller amount of code iteratively. Iteration can be viewed as a programming tool used to save source code space and programmer typing time. Decomposition using the iteration rule need not produce programs which are more easily visualized than a program implementing the original relation R. In fact, the operation of iterative programs can be much harder to grasp than equivalent, but longer, "linear" programs.

1.3. Conclusions

Decomposition is recognized in the general systems theory literature as an important tool for both systems analysis and design. Systems exhibiting "good" decompositions are seen to be more stable than monolithic systems.

However, Simon and Ando's short-run and long-run dynamic criteria appear to be the only clear contributions to an understanding of what constitutes "good" decomposition. On the other hand, computer program decomposition has long been a major problem in software engineering, and as a result some practical solutions have been devised.

There are three basic types of computer program decomposition. STS decomposition, transform analysis, HOS JOIN, and the sequence statement rule refer to the separation of sequentially activated functions into separate modules. These techniques could be referred to as "sequential decomposition". Transactional decomposition, transaction analysis, HOS OR and the alternation statement rule are used to decompose a set of functions that are activated conditionally. These techniques could be referred to as "conditional decomposition". The HOS INCLUDE can be used to separate functions that are independent of each other. This technique could be referred to as "parallel decomposition".

The Structured Design methodologies of Myers and Yourdon and Constantine, are derived from experience and require human intelligence for their application. They place some structure on the process of finding the lower-level modules, but their precise definition is left to the program designer. The dictum stating that a module should contain at most one functional idea is both highly subjective and language dependent. Myers' framework of coupling and cohesion along with his measure of a decomposition's quality is only useful after the system has been coded.

The HOS methodology does not consider how a module is to be decomposed. Rather, it is concerned with ensuring that the decomposition is good with respect to the HOS axioms, namely, it can be represented using primitive control structures.

The decomposition rules of Mili et al. can be used to ensure that given modules can be combined to form the original program. They do not provide an algorithm for finding the modules.

The next chapter describes a system modelling formalism which will support both automated decomposition and completeness and consistency verification.

Chapter 2: System Modelling

2.1. Introduction

Bubenko (1986, p. 289) notes that the practice of information systems analysis and design is characterized by hundreds of different methodologies. Yet there is general agreement that most large information systems are difficult to maintain and change, and that assessing their correctness and completeness is usually impossible (p. 292). Several undesirable characteristics possessed by many methodologies are identified (p. 298). These include the following:

a. **Fuzzy Concepts**

Many of the concepts advocated in analysis and design methodologies are not well defined. It is difficult to know which ones to use, and how to use them in varying, non-trivial design situations.

b. **No Verification**

There is usually no way to verify the correctness, completeness, and consistency of conceptual specifications.

Examples of poorly-defined concepts include: system, decomposition, subsystem, statics, and dynamics. It is impossible to develop a theory of system decomposition without exact definitions of these concepts. The main purpose of this chapter is to present a formalization of the modelling constructs deemed necessary for the automation of system decomposition. These constructs have been implemented in the form of computer-based specifications analysis tools. The tools are described and their use will be demonstrated using two rather simple examples in later sections of this chapter. A more complicated "real" system will be examined in Chapter 6.

There are no generally accepted definitions for correctness, completeness, and consistency with respect to system models. Roman (1985) claims that "a requirements specification is complete if some relevant aspect has not been left out and is consistent if the parts of the specification do not contradict each other. Both completeness and consistency require the existence of criteria against which one may evaluate the model. Completeness and consistency checks...presuppose the analyzability of the requirements by mechanical or other

means. The higher the degree of formality the more likely it is that requirements may be analyzed by mechanical means." (p. 16). It is not surprising that few methodologies provide for any form of verification since few produce formal requirement specifications. Definitions of completeness, consistency and correctness will be suggested in this chapter. These definitions are sufficiently formal to allow computerized analysis. Tests for completeness and consistency have been included in the specifications analysis tools.

Bubenko (1986, p. 298) also notes that there appears to be an underlying assumption, among the various analysis and design methodologies, that "in the early stages, conceptual specification and analysis of the behaviour (dynamics) of the information system is less important (for the purpose of understanding) than the description of its 'structure'". It is not clear how any analysis of structure can be performed without some knowledge of behaviour. Part of a system's structure consists of a collection of objects[12]. In the object-oriented programming literature, Nierstrasz (1987) notes that "perhaps the most difficult task is deciding how to naturally decompose a problem into objects" (p. 11-12). In order to separate two objects in a system, the analyst must be aware of a circumstance in which the objects behave independently. For example, consider an employee's first and last names in a personnel system. If a decision is made to represent them as separate objects, the analyst must know that they could be separated. The analyst must know of some process which does not require both parts of an employee's name. This knowledge could come from his or her understanding of the system's operation or previous experience. In Chapter 4, it will be argued that previous experience is not a sufficient basis for good design decisions. The formalism presented here explicitly models system dynamics, and decomposition decisions (as described in the next chapter) are based solely on the characteristics of the system as described by the analyst.

[12] "Objects" as used in this research are related to the objects of object-oriented programming, but they are not identical. The relationship shall be discussed in Chapters 3 and 6.

2.2. The Formalism

2.2.1. General

The process of systems analysis and design can be viewed as a three-stage

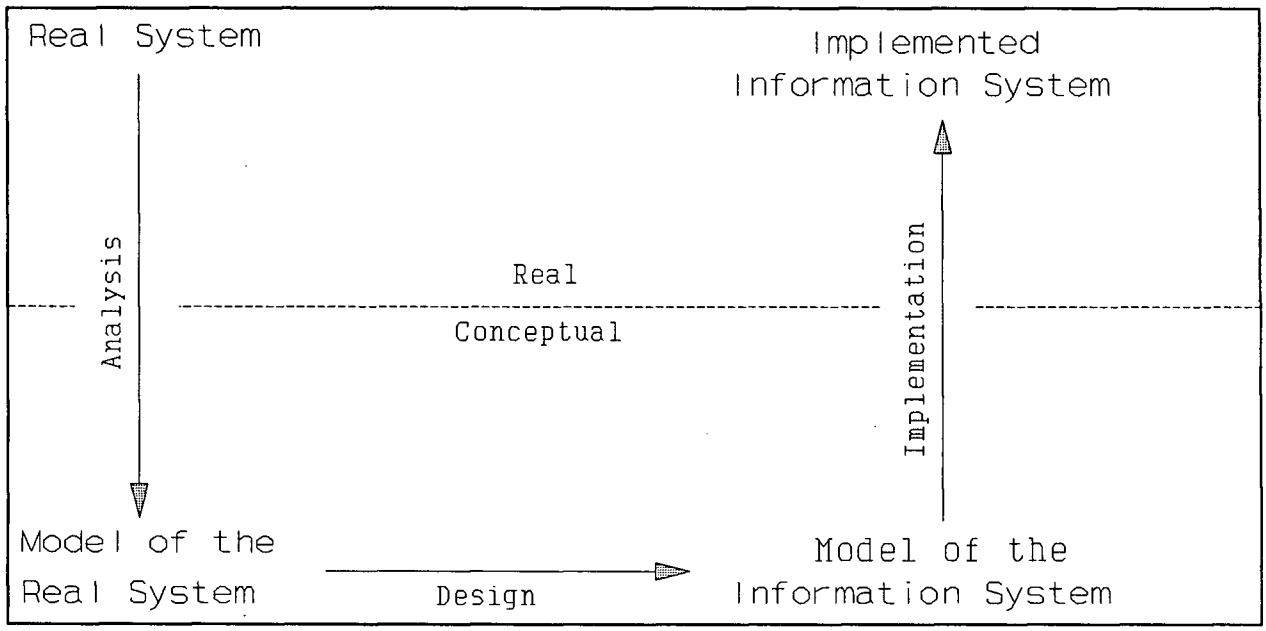


Figure 6: The system analysis and design process (adapter from Wand and Weber, 1988).

transformation as shown in Figure 6. To illustrate these stages, consider an analogy with the design and construction of an office building. An architect creates a set of drawings and specifications which reflect the desires of his or her client. An engineer translates these into a detailed plan for the construction of the building. Finally, a contractor constructs the office building itself. In this example, the "real system" consists of the client's desires. The activities of the architect are called "analysis". The architect's drawings and specifications are a "model of the real system". The activities of the engineer are called "design". The set of detailed construction drawings forms a model of the office building and is analogous to a "model of the information system". The activities of the contractor are called "implementation" and the office building itself is analogous to an "implemented information system".

This research is primarily concerned with the transformations from the "real system" to the "model of the real system", and from the "model of the real

system" to the "model of the information system", that is, with both the "analysis" and "design" transformations. System decomposition is the process by which an analyst identifies the parts of the real system which should be reflected in his model. These parts, and their relationships with each other, have a direct influence on the structure of the information system.

2.2.2. Systems

In order to support automated system decomposition a modelling formalism must be able to represent the following:

a. The parts of the system[13],

b. The allowed states of the system, and

c. The manner in which these states may change.

The first requirement refers to system statics; the last two refer to system dynamics. Most existing analysis and design methodologies meet these requirements at least implicitly. However, the basic constructs of most methodologies (with the notable exception of HOS) are not clear. An important premise of this research is that understanding of system properties, in particular decomposability, will be greatly facilitated by carefully defining what we are studying. That is: What exactly is a system and what governs its behaviour?

2.2.3. An Intuitive Beginning

The system modelling formalism used in this research is largely based on the works of Bunge (1978, 1979) and Wand and Weber (1988, 1989). The system

[13] The modelling formalism selected for use in this research does not deal directly with the "parts" or things belonging to a system. As will be shown, only knowledge of the properties which are used to describe the things is required to specify a system. However, at this stage it may be more convenient to visualize a system based on a collection of things, rather than a set of properties.

modelling approach based on the formalism will be called SELMA (for States, Events, and Laws Modelling Approach)[14].

One of the goals of system decomposition is the identification of the objects comprising a system. As shall be illustrated in the next chapter, there is no unique set of objects describing most interesting systems. In general, labelling of the objects from which a system is constructed depends upon the analyst's point of view. Von Bertalanffy (1974) notes that identification of objects in the real world is not a trivial task. "The spatial boundaries of even what appears to be an obvious object or 'thing' actually are indistinct. From a crystal consisting of molecules, valences stick out, as it were, into the surrounding space; the spatial boundaries of a cell or an organism are equally vague because it maintains itself in a flow of molecules entering and leaving, and it is difficult to tell just what belongs to the 'living system' and what does not. Ultimately all boundaries are dynamic rather than spatial." (p. 22). There is a real danger that an analyst may be tempted to decompose a system on the basis of spatial relationships (i.e. relative positions in space). As will be further discussed in the next chapter, if component objects discovered in this way are used to form the structure of the information system, it is likely that alternative, and possibly superior, structures will not be considered. Spatial relationships are primarily static in nature. The theory of decomposition presented in the next chapter is based on an analysis of both system statics and dynamics.

There is no generally accepted definition for the term "system" and it will not be rigorously defined here. In the modelling formalism, and in the theory of decomposition presented in the next chapter, "the system" shall mean whatever collection of objects and processes the analyst chooses to consider. A system is described by properties[15] and relations between these properties. Of course, a system may itself be considered to be an object, and as such suffers from the same identification problems discussed above. It is assumed that the system is

---

[14] Just as a point of interest, "selma" is derived from the Arabic word for "secure" and is the short form of "anselma" which is Old Norse for "divinely protected" (Browder, 1987, p. 185). Given that automation of consistency and completeness testing is one of the major advantages of SELMA over other modelling schemes, these are not entirely inappropriate meanings.

[15] These properties will also describe the things from which the system is composed. However, the modelling formalism is not concerned with identification of the component things of a system.

described by a well defined set of properties, and that all relevant[16] interactions between the system and the rest of the universe (i.e. its environment) are known.

It should be noted that Bunge (1979, p. 6) defines a system as an object consisting of at least two different connected[17] things. This definition was found to be too restrictive. For the purposes of system modelling, it is sufficient to accept as a system anything which the analyst claims is a system. As far as this research is concerned, it does not matter whether each component of the system is connected directly, or indirectly, to any other component of the system. For example, consider a system consisting of two independent subsets of things, but where the things in each subset are interconnected[18]. As we shall see, the decomposition algorithm (described in the next chapter) will find that the system consists of two independent subsystems[19]. The stance taken here is that analysts know what systems are and that too detailed a definition will only confuse matters.

---

[16] Relevant to the purpose of the analysis effort.

[17] Bunge also defines the term "connected". Unfortunately, any discussion of connection or interaction degenerates into a discussion of causality. Such a discussion is not appropriate here.

[18] Normally there should be some reason to consider independent subsets as parts of the same system. Perhaps the independent subsets describe are parts of another subsystem defined at a higher level of abstraction. That is, using the terminology to be introduced in the next chapter, the two subsets may each contribute input state variables to a subsystem which determines the value of some emergent state variable at a higher level.

[19] Note that in reality the subsystems may not be independent. It could be argued that in some sense all parts of the universe are interconnected. However, it is possible for two subsystems to be independent with respect to a particular model. The model is a man-made abstraction of some aspects of the real world. Not all interactions will be described.

2.2.4. Definitions

2.2.4.1. The Basics

Definition: System State

At a given time, the values attained by the properties of a system σ comprise a STATE s of σ.

Definition: State Variable and Value

State variables are the properties required to describe some part of the real world for some given purpose[20]. A system σ is that part of the real world described by the set of STATE VARIABLES $\{v_1,\ldots,v_n\}$ selected by an analyst[21]. A state variable is a function mapping the set of all system states into the set of VALUES. That is, the value of state variable v_i at time t is $v_i(t)$. For example, a state variable called "employee-type", describing some part of a personnel system, might have values of "full-time" or "part-time". A system state s can be represented by a vector of state variable values.

$s = [v_1(t),\ldots,v_n(t)]$

Definition: Possible State Space

The POSSIBLE STATE SPACE S of a system σ is the Cartesian product of the sets of all possible values of each state variable of σ. Bunge (1979, p. 20) calls this space the "conceivable state space of σ". For example, consider a system which can be described by three state variables, "a", "b" and "c". A state s of this system could be described by the vector [a(t),b(t),c(t)], where a(t), b(t), and c(t) are functions returning the values of state variables a, b, and

---

[20] This implies an appropriate choice of level of abstraction. That is, it is not necessary to include all properties of the part of the real world being modelled. For example, when modelling a company's payroll system, the analyst may choose not to include a state variable describing the size of an employee's desk.

[21] The concept of "system" is discussed further in Chapter 7.

c at time t respectively. If each state variable could have values of 0 and 1, the possible state space of the system would consist of [0,0,0], [0,0,1], [0,1,0], [0,1,1], [1,0,0], [1,0,1], [1,1,0], and [1,1,1].

Definition: System Law

The interactions between the properties of a system σ comprise the SYSTEM LAW L of σ (Wand and Weber, 1989). Given any state s of σ, L is a function[22] such that

$s' = L(s)$

where

$L(s) = s$ if the system may remain indefinitely in s, and

$L(s) \neq s$ if the system state must change, and s′ will be the next state of the system where $s' = L(s')$ (i.e. the law does not change the next state).

Every system has one and only one system law. This law completely defines the behaviour of the system. Full knowledge of a system law is generally impossible or a least very difficult to obtain[23]. The concept of a system law is seen as useful tool for theory building, but practical problems will require more operational definitions. These will be developed later in this section.

Definition: Stable and Unstable System States

Given a state s of a system with system law L:

If $s = L(s)$ then s is said to be STABLE with respect to L.
If $s \neq L(s)$ then s is said to be UNSTABLE with respect to L.

---

[22] This research deals only with deterministic system. That is, each initial system state transforms into one and only one final state.

[23] For example consider the physical universe. One could think of the universe as being governed by a single all-encompassing physical law, which mankind is struggling to understand through science. We currently have only a partial understanding of this law. This partial understanding is expressed by our chemical, biological and mathematical principles and laws of physics.

For example, consider a simplified accounting system described by state variables representing "account balance" and "value of assets". Assume the system law simply states that the values of the two state variables should be equal, and if they are not, the value of "account balance" must be set equal to the value of "value of assets". That is, if the values of "account balance" and "value of assets" are not equal, the system law would alter the system state by setting the value of "account balance" equal to the value of "value of assets". This means the system was in an unstable state with respect to the system law, because the law maps the original state into a different state. On the other hand, when the values of "account balance" and "value of assets" are equal, the system is in a stable state because the law is fulfilled.

Definition: Stable State Space

The set of all stable states of a system σ is called the STABLE STATE SPACE of σ.

Definition: External Event

The environment[24] acts on a system in the form of EXTERNAL EVENTS. An external event e occurs when the environment acts to set the value of some state variables within the system. This change of value might move the system into another stable state, an unstable state, or the system might remain in the same state. In other words, if s is a system state and S is the possible state space of the system, e is a function[25] of the following form.

e: {s such that s = L(s), s ∈ S} --> {s such that s ∈ S}

If the state is stable, no further state changes occur. However, if the new state is unstable, the system must respond so as to return to a stable state. These system state changes in response to external events define the system's

---

[24] The environment of a system is described by all properties of the real world which are not properties of the system.

[25] It is assumed that external events can only occur when the system is in a stable state (ie. L(s) = s).

dynamics. For example, a system initially in a stable state "$stable_{i1}$" may be moved to an unstable state "$unstable_1$" by an external event e. The system will respond by moving to another stable state "$stable_{f1}$". This is shown graphically in Figure 7a. It is also possible that the same event may move the system from a stable state "$stable_{i2}$" to another stable state "$stable_{f2}$" directly, as shown in Figure 7b.

An analyst may find it difficult to specify a monolithic system law which describes the overall behaviour of all state variables. Fortunately, a system law may be decomposed into smaller SUBLAWS, and perhaps more importantly, a system law may be synthesized from a number of sublaws.

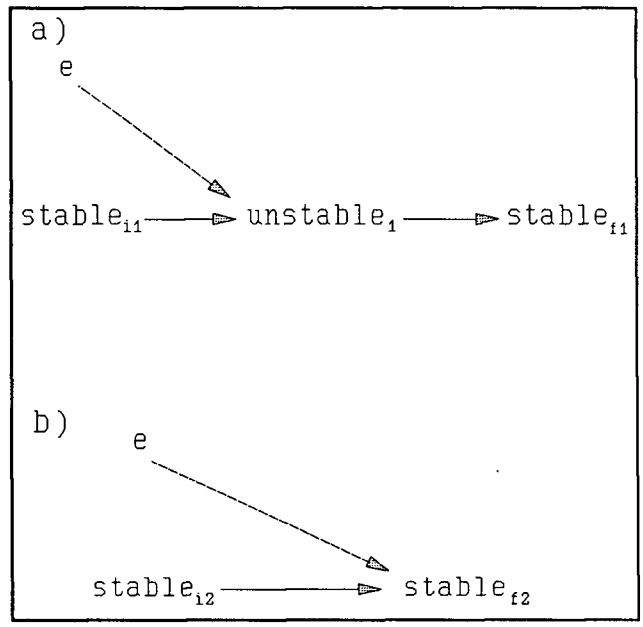


Figure 7: The action of external event "e" on a system.
a) The event moves the system into an unstable state.
b) The event moves the system into a stable state.

Definition: Sublaw

A SUBLAW l is a function defined on a subset of the state variables describing a system, such that for any stable state s of the system σ with system law L, $s = l(s)$ and for any unstable state s of the system, $s \neq l(s)$. That is,

$s = l(s)$ if $s = L(s)$
and
$s \neq l(s)$ if $s \neq L(s)$

Notice that if s is unstable, a sublaw need not map s into the same stable state as the system law. That is,

$s' = l(s)$ and $s \neq s'$ and $s'' = L(s)$ does not imply $s' = s''$.

Also notice that there are two parts to any system law or sublaw:

a. Stability Conditions

This part applies when a state s is stable (i.e. $s = L(s)$ or $s = l(s)$). The condition specifies the system states allowed by the sublaw.

b. Corrective Actions

This part applies when a state s is unstable (i.e. $s \neq L(s)$ or $s \neq l(s)$). The action specifies how the values of the state variable must change should the system enter an unstable state.

Before an example of a sublaw is provided, one more definition is required.

Definition: Rule

A sublaw may be expressed as a set of RULES. Each rule specifies a single stable condition or corrective action.

For example, a description of a very simple accounting system (with real-time asset change posting) might include the following rules.

1. The value of the "account balance" state variable must equal the value of the "value of assets" state variable.
2. The "last change status" state variable must indicate that the last change to asset value has been posted (i.e. the value of the last change to the value of the assets has been added to or subtracted from the account balance).
3. If the value of the "account balance" state variable is <u>not</u> equal to the "value of assets" state variable (i.e. the system is out of balance), then adjust the value of the "account balance" state variable to equal the value of the "value of assets" state variable, and set the value of the "last change status" state variable to indicate that the last change has been posted.

The above rules constitute a sublaw. There may be other sublaws describing other parts of the system. The first two rules specify stability conditions and the last specifies a corrective action. Notice that this sublaw assumes that the only way the system can become out of balance is by altering the value of the

assets. That is, the sublaw could not handle a situation where the account balance was changed directly, either by accident or deliberate tampering. In other words, the above rules have been formulated with a specific set of external events in mind. The system modelling tools, described later in this chapter, require explicit identification of external events so that deficiencies in the rules can be immediately identified.

Traditionally, the behaviour of office information systems has been described in terms of procedures. In SELMA, the behaviour of systems is entirely defined in terms of sublaws. Sublaws are not equivalent to procedures. Panko (1984, p. 227) defines a procedure as a program "in which there is a predetermined flow of work involving many steps, whether the flow consists of the same steps each time or involves a more complex logic flow". In a study involving the creation of computerized systems to support executive work, Panko notes that none of the executives interviewed "could articulate definite processes, much less well-defined procedures, to describe how their goals were achieved" (p. 228). The order of activation of sublaws is not predetermined. It is hypothesized that in many cases it may be easier to discover the sublaws under which an executive operates, than to determine all the procedures he or she may choose to follow. However, empirical testing of this hypothesis is beyond the scope of this research.

Before formal definitions of correctness, completeness, and consistency can be given, one more basic definition is required.

Definition: Response Path and Response Function

Let Q be the set of sublaws describing the behaviour of some system. An ordered list of sublaws $[l_1,\ldots,l_j]$ where $\{l_1,\ldots,l_j\} \subseteq Q$ is called a RESPONSE PATH and the composition of those sublaws[26]

$$P_Q(s) = l_j(\ldots l_2(l_1(s))\ldots) = l_j \circ \ldots \circ l_2 \circ l_1(s)$$

is called a RESPONSE FUNCTION defined on Q.

[26] The symbol $\circ$ is used to denote the composition of functions.

2.2.4.2. Completeness, Consistency and Correctness of Sublaws

Brooks (1987) notes that "the hardest part of the software task is arriving at a complete and consistent specification..." (p. 16). SELMA supports formal definition of completeness and consistency. As will be demonstrated later in this chapter, these definitions can be computerized to automatically test a system model consisting of state variables and sublaws. Informally, the notions of sublaw completeness and consistency can be described as follows:[27]

Completeness: All system states may be transformed to stable states by the sublaws (i.e. no states have been "left out" in the analysis of system dynamics).

Consistency: Every system state, which may be transformed to a stable state by the sublaws, may be changed into one and only one stable state (i.e. the sublaws do not contradict one another).

In addition, the notion of correctness is informally described as follows:

Correctness: Taken together, the sublaws transform the initial system states to exactly the same final states as the system law (i.e. all the sublaws combined describe the actual behaviour of the system).

Each of these definitions depends to some extent on the system law. Completeness and consistency require that stable states be identified. Stable and unstable states were defined in terms of the system law. Correctness requires that the "operation" of the sublaws be the same as the system law. Unfortunately, system laws are generally unknowable[28]. The best an analyst can hope for is an approximation to the system law in terms of sublaws. This does not imply that completeness, consistency and correctness are useless notions. While correctness is usually impossible to verify, two levels of completeness and consistency are formally defined below. At a conceptual level, global

---

[27] These informal notions are similar to those of Roman (1985, p. 16).

[28] Olivé (1983, p. 73) states "it is not possible to formally verify the validity of the conceptual model with respect to the user's 'real' requirements...".

completeness, global consistency and correctness are defined in terms of the system law. At an operational level, local completeness and local consistency are defined using a more restricted definition of stability called local stability[29].

2.2.4.2.1. Conceptual Definitions for Completeness, Consistency and Correctness

Definition: Global Completeness of Sublaws

A set of sublaws Q of system σ with system law L completely describes the behaviour under L of σ with respect to the possible state space S of σ, if for every state s in S there exists a response function P(s) which maps that state into a stable state s'. That is:

Q is globally complete with respect to S if and only if
  FOR ALL s such that $s \in S$,
    THERE EXISTS P_Q such that $s' = P_Q(s)$ and $s' = L(s')$

Notice that while s' must be stable, it need not be the same stable state into which the system law maps s (i.e. s' is not necessarily equal to L(s)). Equivalence of the sublaws to the system law is assured by sublaw correctness as defined later.

Definition: Global Consistency of Sublaws

A set of sublaws Q of system σ with system law L is globally consistent with respect to the possible state space S of σ, if all response functions which map a state s in S into a stable state, map s into the same stable state. That is:

Q is globally consistent with respect to S if and only if
  FOR ALL s, P_Q, P'_Q
    such that $s \in S$ and $L(P_Q(s)) = P_Q(s)$ and $L(P'_Q(s)) = P'_Q(s)$,
  $P_Q(s) = P'_Q(s)$

---

[29] Conceptual and operational levels are concerned with aspects of the "real system" and the "model of the information system" (as defined earlier in this chapter), respectively.

Again, notice that while the final states $P_Q(s)$ and $P'_Q(s)$ must be stable and equal, they need not be equal to the state into which the system law would have mapped s.

Definition: Correctness of Sublaws

A set of sublaws Q of system σ with system law L correctly describes the behaviour under L of σ with respect to the possible state space S of σ, if for every state s in S every response function $P_Q(s)$, which maps s into a stable state, maps s into the same state that L maps s. That is:

Q is globally correct with respect to S if and only if
FOR ALL s, P_Q such that $s \in S$ and $L(P_Q(s)) = P_Q(s)$,
$P_Q(s) = L(s)$

Global completeness and consistency are prerequisites for correctness. That is, the definition of correctness implies that every state can be mapped into one and only one stable state. However, notice that global completeness and consistency do not imply correctness. That is, the definitions of global completeness and consistency do not ensure that the mappings provided by the sublaws and the system law are the same. This observation may be expressed by the following corollary.

Corollary: If a set of sublaws Q is correct with respect to a possible state space S, then Q is globally consistent and globally complete with respect to S.

2.2.4.2.2. Operational Definitions of Completeness and Consistency

Knowledge of the system law is required to test a set of sublaws for global completeness, global consistency, and correctness. In practice, the system law governing the behaviour of most real systems is approximated by the sublaws themselves. Notice that the global completeness and global consistency conditions require only knowledge of whether a system state is stable. While knowledge of the system law is required to assess stability (i.e. s is stable

if $s = L(s)$), if a weaker definition of stable state is employed, a form of completeness and consistency testing becomes possible. Consider the following definition for LOCALLY STABLE STATE where P is some response function derived from the set of sublaws Q.

Definition: Locally Stable State

A system state s is locally stable if and only if there is no composition P_Q of sublaws Q which can map the state into a different state. That is,

s is locally stable with respect to Q if and only if
  THERE DOES NOT EXIST P_Q such that $s \neq P_Q(s)$

A weaker form of completeness, called LOCAL COMPLETENESS, of the sublaws could be guaranteed by ensuring that there exists some response function mapping each system state into a locally stable state.

Definition: Local Completeness of Sublaws

Let Q be a set of sublaws describing the system σ which may enter states S'[30], then

Q is locally complete if and only if
  FOR ALL s, $s \in S'$
    THERE EXISTS $s' = P_Q(s)$ such that s' is locally stable

A weaker form of consistency, called LOCAL CONSISTENCY, of the sublaws could be established by ensuring that all possible response paths lead to the same final locally stable state.

---

[30] S' may not equal the possible state space S. S' is the set of states, both stable and unstable, which the sublaws and external events included in the model of the system are designed to consider. As will be discussed in more detail later, the stability conditions of the sublaws define the stable states of S' and the corrective actions define the unstable states.

Definition: Local Consistency of Sublaws

Let Q be a set of sublaws describing the system σ which may enter states S′, then

Q is locally consistent if and only if
FOR ALL s, P_Q, P'_Q
such that $s \in S'$ and $P_Q(s)$, $P'_Q(s)$ are locally stable,
$P_Q(s) = P'_Q(s)$

Tests for local consistency and completeness are clearly inferior to the tests possible if the system law is known. However, local consistency and completeness testing does ensure that all known information is consistent and complete with respect to itself.

Olivé refers to local consistency and completeness as the "logical consistency of the model" (p. 73). A model is "logically consistent" if the outputs of the system are derivable from the inputs. Most systems analysis and design methodologies do not provide any way to systematically verify the logical consistency of a model. CIAM (Gustafson, et al., 1982) and DADES (Olivé, 1982) are notable exceptions. CIAM refers to tests for local completeness and consistency as "checking the satisfiability of information requirements". Each output must be expressible in terms of system inputs or information derived from those inputs. DADES refers to these tests as "derivability analysis" (p. 229). Derivability analysis is a formal method to show that outputs are derivable from inputs.

2.2.5. A Simple Example

Consider a hypothetical system consisting of four interconnected lights. Light "a" is connected in series with "b" so that if "a" is on then "b" will be on and if "a" is off, "b" will be off. If light "a" is off then light "c" will be on, and if "a" is on, light "d" will be on. Only the state of light "a" may be set manually. The schematic diagram of a system implementation using digital logic and light emitting diodes is included as Appendix L.

The system may be described by the following state variables and states. The "on" state of a light will be represented by the integer 1 and the "off" state by 0.

State Variables	States
a	1 or 0
b	1 or 0
c	1 or 0
d	1 or 0

One of many possible sets of sublaws, describing the stable states of the system and the actions to be taken should the system find itself in an unstable state, is given below. The corrective action rules are numbered for future reference when describing system response paths.

Sublaws

1. Stability Conditions:

a	**b**
0	0
1	1

Corrective Actions:

	Conditions		Actions
	a	-->	**b**
R1:	1		1
R2:	0		0

2. Stability Conditions:

a	**c**
0	1
1	0
1	1

Corrective Action:

	Conditions		Actions
	a	-->	**c**
R3:	0		1

3. Stability Conditions:

a	**d**
1	1
0	1
0	0

Corrective Action:

	Conditions		Actions
	a	-->	**d**
R4:	1		1

Since only light "a" may be switched by the environment, there are two possible external events.

External Events

1. Set a = 1
2. Set a = 0

The stable[31] state space of this system is shown below. Each state is labelled for future reference.

Stable States

	State Variable			
State Label	a	b	c	d
A	0	0	1	0
B	0	0	1	1
C	1	1	0	1
D	1	1	1	1

Response paths are generated by first applying each event to each stable state, thus obtaining a state which might be unstable. Then the sublaws are used to try to bring the system to a final stable state. For example, a possible response path corresponding to the application of the event "set a = 1" to the first stable state A is as follows:

	a	b	c	d	Label
Initial stable state	0	0	1	0	A
Unstable state after event1	1	0	1	0	E
Unstable state after rule R1	1	1	1	0	F
Stable state after rule R4	1	1	1	1	D

There may be more than one possible response path associated with each unstable state. In the above example, rule R4 could have been activated before rule R1. The precise ordering of activation of sublaws is not important so long as each response path ends in the same stable state (i.e. the sublaws are consistent).

---

[31] For the remainder of this thesis, the terms "stable", "complete" or "consistent" shall mean locally stable, locally complete or locally consistent with respect to the defined sublaws.

Response paths are described using the following notation.

$$[(\text{initial}=\text{State}_0),|\text{Event},\text{State}_1>,|\text{Rule}_1,\text{State}_2>,|\text{Rule}_2,\text{State}_3>,\ldots,|\text{Rule}_n,\text{State}_{n+1}>]$$

where State_0 is the initial state to which event Event, is applied, Rule_k is the name of the corrective action rule which moves the system from State_k to State_{k+1}, and State_{n+1} is the final (and therefore stable) state of the system. If the above analysis is repeated for the remaining stable states and events, the following unstable states and response paths may be generated.

Unstable States

E	1	0	1	0
F	1	1	1	0
G	1	0	1	1
H	0	1	0	1
I	0	0	0	1
J	0	1	1	1

Response Paths

Path #	Event	Response Path
1	a = 1	[(initial=A),|Event 1,E>,|R1,F>,|R4,D>]
2	a = 1	[(initial=B),|Event 1,G>,|R1,D>]
3	a = 1	[(initial=C),|Event 1,C>]
4	a = 1	[(initial=D),|Event 1,D>]
5	a = 0	[(initial=A),|Event 2,A>]
6	a = 0	[(initial=B),|Event 2,B>]
7	a = 0	[(initial=C),|Event 2,H>,|R2,I>,|R3,B>]
8	a = 0	[(initial=D),|Event 2,J>,|R2,B>]

Every state entered as a result of an external event is transformed into a stable state. This means the sublaws are complete. The sublaws are consistent because all alternative response paths lead to the same stable states.

2.3. Implementation: The Specifications Analysis Tools

A set of Prolog-based[32] specifications analysis tools has been created to facilitate the construction of a system model based on the system theory concepts presented in the previous section. These tools provide the following functions:

a. Testing for:
   1) syntactic errors in the system model (e.g. misplaced punctuation, inconsistent naming, etc.),
   2) stable condition coverage of the state variables (i.e. each state variable is referenced in at least one rule from the stability conditions of a sublaw),
   3) state variable variance (i.e. each state variable is assigned all of its defined values),
   4) conflicting sublaws, and
   5) local completeness and local consistency of the sublaws.

b. Determination of the stable state space of the system.

c. Determination of the unstable state space and response paths of the system.

d. Suggestion of possible decompositions.

The modelling syntax required by the specifications analysis tools, and the various tests which can be applied to the model, will be described in the context of the four-lights example. The tests are further described in Appendix D. The procedures used to find the stable and unstable state spaces as well as the response paths of a system are described in Appendix E. System decomposition is described in the next chapter.

2.3.1. Entering a System Model

The user is required to create a text file listing all of the state variables, state variable values, sublaws, and external events to be included

---

[32] The tools were implemented using Turbo Prolog (Borland, 1986) running on an IBM AT compatible microcomputer.

in the model. The four-lights example described earlier would be entered as described in the following subsections. Text enclosed by /*...*/ is added for explanation only and is ignored by the tools.

2.3.1.1. State Variables

A simple one-place predicate is used to inform the tools that certain variables are to be included in the model. All state variables must be declared in this way. No additional state variables may be included in any further description of the model (e.g. in the sublaws or events). Predicates declaring the state variables used to describe the four-lights example would be created as follows:

```
/* state variables */
state_variable(a).
state_variable(b).
state_variable(c).
state_variable(d).
```

2.3.1.2. Values

Each state variable may be assigned only a limited number of values[33]. All possible values must be declared using the binary predicate "values(StateVariableName,Values)", where StateVariableName is the name of the state variable and Values is a list of possible values. If a state variable, which was not declared using the "state_variable()" predicate, is used here an error message will be generated by the specifications analysis tools. Another error message will be produced if a state variable does not assume one of its defined values during the determination of the system's response paths. This last test ensures that the state variable value declarations are consistent with the defined dynamics of the system and is referred to as testing "state variable variance". Any mismatch would indicate either insufficiently defined dynamics (in the form of sublaws) or too many defined values.

---

[33] The problem of state variables which may be assigned a large number (perhaps infinite) of different values is addressed later in this chapter.

```
/* state variable values */
values(a,[0,1]).
values(b,[0,1]).
values(c,[0,1]).
values(d,[0,1]).
```

2.3.1.3. Sublaws

The two components of sublaws (namely, stability conditions and corrective actions) are defined separately. Stability conditions describe the allowed combinations of state variable values in stable system states, and are used to determine the stable state space of the system. Corrective actions specify actions to be taken if the system is not in a stable state, and are used to find all response paths of the system. There is some duplication of information in the two parts of a sublaw. The stable state space of the system could be determined by generating all possible combinations of state variable values and testing to see whether some corrective action rule could alter each of the possible states. If there is no corrective action rule which could alter a state, that state would be added to the stable state space of the system. Such a "generate and test" algorithm becomes computationally intractable as the number of combinations of state variable values increases. The use of stability conditions as formulated above allows a much more efficient method of determining the stable state space of the system. Also, if there were a large number of possible system states, it would be easy for the analyst to accidentally omit a corrective action rule required to "correct" an unstable state. Should such an error occur, the unstable state would be incorrectly assumed to be stable. When separate definitions of both stability conditions and corrective actions are required, tests for local consistency and completeness can point to accidentally omitted rules[34].

Explicit statement of stability conditions also allows the system to be described by a smaller number of rules. There may be some possible states which the system should never enter. In a natural system, where the system law reflects fundamental properties of the physical universe, such states could be

---

[34] The tests for local completeness and consistency provide a kind of cross-check between the stability conditions and corrective actions of the sublaws as well as the defined external events.

impossible (e.g. it is impossible for a mass on the surface of a planet to be falling up). However, in a man-made system where the system law may be imperfectly enforced, it is possible for the system to enter an unexpected state. For example, in a simple accounting system the value of the assets represented by some account may not equal the balance of that account, if some user manually altered the balance. Most accounting systems would have controls in place to prevent such alterations. However, if this sort of tampering was not included among the defined external events for the system, the model might not include any sublaws to deal with the situation. It would be extremely difficult, if not impossible, to anticipate all such undesirable events. When dealing with man-made systems, a model can only approximate the operation of the original system since the external events considered comprise only a subset of all possible external events.

2.3.1.3.1. Stability Conditions

Stability conditions are represented using the binary predicate "static(LawName,Conditions)", where LawName is some arbitrary name for the law and Conditions is a list of state variable name and value pairs all of which must occur together in each stable state of the system. A stable condition may consist of more than one rule. This is modelled using several clauses with the same LawName parameter. A stable state need satisfy only one of the rules forming the stability conditions of a particular sublaw[35]. That is, "static()" clauses with the same name are combined using an inclusive OR condition. Clauses with different LawNames are combined using an AND condition. If a state variable name or value which was not declared with a "state_variable()" or "value()" predicate is used to define a stable condition rule, the specifications analysis tools will issue an error message. Other tests of the stability conditions are described in Appendix D. These ensure that every defined state variable is mentioned in at least one stable condition rule (referred to as "stable condition coverage"), and that the stable condition rules do not conflict with each other. The stability conditions for the four-lights example are defined as follows:

---

[35] The relationship between stable states and stability conditions is described in more detail in Appendix E.

```
/* stability conditions */
static("S1",[v(a,"0"),v(b,"0")]).
static("S1",[v(a,"1"),v(b,"1")]).

static("S2",[v(b,"0"),v(c,"1")]).
static("S2",[v(b,"1"),v(c,"0")]).
static("S2",[v(b,"1"),v(c,"1")]).

static("S3",[v(b,"1"),v(d,"1")]).
static("S3",[v(b,"0"),v(d,"0")]).
static("S3",[v(b,"0"),v(d,"1")]).
```

2.3.1.3.2. Corrective Actions

Corrective actions are represented using the ternary predicate "dynamic(LawName,Conditions,Actions)". Again, LawName is some arbitrary name. Conditions is a list of activation conditions, consisting of state variable name and value pairs, which must be satisfied by an unstable state before the corrective action rule is allowed to affect the system state (i.e. to "fire"). Actions is a list of state variable name and value pairs which specify the final values of certain state variables after the sublaw is allowed to "fire". For example, the corrective action rule

```
dynamic("D3",[v(a,"1")],[v(d,"1")]).
```

means that if the value of "a" is "1" then "d" should be set to "1". If a state variable is not mentioned in the list of final values, it is assumed to have the same state as it had before the sublaw was fired. All corrective action rules, whether they have the same name or not, are combined using an OR condition. If a state variable name or value which was not declared with a "state_variable()" or "value()" predicate is used to define a corrective action rule, the specifications analysis tools will issue an error message. The corrective actions for the four-lights example are defined as follows:

```
/* corrective actions */
dynamic("D1",[v(a,"0")],[v(b,"0")]).
dynamic("D1",[v(a,"1")],[v(b,"1")]).

dynamic("D2",[v(a,"0")],[v(c,"1")]).

dynamic("D3",[v(a,"1")],[v(d,"1")]).
```

2.3.1.4. External Events

External events are defined using the binary predicate "event(EventName,Actions)", where EventName is some arbitrary name for the event, and Actions is a list of the state variables altered by the external event together with their altered values. As in the case of sublaws, if a state variable name or value which was not declared with a "state_variable()" or "value()" predicate is used to define an external event, the specifications analysis tools will issue an error message. The external events affecting the four-lights example are defined as follows:

```
/* External Events */
event("E1",[v(a,"0")]).
event("E2",[v(a,"1")]).
```

The various tests performed by the specifications analysis tools are summarized in Table I.

2.3.2. A More Extended Example

The above example was simplified by the fact that each state variable had a small number of discrete values. What happens if there exists a state variable with a very large or even infinite number of possible values? Complete explication of sublaws in the manner described above would be impossible. To model systems described by such state variables it is necessary to reduce the level of detail of the sublaws. The statics and dynamics must be described qualitatively where each state variable may take on only a small number of values. These state variable values are called SUBREGIONS in keeping with the work of De Kleer and Brown (1985). Subregions are bounded by critical values of the real-world state variable. For example, in an inventory management

Table I: Tests performed by the specifications analysis tools.

Diagnostic Type	When Identified	How Identified	Possible Meaning
syntax	when model is loaded into Prolog	illegal Prolog syntax	typing errors
naming consistency	before generation of stable state space	name or value in sublaw or event does not match declarations	1. spelling error 2. insufficient defined state variables 3. insufficient defined values
static sublaw coverage	before generation of stable state space	a state variable is not referenced in a static sublaw	1. missing static sublaw 2. too many defined state variables
static sublaw conflict	before generation of stable state space	see Appendix D	inconsistent specification of static sublaws
state variable variation	after generation of response paths	a state variable does not obtain all of its defined values	1. too many defined values 2. missing external events
local completeness	during generation of response paths	dynamic sublaws cannot move system to a stable state after application of some event	1. too many defined external events 2. missing or incorrect dynamic sublaws 3. missing or incorrect static sublaws
local consistency	during generation of response paths	dynamic sublaws can move the system to more than one stable state following the application of some event	improperly defined dynamic sublaws

system, knowledge of the exact quantity on hand of a particular item is probably not important in order to describe the operation of the system. It is likely that certain actions will be taken if the quantity is either above or below a certain critical value, say the economic order quantity. In this case the state variable "quantity_on_hand" might be modelled as having two discrete values: "under_eoq" and "over_eoq". Use of state variables with values that are actually subregions is illustrated in the following example adapted from Wand and Weber (1989).

Consider a payroll system for a company[36]. The company has two types of jobs: office and sales. An employee may be in either a regular or in a managerial position. Salaries are comprised of base pay, overtime pay and commissions. The way in which total salary is calculated depends on the job type and employee position. Company policy is as follows:

- the office staff is entitled to overtime pay but not to commissions.
- the sales staff is entitled to commissions but not to overtime pay.
- managers are not entitled to overtime pay nor commissions.
- hours and sales are recorded for all employees. (This might happen if managers are required to report hours and office workers may take a telephone order.)
- all employees receive benefits.

Also assume that all payroll processing takes place at the end of some period.

This system would be entered into the specifications analysis tools as shown in Appendix F. The only external events modelled affect the state variable "end". Its value changes from "0" to "1" at the end of the period and from "1" to "0" at the start of the next period. Most continuous state variables are represented using two subregions. For example, the "sales" state variable may have values of either zero or positive ("0" or "nz" in the model). The state variable for "hours worked" is somewhat more complicated. An employee may work sufficient hours to qualify for overtime pay and base pay, a lesser number of hours for

---

[36] This system will be later referred to as the "initial" payroll system to distinguish it from a similar system to be referred to as the "modified" payroll system. These systems are fairly simple. There is no intention to suggest that real payroll systems can be as easily modelled as these examples. A more complicated "real" system will be examined in Chapter 6.

which he will only receive base pay, or no hours at all. In the model, the state variable representing "hours worked" may take on any of three values.

hours = ot - sufficient hours to qualify for overtime pay and base pay.
hours = reg - employee to receive base pay only.
hours = "0" - no hours worked.

This system model has ninety-six stable states. Forty-eight of these states represent the initial states of the system when the state variable "end" has a value of "0". In these states all quantities to be calculated at the end of the period have a value of "0". The other forty-eight states represent the final system states when "end" has a value of "1" and base pay, overtime, benefits, commissions and total pay have been calculated.

Alternatively, the payroll system could be modelled without use of the state variable "end" as shown in Appendix G. It is a somewhat more abstract representation, in that the concept of end-of-period processing has been eliminated. In a sense, the above model is "batch" and this model is "interactive"[37]. The model describes the allowable configurations (i.e. stable states) of state variable values after all processing has been completed. External events then become those occurrences which can alter these stable configurations, as opposed to the massive transition represented by end-of-period processing. The five events defined for this system occur when an employee is reported to have worked a number of hours or made some sales. Also notice that the benefits state variable has only one possible value: non-zero. This is because its value does not depend on the value of any other state variable in the new model. As shall be shown in the next chapter, the decompositions generated automatically by the specifications analysis tools are similar, but not the same, for the model of the batch and interactive systems. In particular, the benefits state variable does not appear in any decomposition of the interactive system. The reason for this will be discussed in the next chapter.

---

[37] Pick (1986) defines "batch and "interactive" as follows. "Batch" describes systems where a number of similar input items are grouped together for processing during the same machine run (p. 622). In the batch example, the machine run occurs at the end of the period. "Interactive" describes systems where the user has rapid two-way communication with a computer (p. 670). In the interactive example, calculated values are updated without an end of period external event.

This new model has forty-eight stable states. These states are identical to the stable states of the "batch" system when the state variable "end" has the value "1" except that "end" is not included. It should be noted that although this representation has fewer stable states, it is not necessarily more efficient than the "batch" representation. There were only two events defined for the first model, whereas this model has five. This means that 192 (96 times 2) response paths had to be determined for the "batch" model, and 240 (48 times 5) had to be found for the "interactive" model.

2.4. Conclusions

A formalism for the representation of systems has been developed. SELMA is notable for its focus on laws rather than on procedures. Consistent representation of the linkages between the properties of the system, in the form of sublaws, facilitates tests of both completeness and consistency of the system description. Sublaws are seen as a practical way to formulate a global system law. The analyst may focus his or her attention on small parts of the system, and still ensure that the sublaws form a complete and consistent model of the system.

A basic implementation of a set of Prolog tools to support SELMA has been described. While its use has been shown to be feasible for some small problems, further testing is required. A larger "real" system needs to be considered. It is possible that, even with the use of state variable subregions, as the number of state variables increases, there could be an unacceptably rapid increase in the number of stable system states. However, it should be noted that a relationship of the form:

number of system states = number of values for state variable 1 *
number of values for state variable 2 *
...
number of values for state variable n

could only occur if each state variable were "independent" of every other state variable. In other words, a very large number of system states is only expected if for every possible value of each state variable every other state variable could have each of its possible values. This would mean that the size of the

stable state space of the system equals the size of its possible state space. This sort of behaviour is not expected for most interesting systems, as it implies no coupling among the state variables. For example, the simple four-lights example has a possible state space with 16 (= 2^4) states, but there are only four stable states. Also, the "batch" payroll system has a possible state space with 3072 (= $2^{10}*3$) states, but there are only 96 stable states.

As currently implemented, the specifications analysis tools have a very limited syntax. In some cases, coding of sublaws could be made more efficient if state variables could be described as <u>not</u> having a certain value, rather than specifying all the values it may have. Also, many systems are likely to require qualitative addition and multiplication (e.g. the payroll system example). The formats of the sublaws which represent these operations are well defined. De Kleer et al. (1985) define qualitative addition and multiplication as follows:

Addition

X	-	0	+
Y			
-	-	-	?
0	-	0	+
+	?	+	+

Multiplication

X	-	0	+
Y			
-	+	0	-
0	0	0	0
+	-	0	+

Ambiguities may arise when adding quantities of different sign. However, they can probably be avoided through careful definition of state variable values. Avoiding ambiguities then becomes the responsibility of the analyst and not the specifications analysis tools. The specifications analysis tools could be enhanced to support simple rendering of addition and multiplication operations. However, a limited syntax is sufficient to illustrate the feasibility of SELMA.

Chapter 3: A Theory of Decomposition

The problem of identifying the subsystems from which a system is composed is not trivial. This chapter begins with an intuitive example illustrating the difficulty of decomposition. This is followed by formal definitions of several concepts related to system decomposition. A number of heuristics and theorems, used to limit the search space of possible decompositions, are also presented. Finally, a decomposition algorithm compatible with SELMA is described and its use is demonstrated on several simple systems.

3.1. General

Bunge (1979, p. 11) describes system decomposition on the basis of identifiable things. However, only by observing a system's behaviour can a designer hope to discover into what parts the system may be decomposed. The behaviours of the properties[38] describing a system, and not the things from which it is constructed, are of primary importance to decomposition (Simon and Ando, 1961). Consider a simplified bicycle system. Many people would recognize the following things as being parts of a bicycle.

Things

front wheel	rear wheel	pedals	frame
front forks	handle bars	chain	

Some state variables representing the properties of the bicycle are listed below. Notice, that only normal operation of a bicycle is being modelled. That is, we are not concerned with skidding, falling over, etcetera.

[38] It could be argued that only things can exhibit behaviour. However SELMA does not explicitly model things. For the purposes of this research, it is sufficient to describe a system's behaviour by the describing the observed changes of the properties.

State Variables

front wheel angle[39]	front fork angle
front wheel rotational speed	handle bar angle
rear wheel rotational speed	frame speed
pedal rotational speed	chain rotational speed

A reasonable decomposition on the basis of things might include the following subsystems:

front end:
- front wheel
- front forks
- handle bars

rear end:
- rear wheel
- pedals
- chain

It is not clear with which subsystem the frame should be associated as it spans both the front and rear ends. Also notice that the behaviour of the front wheel will be related to the behaviour of the rear wheel. Under normal operating conditions the rotational speed of the two wheels will be the same[40]. This dependency implies that the two subsystems are coupled. In general, coupling between two subsystems exists when the behaviours of the two subsystems are not independent. In this case, coupling can be avoided if the subsystems are selected on the basis of steering and forward motion state variables as shown:

steering subsystem:
- front wheel angle
- front fork angle
- handle bar angle

forward motion subsystem:
- front wheel rotational speed
- rear wheel rotational speed
- pedal rotational speed
- chain rotational speed
- frame speed

---

[39] Front wheel angle, front fork angle, and handle bar angle are all horizontal angles measured relative to the frame of the bicycle.

[40] Assuming front and rear wheels of the same radius.

Insofar as these last two subsystems can been given meaningful names, they do represent things. However, it is argued that the things represented are not intuitively obvious. Many analysts would not consider "splitting" a physical object (e.g. the front wheel) between two subsystems. The only "behaviour" suggesting the first decomposition occurs during bicycle assembly. Assembly contexts are far too tempting a criteria for decomposition. An analyst needs to consider the behaviour of a system in all contexts of interest. In SELMA, different contexts are represented by different external events.

Analysts who consider only decompositions consisting of obvious things may miss "superior" alternative decompositions. It may happen that the "good" subsystems, identified by the decomposition methodology presented here, will have state variables corresponding to the properties of an intuitively obvious thing, but this is by no means certain.

3.2. The Decomposition Formalism

The meaning of decomposition will be formally defined in this section, but first some terms for describing system dynamics must be introduced.

Definition: Subsystem

Any subset X of the state variables describing a system σ will describe a SUBSYSTEM of σ. For convenience, X may be referred to as a subsystem[41].

Not all subsets of state variables will describe reasonable subsystems. For the bicycle example, one possible unreasonable subsystem would be described by "front wheel angle" and "pedal rotational speed". The development of criteria for selecting reasonable subsystems is the major purpose of this chapter.

---

[41] A subsystem consists of more than just a set of descriptive state variables. There must also be rules for governing subsystem behaviour. However; for purposes of system decomposition, it is sufficient to identify a subsystem by a set of state variables.

Definition: Projection of a Subsystem

The state x of a subsystem X of a system σ when σ is in state s is called the PROJECTION of s onto X, x = proj(s,X).

For example, consider the following state of the bicycle system.

State Variable	Value
front wheel angle	turning left
front wheel rotational speed	positive
rear wheel rotational speed	positive
front fork angle	turning left
handle bar angle	turning left
pedal rotational speed	zero
chain rotational speed	zero
frame speed	positive

That is, the bicycle is coasting around a left turn. The projection of this state onto the previously identified steering subsystem is

State Variable	Value
front wheel angle	turning left
front fork angle	turning left
handle bar angle	turning left

It should be noted that there may be many system states having the same projection. The state of the steering subsystem would be the same if the bicycle was pedalled (as opposed to coasted) around a left corner.

Definition: Deterministic Subsystem

Wand and Weber (1988) hypothesize that all good decompositions will satisfy the following requirement:

The behaviour of each subsystem is determined only by those state variables describing the subsystem.

This means a decomposition is good if each subsystem behaves deterministically. A subsystem behaves deterministically if its final state is functionally determined by its initial state, or if for every initial state of the subsystem there is only one possible final state of the subsystem. This implies that all the information necessary to determine the final state of the subsystem is already contained in the subsystem. It is not necessary to consider the states of other subsystems in order to decide how the subsystem will behave. If the state of a subsystem depends on the state of another, the subsystems are coupled. Therefore, this requirement will ensure that there is no coupling between the subsystems of a good decomposition. Wand and Weber's requirement may be formally expressed as follows:

> Let σ be a system with system law L, and let R be a set of states of σ. A subsystem X of σ is deterministic with respect to R and L, if and only if all system states s in R, having the same initial subsystem state proj(s,X), have the same final subsystem state proj(L(s),X). That is:

X is deterministic with respect to R and L if and only if

FOR ALL s_1, s_2

such that $s_1 \in R$ and $s_2 \in R$, and

such that $proj(s_1,X) = proj(s_2,X)$,

$proj(L(s_1),X) = proj(L(s_2),X)$

A subsystem is characterized by a set of descriptive state variables. The behaviour of a deterministic subsystem can be defined by a function involving only these state variables. This function may be expressed by a sublaw after considering the subsystem state changes between initial and final states. For example, consider a system described by binary state variables {x,y,z}. Assume that the system dynamics are defined by the following unstable state space and corresponding final stable states.

Unstable States				Corresponding Final Stable States		
x	**y**	**z**	---->	**x**	**y**	**z**
0	1	0		1	1	1
0	1	1		1	1	1
1	1	1		1	0	0
1	1	0		1	0	0

We see that {y,z} is not a deterministic subsystem since the subsystem state {1,1} corresponds to final subsystem states {1,1} and {0,0}. However, {x,y} is a deterministic subsystem in that no initial subsystem state corresponds to more than one final state. The state transitions for the subsystem {x,y} are as shown:

x	**y**	---->	**x**	**y**
0	1		1	1
1	1		1	0

The corrective actions of a sublaw describing this behaviour could be expressed as follows:

Corrective Actions:

Conditions			Actions	
X	**Y**	-->	**X**	**Y**
0	1		1	1
1	1		1	0

Definition: INTERNAL EVENT

External events alter the values of some of the state variables describing a system. The system responds to the external event by further altering the values of its state variables until it enters a stable state. This further alteration of state variables is accomplished through INTERNAL EVENTS.

The action of a sublaw, as described in Chapter 2, corresponds to an internal event. The actions of external and internal events both result in

system state changes. The sequence of state variable value changes as the system moves towards a stable state constitutes a response path. Since there may exist many system response paths leading to the same stable state, an external event need not be followed by a unique sequence of internal events.

The change of state from an unstable to a stable state can be viewed as a sequence of internal events. For example, consider the bicycle system at rest (i.e. "frame speed" = zero). If the pedals are made to rotate, the chain, rear wheel and front wheel must also begin to rotate. However, the bicycle might be modelled such that the chain and rear wheel begin to rotate before the front wheel and frame begin to move (e.g. some "play" in the free wheel mechanism). In this case two internal events would follow the external event as shown:

external event: rotate pedals
internal event 1: rotate chain and rear wheel
internal event 2: rotate front wheel and move frame

The bicycle system can be viewed as entering a number of unstable states after the action of an external event before once again achieving a stable state. One possible series of unstable states leading to a stable state is shown below. Other sequences of internal events are possible, but if the system is complete and consistent, all such sequences will lead to the same stable state. Changes to system states are indicated with a "*".

Initial Stable State: stopped, with front wheel pointing straight ahead

State Variable	Value
front wheel angle	straight
front wheel rotational speed	zero
rear wheel rotational speed	zero
front fork angle	straight
handle bar angle	straight
pedal rotational speed	zero
chain rotational speed	zero
frame speed	zero

External Event: start peddling

First Unstable State:

State Variable	Value	
front wheel angle	straight	
front wheel rotational speed	zero	
rear wheel rotational speed	zero	
front fork angle	straight	
handle bar angle	straight	
pedal rotational speed	positive	*
chain rotational speed	zero	
frame speed	zero	

First Internal Event: set values of chain and rear wheel rotational speed

Second Unstable State:

State Variable	Value	
front wheel angle	straight	
front wheel rotational speed	zero	
rear wheel rotational speed	positive	*
front fork angle	straight	
handle bar angle	straight	
pedal rotational speed	positive	
chain rotational speed	positive	*
frame speed	zero	

Second Internal Event: set values of front wheel rotational speed and frame speed

Final Stable State: moving straight ahead

State Variable	Value	
front wheel angle	straight	
front wheel rotational speed	positive	*
rear wheel rotational speed	positive	
front fork angle	straight	
handle bar angle	straight	
pedal rotational speed	positive	
chain rotational speed	positive	

frame speed positive *

The external event altered the value of "pedal rotational speed". The first internal event changed the values of state variables "chain rotational speed" and "rear wheel rotational speed" based on the value of "pedal rotational speed". The second internal event updated the values of state variables "frame speed" and "front wheel rotational speed". The final values of any of the previously altered state variables could have been used as the basis for this second change. For the sake of argument, assume that the final value of "rear wheel rotational speed" was used. The process of altering the values of state variables through internal events shall be called an UPDATE. Updates involve sets of state variables, or subsystems. In the above example, the subsystem {pedal rotational speed, chain rotational speed, rear wheel rotational speed} was used to update the first unstable state to the second unstable state. Then the subsystem {rear wheel rotational speed, frame speed, front wheel rotational speed} was used to update the second unstable state to the final stable state. The notion of system updates can be formalized as follows:

Definition: Updating

Let σ be a system with system law L, R be a set of states of σ, and U be the state variables used to describe a set of subsystems of σ[42]. A set of states R′ is UPDATED with respect to U and R by setting the values of those state variables in each system state s in R, which are elements of U equal to their values in the final stable state L(s). That is, if SV is the set of all state variables describing σ, then

R′ = {s′} such that
THERE EXISTS s such that s ∈ R and
FOR ALL v such that v ∈ SV,
(proj(s′,{v}) = proj(L(s),{v}) and v ∈ U)
or
(proj(s′,{v}) = proj(s,{v}) and v ∉ U)

[42] In the previous example, updates involved single subsystems only. In general, a set of subsystems may be used to perform an update.

The "or" separates two possibilities for the value of each state variable in an updated system state. The first possibility occurs when the state variable is found in the set of subsystems used to update the initial system state. In this case, the value of the state variable in the updated system state is equal to its value in the final stable state of the system. The second possibility occurs when the state variable is not used to describe any subsystem used to update the initial system state. In this case, the value of the state variable is left unchanged.

As another example, consider a bicycle beginning to move to the left after the rider begins to pedal. A possible initial unstable state/final stable state pair for this situation is shown below.

State Variable	Initial Unstable Values	Final Stable Values
front wheel angle	straight	turning left
front wheel rotational speed	zero	positive
rear wheel rotational speed	zero	positive
front fork angle	straight	turning left
handle bar angle	turning left	turning left
pedal rotational speed	positive	positive
chain rotational speed	positive	positive
frame speed	zero	positive

The initial state is clearly unstable as the pedals are turning but the wheels are not yet spinning. The state could be updated with respect to the forward motion subsystem, identified earlier, by setting "front wheel rotational speed" and "rear wheel rotational speed" to "positive". However, the resulting updated system state would still not be stable, since the handle bars and the wheels are not pointing in the same direction. If the system were further updated with respect to the steering subsystem, the resulting system state would be stable.

Updating refers to altering a set of states to reflect the completion of some activities within the system. As shall be shown, it is the update which reflects sequential decomposition. A few more definitions will make it easier to discuss updates as they pertain to system decomposition.

Definition: First Intermediate State Space and System Relation

The set of all system states which result from the action of any external event in the set of external events E on a stable state of the system σ, is called the FIRST INTERMEDIATE STATE SPACE (First ISS) of σ with respect to E. E will always include the NULL EVENT. The null event does not change the value of any state variable. The first ISS and the final stable system states associated with each member state comprise the FIRST SYSTEM RELATION of σ with respect to E.

These concepts were used in Chapter 2. The first intermediate state space is the set of states for which response paths leading to unique stable states must be found, if the system model is to be complete and consistent. The initial unstable states and the associated stable states comprise the first system relation. The first system relation shows to which stable state the system will move should it be in an unstable state as the direct result of the action by an external event.

Definition: Nth Intermediate State Space and System Relation

The set of all system states, where s results from a given set of N updates being applied to each state of the first intermediate state space of σ, is called an Nth INTERMEDIATE STATE SPACE (Nth ISS). The Nth ISS and the final stable system states associated with each member state comprise the Nth SYSTEM RELATION.

Definition: Level

The set of subsystems used to update an intermediate state space will be called a LEVEL.

Decomposition, itself, may now be formally defined.

Definition: Decomposition

If a series of updates is begun with the first ISS of σ with respect to external events E, and ends when the updated ISS contains only stable states, the resulting sequence of levels is called a DECOMPOSITION of system σ with respect to external events E. If only deterministic subsystems (as defined above) are used to perform the updates, the resulting sequence of levels is called a DETERMINISTIC DECOMPOSITION.

Unfortunately, there will be in general, a very large number of deterministic subsystems with respect to any ISS of a system. For example, any subset of state variables whose values do not change between states in the ISS and the corresponding final stable states will describe deterministic subsystems. Consider the following first system relation for a system described by four binary state variables {a,b,c,d}:

First Intermediate State Space					Corresponding Final Stable States			
a	**b**	**c**	**d**	-->	**a**	**b**	**c**	**d**
0	0	0	0		0	0	1	1
0	1	0	0		0	1	0	0
1	0	0	0		1	0	0	0
1	1	0	0		1	1	1	1

The subsystems {a}, {b}, {a,b}, {a,b,c}, {a,b,d}, and {a,b,c,d} are all deterministic. Any subset of these deterministic subsystems may be used to update the first ISS. Any ISS thus created may be further updated using any subsystem that is deterministic with respect to the new space. This process will lead to at least $2^6! \approx 1.3X10^{89}$ deterministic decompositions[43]. Most of these deterministic decompositions will be of no interest to the analyst. Several rules for avoiding these "useless" decompositions will be discussed following the next section of this chapter.

---

[43] If n is the number of good subsystems, there are 2^n ways to select a subset of the good subsystems. Each permutation of these subsets will correspond to a good decomposition. Therefore, there are at least $2^n!$ good decompositions for a system with n good subsystems. There may be even more good decompositions if subsystems, which are not good with respect to the first intermediate state space, become good as a result of an update operation.

3.3. Decomposition Syntax

In this and later chapters it will be necessary to discuss, and even compare, many decompositions. A consistent representation scheme is required. Two such schemes will be defined in this section. The first conveys the most information, but is somewhat difficult to interpret without practice. The second is diagrammatic and emphasizes the linkages between subsystems. Both will be used as appropriate.

Consider the system described by binary state variables {a,b,c,d} with a system relation as shown above. If the first ISS is updated using {a,b,c} and {b}, the new or second ISS contains the following states.

First ISS		Second ISS		Corresponding Final Stable States
a b c d	-->	a b c d	-->	a b c d
0 0 0 0		0 0 1 0		0 0 1 1
0 1 0 0		0 1 0 0		0 1 0 0
1 0 0 0		1 0 0 0		1 0 0 0
1 1 0 0		1 1 1 0		1 1 1 1

The subsystems {a}, {b}, {c}, {a,b}, {a,c}, {b,c}, {a,b,d}, and {c,d} are all deterministic with respect to this second ISS. If {c,d} is selected for an update, the third ISS becomes

Second ISS		Third ISS		Corresponding Final Stable States
a b c d	-->	a b c d	-->	a b c d
0 0 1 0		0 0 1 1		0 0 1 1
0 1 0 0		0 1 0 0		0 1 0 0
1 0 0 0		1 0 0 0		1 0 0 0
1 1 1 0		1 1 1 1		1 1 1 1

The states of the third ISS are the same as the corresponding final stable states. Two levels have been defined: {{a,b,c},{b}} and {{c,d}}. Together they form a deterministic decomposition of the system. The decomposition may be represented as shown below. State variables with values that change during an update are underlined[44].

[44] These state variables will be later defined as OUTPUT state variables.

2: {c,d}

1: {a,b,c} {b}

This decomposition has the following associated semantics.

1. Subsystems {a,b,c} and {b} are deterministic subsystems with respect to the first ISS (or {a,b,c} and {b} are deterministic at level 1).
2. Subsystem {c,d} is a deterministic subsystem with respect to a second ISS (or {c,d} is deterministic at level 2). This state space is formed by updating the first ISS using subsystems {a,b,c} and {b}.
3. The third ISS formed by updating the second ISS using the subsystems {c,d} will contain only stable states.

Decompositions will sometimes be displayed using diagrams similar to Figure 8. Subsystems are represented by boxes containing sets of state variable names. It is easier to see the linkages, or communication, between subsystems in this sort of diagram. Communication (if any) between subsystems is shown by lines between boxes. The lines are labelled with the name of the state variable whose value is passed. Values are passed from lower to higher subsystems only.

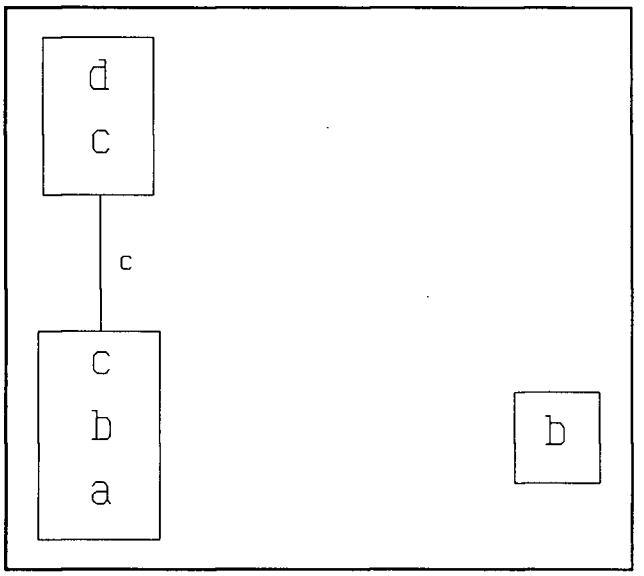


Figure 8: An alternative representation for the parallel/sequential decomposition:

2: {c,d}
1: (a,b,c) {b}

3.4. Limiting the Search Space

3.4.1. General

A number of rules have been found which can considerably limit the number of deterministic decompositions which should be considered by the analyst. Some of these rules are heuristics, in that they cannot be formally proved. Others follow directly from formal definitions and are called theorems. Before the rules may be presented, three more definitions are required. These definitions,

and many of the theorems and heuristics, will be illustrated using the simple system described by state variables {a,b,c,d} as introduced in the previous section.

Definition: Output State Variable

A state variable is an OUTPUT STATE VARIABLE with respect to some intermediate state space R, of a system with law L, if its value in some system state s in R is different from its value in the final stable system state L(s). That is, if v is a state variable then

v is an output state variable with respect to R if and only if
  THERE EXISTS s such that $s \in R$ and
    $proj(s,\{v\}) \neq proj(L(s),\{v\})$

Definition: Input State Variable

The set of state variables whose values are required to predict the final values of the output state variables with respect to some intermediate state space R is called the set of INPUT STATE VARIABLES with respect to R[45].

Definition: Constant State Variable[46]

Any state variable which is not an output state variable with respect to some intermediate state space R is CONSTANT STATE VARIABLE with respect to R.

---

[45] The sets of input and output state variables with respect to some intermediate state space are not necessarily mutually exclusive. The final value of some output state variable could depend on its initial value. Such a state variable would be both an input and an output. A state variable which is both an input and an output will be named twice in the set of state variables describing a subsystem. For example, {x,y,z,z} indicates that values of "x" and "y" and the initial value of "z" are all required to determine the final value of "z".

[46] In many of the examples to be considered in this and later chapters, the set of input state variables will equal the set of constant state variables. The sets only differ when the initial value of an output state variable is required to determine its own final value.

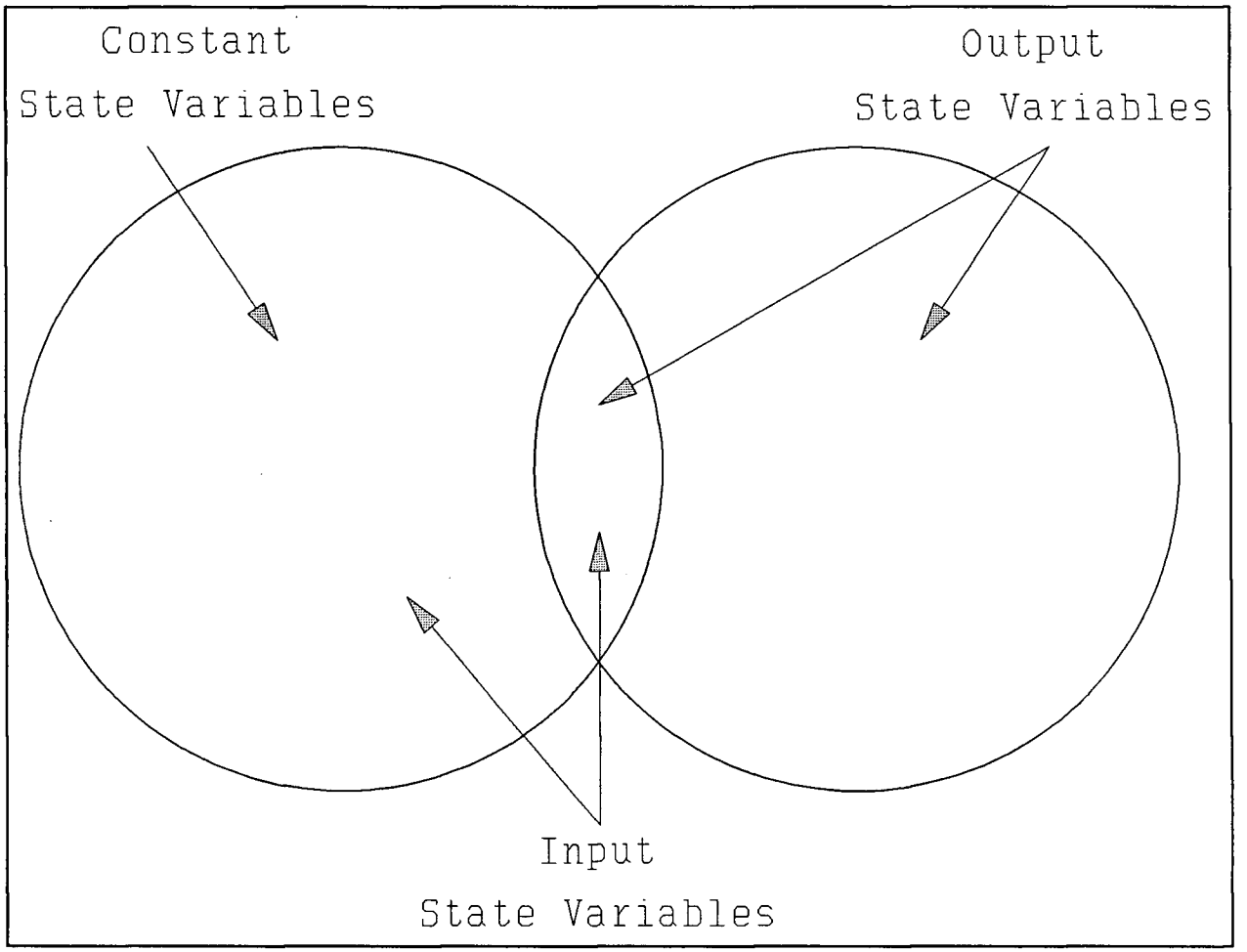


Figure 9: The relationship between output, input, and constant state variables with respect to a given intermediate state space.

The relationships between the set of output, input, and constant state variables with respect to a given intermediate state space are diagrammed in Figure 9.

For example, consider the system described by binary state variables {a,b,c,d}.

First Intermediate State Space					Corresponding Final Stable States			
a	**b**	**c**	**d**	-->	**a**	**b**	**c**	**d**
0	0	0	0		0	0	1	1
0	1	0	0		0	1	0	0
1	0	0	0		1	0	0	0
1	1	0	0		1	1	1	1

State variables {a,b} are input and constant state variables and {c,d} are output state variables with respect to the first ISS. Now consider the second ISS formed by updating the first ISS using the subsystem {a,b,c}.

First ISS		Second ISS		Corresponding Final Stable States
a b c d	-->	a b c d	-->	a b c d
0 0 0 0		0 0 1 0		0 0 1 1
0 1 0 0		0 1 0 0		0 1 0 0
1 0 0 0		1 0 0 0		1 0 0 0
1 1 0 0		1 1 1 0		1 1 1 1

State variables {a,b,c} are all inputs with respect to the second ISS. State variable "c" is no longer an output since its value does not change in any initial/final state pair. Thus output state variables need not remain output state variables after an update.

These definitions of "input" and "output" are not the same as those in common use. It is more usual to refer to external events as inputs and the actions of the system on its environment as outputs.[47] In SELMA, the state variables affected by external events are constant state variables. Their values do not change between states in the first ISS and the corresponding final stable states of the system[48]. They may also be input state variables if they are required to determine the final values of the output state variables. However, there may be other input state variables which are not affected by external events. The definition of output state variable is also somewhat unusual. Outputs are defined for every ISS with the exception of the space containing only stable states. Interaction between output state variables and the environment is not modelled. Any such interactions would form the external events to another system located in the environment of the system under study, and so are not considered.

The rules for limiting the number of deterministic decompositions to be considered by the analyst may now be presented.

---

[47] Bunge (1979, p. 25) defines input and output in this way.

[48] It is assumed that the values of state variables may be set only once during the system's response to an external event. This assumption is discussed, in detail, later in this chapter.

3.4.2. Heuristics and Theorems

3.4.2.1. Subsystems should have outputs

Deterministic subsystems are subsystems whose final states can be predicted knowing only their initial states. By definition, constant state variables do not change their values between the initial and final states of the system. Therefore, it is a trivial exercise to predict the final state of a subsystem described by only constant state variables. The following theorem is suggested by this fact[49].

Theorem 1: Any subsystem X, described only by constant state variable with respect to some intermediate state space R and the corresponding final stable states, will be a deterministic subsystem with respect to R. That is:

IF FOR ALL s such that $s \in R$, proj(s,X) = proj(L(s),X)
THEN X is deterministic with respect to R and L.

Such deterministic subsystems are unlikely to be interesting to an analyst as they contain no information about the dynamics of the system. A program module based on this sort of subsystem would always return the same values it received. This fact leads to the first heuristic for limiting the number of deterministic subsystems which may be used to update an ISS.

Heuristic 1: All deterministic subsystems used to update an intermediate state space must be described by at least one OUTPUT state variable.

For example, Heuristic 1 will ensure the subsystems {a} and {b} are not used to update the first ISS of the system described by binary state variables {a,b,c,d}.

---

[49] Proofs for the theorems included in this document are straightforward and proceed directly from the definitions. They have not been included here.

3.4.2.2. Subsystems should be small

Any subsystem formed by adding a constant state variable to a deterministic subsystem will be deterministic. The value of a constant state variable does not change between initial and final states, and so cannot cause a deterministic subsystem to behave non-deterministically. For example, consider the following second intermediate, and final stable, state spaces of the system described by binary state variables {a,b,c,d}. This second ISS was created by updating the first ISS using the subsystem described by state variables {a,b,c}.

First ISS		Second ISS		Corresponding Final Stable States
a b c d	-->	a b c d	-->	a b c d
0 0 0 0		0 0 1 0		0 0 1 1
0 1 0 0		0 1 0 0		0 1 0 0
1 0 0 0		1 0 0 0		1 0 0 0
1 1 0 0		1 1 1 0		1 1 1 1

State variable "a", "b", and "c" are constant state variables with respect to this ISS, since their values do not change between the intermediate and corresponding final stable states. The subsystem {a,b,d} is deterministic with respect to this ISS, since no initial subsystem state leads to two different final subsystem states. If the state variable "c" is added to {a,b,d}, the resulting set of state variables also describes a deterministic subsystem. This result is expressed by the following theorem.

Theorem 2: Let X be a set of state variables containing output state variables O, and let X′ be a subset of X also containing O. If X′ describes a deterministic subsystem, with respect to some intermediate state space R and system law L, then X will be a deterministic subsystem with respect to R and L. That is:

IF X′ is deterministic with respect to R and L and $X' \subseteq X$ and
  FOR ALL o such that $o \in X$ and
    $proj(s,\{o\}) \neq proj(L(s),\{o\})$ and
    $s \in R$,
    $o \in X'$
THEN X is a deterministic subsystem with respect to R and L.

In the above example, the set of state variables {a,b,c,$\underline{d}$} contained one output state variable. It also contained a subset {a,b,$\underline{d}$} which described a deterministic subsystem. Since this subset also contained the output state variable {d}, the state variables {a,b,c,$\underline{d}$} had to describe a deterministic subsystem. Deterministic subsystems formed by adding constant state variables to existing deterministic subsystems are probably not interesting to an analyst. A program module corresponding to such a subsystem would contain a redundant variable, since the outputs of the subsystem could have been determined by the original variables. This fact suggests the following heuristic.

Heuristic 2: Let X be a set of state variables containing output state variables O, and let X′ be subset of X also containing O. If X′ describes a deterministic subsystem, with respect to some intermediate state space R and system law L, then X may not be used to update R.

This rule ensures that the subsystems used to update an ISS are described by as small a number of input state variables as possible. It is required to avoid trivial decompositions formed by adding constant state variables to deterministic subsystems. For example, without this heuristic both of the following would be considered as possible decompositions of the system described by binary state variables {a,b,c,d}. Output state variables are underlined.

2:	{a,b,$\underline{d}$}	and	2:	{a,b,c,$\underline{d}$}
1:	{a,b,$\underline{c}$}		1:	{a,b,$\underline{c}$}

The second decomposition does not add any information as it could have been deduced from the first decomposition and Theorem 2.

Now recall that the subsystems {a,b,$\underline{c}$} and {a,b,$\underline{d}$} are both deterministic with respect to the first ISS. So is the union of the two subsystems. That is, the subsystem {a,b,c,$\underline{d}$} is also deterministic. This result is generalized in the following theorem.

Theorem 3: A subsystem described by the union of the state variables describing two deterministic subsystems will be deterministic.

Two possible deterministic decompositions of the system are

1: {a,b,c} {a,b,d} and 1: {a,b,c,d}

The second decomposition implies that four state variables are required to predict the final state of the subsystem. The first decomposition contains more information than the second. It tells the analyst that the final values of "c" and "d" may be predicted if the initial values of only "a" and "b" are known. The second decomposition does not indicate whether the values of "a" and "b" are both required to predict the final value of both "c" and "d", or whether just one state variable could serve to predict one of the outputs. Since the second decomposition can be deduced from the first through the use of Theorem 3, the following heuristic is suggested.

Heuristic 3: Do not generate alternative decompositions resulting from the union of smaller subsystems.

Together, Heuristics 2 and 3 ensure that the subsystems presented to the analyst for consideration will be described by as small a number of state variables as possible.

3.4.2.3. Subsystems should show emergence

Consider the example system described by binary state variables {a,b,c,d}. The sets of state variables {a,b,c} and {a,b,d} describe subsystems which are deterministic with respect to the first ISS. An update using these subsystems would lead to an ISS containing only stable states. The first ISS could also be updated using only the subsystem {a,b,c}, to obtain the second ISS shown below.

First ISS		Second ISS		Corresponding Final Stable States
a b c d	-->	**a b c d**	-->	**a b c d**
0 0 0 0		0 0 1 0		0 0 1 1
0 1 0 0		0 1 0 0		0 1 0 0
1 0 0 0		1 0 0 0		1 0 0 0
1 1 0 0		1 1 1 0		1 1 1 1

The subsystem {a,b,d} is still deterministic with respect to this ISS. If this second ISS were further updated using {a,b,d}, the resulting ISS would contain only stable states. Therefore, the following are both deterministic decompositions of the system:

1: {a,b,c} {a,b,d} and 2: {a,b,d}
1: {a,b,c}

The second decomposition does not add to the information provided by the first, and should not have to be considered by the analyst. This observation may be generalized with the following theorem.

Theorem 4: Let X and Y be deterministic subsystems with respect to an intermediate state space R. If X is used to update R to obtain intermediate state space R′ then Y will be deterministic with respect to R′.

This theorem expresses the commutativity of the update. That is, if two subsystems X and Y are deterministic with respect to some ISS, the final ISS's resulting from updating using X and then Y and using Y and then X will be the same.

Theorem 4 suggests another heuristic for limiting the number of deterministic decompositions which have to be considered by the analyst. However, the following definition will make its formulation easier.

Definition: Emergent State Variable

Let x be a output state variable used to describe a subsystem at the nth level of some decomposition. If x is not used to describe any subsystem at any mth level where $m < n$, then x is an EMERGENT STATE VARIABLE at level n.

The concept of an emergent state variable is analogous to the notion of a holistic property. That is, holistic properties are "those characteristics of a particular system that go beyond the qualities of the individual system components" (Mattessich, 1978, p. 31). Holistic properties are a manifestation of the fact that a system is more than the sum of its parts.

For example, consider a payroll system. "Total pay" might be determined by the values of "regular pay" and "overtime". The values of "regular pay" and "overtime" might be determined by the values of "hours worked" and "pay rate". Such a system could be decomposed as shown:

2: {regular pay,overtime,total pay}
1: {hours worked,pay rate,regular pay}
{hours worked,pay rate,overtime}

In this case "total pay" is an emergent state variable at level 2.

Emergent state variables allow the analyst to focus his or her attention on higher-level abstractions of the system under study. The "total pay" emergent state variable could be considered an abstraction of the "hours worked" and "pay rate" state variables. If the analyst were not interested in the degree of detail provided by these state variables, "total pay" may be a perfectly adequate substitute. Decompositions which show the emergence of state variables whenever possible are assumed to be superior to those that do not. The following heuristic is based on this assumption.

Heuristic 4a: All subsystems used to update an nth intermediate state space must by described by at least one state variable which is emergent at level n.

However, this heuristic alone is not enough to avoid redundant alternative decompositions as discussed above. In the decomposition

2: {a,b,d}
1: {a,b,c}

state variable "d" is emergent at level 2. Unfortunately, state variable "d" is not a useful abstraction of any state variables found at lower levels. While state variables "a" and "b" are found at level 1, they are also found at level 2. They are not abstracted out of the view of the system presented to the analyst at any level of the decomposition. Only if the values of emergent state variables are determined by output state variables at a lower level, do they

become useful abstractions for the analyst. The following heuristic embodies this notion.

Heuristic 4b: Any deterministic subsystem used to update the nth intermediate state space must be described by at least one state variable which is an output state variable with respect to the n-1th intermediate state space.

In other words, subsystems used to update an ISS must be described by at least one state variable which was an output state variable with respect to the previous ISS. This ensures that outputs from deterministic subsystems at a lower level will be used as inputs at a higher level whenever possible.

Consider the following decomposition of the example subsystem described by state variable {a,b,c,d}.

2: {c,<u>d</u>}
1: {a,b,<u>c</u>}

State variable "c" is an output state variable at level 1, and it is an input state variable at level 2. State variable "d" is emergent at level 2. Therefore, this decomposition satisfies Heuristics 4a and 4b. Now consider the following decomposition.

2: {a,b,<u>d</u>}
1: {a,b,<u>c</u>}

While state variable "d" is emergent at level 2, no output state variable from level 1 appears at level 2. Therefore, Heuristic 4b would lead to rejection of this decomposition.

When emergent state variables are used to form abstractions of other state variables, some state variables may be "hidden". The concept of a hidden state variable is analogous to "information hiding" as defined by Parnas (1972, p. 1056). Subsystems at higher levels in the decomposition do not have to be "aware" of all state variables considered by lower-level subsystems. For example, consider the payroll system.

2: {regular pay,overtime,total pay}

1: {hours worked,pay rate,regular pay}

{hours worked,pay rate,overtime}

Here, the state variables "hours worked" and "pay rate" are hidden with respect to "total pay". This means that an analyst, interested only in the final value of "total pay", would be concerned with the view of the system shown at the top of Figure 10. The arrows between "regular pay" and "total pay" and between "overtime" and "total pay" indicate value dependencies (e.g. the final value of "total pay" depends on the value of "regular pay"). On the other hand, if the analyst were interested in both "total pay" and "overtime", he or she would require the view shown at the bottom of Figure 10. No state variables are hidden in this view of the system. The formal definition of a hidden state variable is somewhat obscure, but is equivalent to the above "intuitive" description.

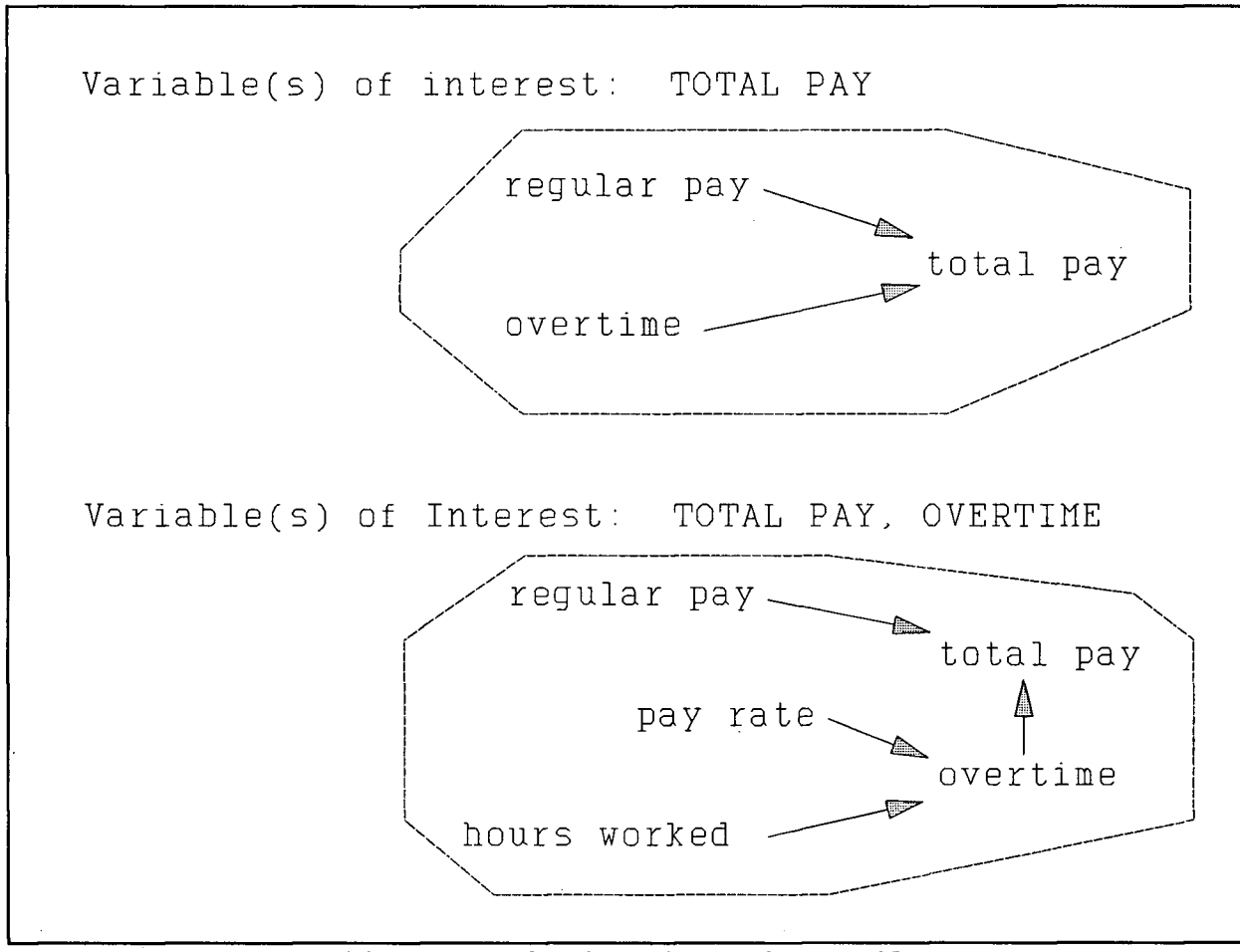


Figure 10: Two possible views of a hypothetical payroll system.

Definition: Hidden State Variable

Let x be an emergent state variable at level n of a decomposition satisfying Heuristics 4a and 4b. If, at level n, the state variable y is not used to describe any subsystem also described by x, then y is HIDDEN with respect to x.

3.4.2.4. Subsystems should not show redundant dependencies

The final value of an output state variable may be functionally determined by more than one subsystem that is deterministic with respect to some ISS. Consider the example system described by binary state variables {a,b,c,d}. The second ISS, formed by updating the first ISS using the subsystem {a,b,c}, is shown below.

First ISS		Second ISS		Corresponding Final Stable States
a b c d	-->	a b c d	-->	a b c d
0 0 0 0		0 0 1 0		0 0 1 1
0 1 0 0		0 1 0 0		0 1 0 0
1 0 0 0		1 0 0 0		1 0 0 0
1 1 0 0		1 1 1 0		1 1 1 1

The final value of output state variable "d" may be functionally determined by either {a,b,d} or {c,d}. However, a decomposition of the form

2: {a,b,d} {c,d}
1: {a,b,c}

would not be considered desirable in that it indicates redundant updating at level 2. There is no need to have "d" set by two subsystems. This observation leads to the following heuristic.

Heuristic 5: The set of deterministic subsystems used to update an intermediate state space may not contain more than one subsystem described by a given output state variable.

This is not meant to imply that an analyst should not be made aware of alternative methods for calculating the final values of output state variables.

The heuristic only forces such alternatives to be shown in different candidate decompositions of the same system. That is,

1: {a,b,c} {a,b,d} and 2: {c,d}
1: {a,b,c}

are possible decompositions of the example system. Both would be suggested by the specifications analysis tools.

3.4.2.5. Bad Subsystems

Theorem 2 specifies a condition under which it is not necessary to scan the ISS in order to see if a subsystem behaves deterministically. The next theorem serves a similar function. Both are used by the specifications analysis tools to speed the search for deterministic subsystems.

Consider the first ISS of the example system.

First Intermediate State Space					Corresponding Final Stable States			
a	b	c	d	-->	a	b	c	d
0	0	0	0		0	0	1	1
0	1	0	0		0	1	0	0
1	0	0	0		1	0	0	0
1	1	0	0		1	1	1	1

The subsystem {b,c,d} is not deterministic with respect to this state space. Neither is the subsystem formed by dropping a constant state variable. That is, the subsystem {c,d} is not deterministic either. This result may be generalized with the following theorem:

Theorem 5: Let a subsystem X, which is not deterministic with respect to some intermediate state space R, be described by the set of output state variables O and constant state variables C. If X′ is another subsystem described entirely by O and a subset of C, then X′ will not be a deterministic subsystem with respect to R.

This means that a subsystem which is not deterministic cannot be made deterministic by dropping some of its constant state variables.

3.4.3. Relationship to the Heuristics of Simon and Ando

The concept of a deterministic subsystem together with the above heuristics are related to the intuitive notions of Simon and Ando (1961) as presented in Chapter 1. Consider their office building example first discussed in Chapter 1. Simon and Ando consider each room to be a subsystem of the building and each office to be a subsystem of a room. They were concerned with describing the thermal equilibrium of the building. Using the decomposition syntax of this chapter, the building system might be characterized by the following decomposition:

3: $\{t_{r1}, \ldots, t_{rN}, \underline{t}_b\}$

2: $\{t_{o11}, \ldots, t_{o1j}, \underline{t}_{r1}\} \ldots \{t_{oN1}, \ldots, t_{oNk}, \underline{t}_{rN}\}$

1: $\{\ldots, \underline{t}_{o11}\} \ldots \{\ldots, \underline{t}_{o1j}\} \ldots \{\ldots, \underline{t}_{oN1}\} \ldots \{\ldots, \underline{t}_{oNk}\}$

where:

t_b = equilibrium temperature of the office building

t_{ri} = equilibrium temperature of the ith room

t_{oij} = equilibrium temperature of the jth office of the ith room

The "t_b" is an emergent state variable at level 3 and each "t_r" is emergent at level 2. Recall that Simon and Ando's necessary criteria for a decomposable system were

a. in a short-term period, as a result of stronger internal bonds, subsystems tend to reach an internal equilibrium "approximately" independently of one another, and

b. in a long-term period, when a whole structure evolves toward a global equilibrium state under the influence of weak interactions among subsystems, the internal equilibriums reached at the end of the short-term period are approximately maintained in relative value.

Suppose a bond is interpreted as a dependency between state variables. Also suppose that bonds are directed.[50] If the subsystems are deterministic, there cannot be bonds between subsystems at the same level[51], but there may be bonds to lower-level subsystems. However, the number of bonds between a subsystem and subsystems at other levels will never be greater than the number of bonds within the subsystem. This can be shown as follows:

1. Heuristic 1 ensures that each subsystem must be described by at least one output state variable.
2. Heuristic 2 ensures that the values of the output state variables are dependent on the values of all input state variables.
3. 1 and 2 imply that if there are n state variables describing the state of a subsystem, there must be at least n-1 bonds between them.
4. There can never be more bonds to other subsystems than there are input state variables (i.e. at most n-1).
5. Therefore, the ratio of the number of internal to external bonds must always be greater than or equal to 1.

If there are stronger links within a subsystem, than between that subsystem and the rest of the system, Simon and Ando argue it is likely to reach equilibrium faster than the whole system. If the number of bonds is assumed to be proportional to the strength of the link, subsystems can never be more strongly linked together that they are internally. This satisfies Simon and Ando's interaction strength requirement. Now suppose that a subsystem is in equilibrium when no descriptive state variable is an output state variable. That is, all state variables have attained their final values. A subsystem will be in equilibrium after it is used to perform an update. By definition, lower-level subsystems will always reach equilibrium before the system as a whole.

---

[50] For example, consider a system where the value of some state variable "b" depends on the value of some state variable "a" and not vice versa. A bond is assumed to exist between the subsystem which determines the value of "a" and the subsystem described by "b", but not the reverse.

[51] A bond between subsystems X and Y at the same level would imply that the value of some state variable in Y is dependent on the value of some other state variable in X. Therefore, Y could not be a good subsystem, since the behaviour of a good subsystem is predictable knowing only the values of its own state variables.

This behaviour is the same as that predicted by Simon and Ando for a decomposable system.

3.5. Automation of Decomposition

3.5.1. An Algorithm for Decomposition

An algorithm[52] employing the notions formally defined in the previous sections has been developed. An implementation of this algorithm comprises a large part of the computerized specifications analysis tools. Operation of the algorithm will be illustrated using a simple system.

The algorithm requires as input an explication of the system law in the form of initial unstable and final stable state pairs. The Decompose() procedure is then called recursively until a set of alternative decompositions has been generated. Each alternative will be a deterministic decomposition, and it will satisfy each of the heuristics described earlier in this chapter.

Required functions:

Outputs(R)

- returns a list of the output state variables with respect to the intermediate state space R.

Subsystems(R,Outputs,PreviousOutputs)

- returns a list of deterministic subsystems with respect to some intermediate state space R. Outputs is a list of output state variables with respect to R. PreviousOutputs is a list of the output state variables describing the subsystems used in the update which produced R. Each deterministic subsystem will be described by a set of state variables such that:

1) As required by Heuristic 1, the set of state variables will contain an element of the list of state variables assigned to Outputs.

---

[52] There is no intention to suggest that the algorithm described here represents the "best" way to operationalize the theory of decomposition. It is possible that more efficient algorithms exist. This particular algorithm is described to show that operationalization is possible. The most important contributions of this research are to be found in the construction and analysis of system models.

2) As required by Heuristics 2 and 3, the set of state variables will be as small as possible.

3) As required by Heuristic 4b, the set of state variables will contain an element of the list of state variables assigned to PreviousOutputs (unless there are no previous outputs, as will be the case with the first ISS).

Theorem 5 is used to further reduce the number of subsystems which must be tested by scanning the ISS.

Subsets(Subsystems,Outputs)

- returns a list of all subsets of the set of subsystems assigned to Subsystems. Outputs is a list of the output state variables used to describe the subsystems assigned to Subsystems. As the subsets will be used to perform updates on some ISS, care must be taken to ensure Heuristic 5 is not violated. That is, no subset may contain two subsystems which are described by the same output state variable.

Update(R,U)

- returns the ISS formed by updating R with respect to the subsystems U.

The body of the algorithm:

```
Begin
    {Set R1 equal to the set of initial unstable states. R1 is the ISS
    formed by applying each defined external event to each stable state of
    the system. The symbol [] refers to a list with no members. The first
    time the Decompose() procedure is called there are no outputs with
    respect to a previous level and no deterministic subsystems have been
    found.}
    Decompose(R1,[],[]);
End.
```

The decomposition procedure:

```
Procedure:  Decompose(R,PreviousOutputs,DecompSoFar)
Arguments:
      R -   an intermediate state space.
      PreviousOutputs - the output state variables of the subsystems used for
                        the update which produced R.
      DecompSoFar -     a list of the sets of subsystems used to obtain R from
                        first ISS via a series of updates.
Begin
      Step 1:     {Find the output state variables with respect to the ISS.}
                  Outputs := Outputs(R);
      Step 2:     {If Outputs is empty all the states in the ISS are stable.
                  This means that the sets of subsystems used to perform updates
                  defines a deterministic decomposition.}
                  If Outputs is empty
                  Begin
                        Output DecompSoFar as a possible decomposition;
                        Exit;
                  End;
      Step 3:     {Find the deterministic subsystems with respect to the ISS
                  subject to certain conditions described for the DetSubsystems()
                  function.}
                  DetSubsystems := Subsystems(R,Outputs,PreviousOutputs);
      Step 4:     {Find all the subsets of the set of deterministic subsystems
                  suitable for updating the ISS.  These subsets must meet certain
                  criteria as described for the Subsets() function.}
                  PossibleUpdates := Subsets(DetSubsystems,Outputs);
      Step 5:     {Perform a depth-first search for deterministic
                  decompositions.  Call the Decompose() procedure recursively
                  for each new ISS formed by updating the current ISS using the
                  subsets identified in step 4.}
                  If PossibleUpdates is not empty then
                  For each element U of PossibleUpdates do
                  Begin
                        R' := Update(R,U);
```

```
                    NewDecompSoFar := DecompSoFar ∪ U;
                    Decompose(R',Outputs,NewDecompSoFar);
                End;
End.
```

As illustrated by the following example, the algorithm will find all deterministic decompositions subject only to the rather elementary heuristics[53] described earlier. The order of discovery of the decompositions does not imply any form of ranking. Even moderately complex systems are likely to have a very large number of possible decompositions. Further heuristics are needed to present the decompositions in some meaningful order.

3.5.2. A Simple Example

Recall the hypothetical system consisting of four interconnected lights. Light "a" is connected in series with "b" so that if "a" is on then "b" will be on and if "a" is off, "b" will be off. If light "a" is off then light "c" will be on, and if "a" in on, light "d" will be on. Only the state of light "a" may be set manually. The "on" state of a light will be represented by the integer 1 and the "off" state by 0.

Sublaws describing the stable states of the system and the actions to be taken should the system find itself in an unstable state, are given below.

Sublaws

1. Stability Conditions:

a	b
0	0
1	1

Corrective Actions:

Conditions		Actions
a	-->	b
1		1
0		0

2. Stability Conditions:

a	c
0	1
1	0
1	1

[53] The heuristics are embedded in the functions Subsystems() and Subsets().

Corrective Action:

Conditions		Actions
a	-->	**c**
0		1

3. Stability Conditions:

a	**d**
1	1
0	1
0	0

Corrective Action:

Conditions		Actions
a	-->	**d**
1		1

There are two external events.

External Events

1. Set a = 1
2. Set a = 0

The stable state space of this system is shown below.

Stable States

a	**b**	**c**	**d**
0	0	1	0
0	0	1	1
1	1	0	1
1	1	1	1

The first system relation may be obtained by applying the events "set a=1" and "set a=0" to each of the four stable states. This yields the first ISS or R_1. The final stable states corresponding to each of the states in the first ISS are obtained by examining the response paths of the system (these paths follow directly from the sublaws and are listed in Chapter 2).

First Intermediate State Space					Corresponding final stable state			
a	**b**	**c**	**d**	--->	**a**	**b**	**c**	**d**
0	0	1	0		0	0	1	0
0	0	1	1		0	0	1	1
0	1	0	1		0	0	1	1
0	1	1	1		0	0	1	1
1	0	1	0		1	1	1	1
1	0	1	1		1	1	1	1
1	1	0	1		1	1	0	1
1	1	1	1		1	1	1	1

Four deterministic decompositions will be suggested when the algorithm is applied to this system. The steps leading to the first deterministic decomposition are shown below. The full solution, showing the generation of all four decompositions, is included as Appendix H. Each step is labelled using the following convention:

x(Ay|Lz)

where

x = Step number starting with 1 and increasing by 1 until the algorithm finishes.

y = Algorithm step number.

z = The current level of recursion with respect to the Decompose() procedure.

START

1(A1|L1) Find the output state variables with respect to the current ISS.

The only state variables which change their values between the first ISS and the corresponding final stable states are {b,c,d}.

2(A2|L1) The set of output state variables is not empty.

3(A3|L1) Find the deterministic subsystems.

The first system relation was as follows:

First State	Intermediate Space				Corresponding final stable state			
a	**b**	**c**	**d**	--->	**a**	**b**	**c**	**d**
0	0	1	0		0	0	1	0
0	0	1	1		0	0	1	1
0	1	0	1		0	0	1	1
0	1	1	1		0	0	1	1
1	0	1	0		1	1	1	1
1	0	1	1		1	1	1	1
1	1	0	1		1	1	0	1
1	1	1	1		1	1	1	1

The smallest deterministic subsystems, with respect the the first ISS, which are described by at least one output state variable are {a,$\underline{b}$}, {a,c,$\underline{c}$}, and {a,d,$\underline{d}$}. Notice that state variables "c" and "d" are both inputs and outputs in their respective subsystems. That is, the final values of "c" and "d" are dependent on their initial values.

4(A4|L1) Find subsets of the deterministic subsystems for ISS update.

The subsets of this set of deterministic subsystems are
{{a,$\underline{b}$}} {{a,c,$\underline{c}$}} {{a,d,$\underline{d}$}}
{{a,$\underline{b}$},{a,c,$\underline{c}$}} {{a,$\underline{b}$},{a,d,$\underline{d}$}} {{a,c,$\underline{c}$},{a,d,$\underline{d}$}}
{{a,$\underline{b}$},{a,c,$\underline{c}$},{a,d,$\underline{d}$}}

5(A5|L1) Update the current ISS using one subset of the set of deterministic subsystems, and call the Decompose() procedure.

The first ISS will be eventually updated using all the sets found in step 5. The first set selected is {{a,$\underline{b}$}}. The second ISS created by this update is as shown below.

First ISS		Second ISS		Corresponding final stable states
a b c d	-->	**a b c d**	-->	**a b c d**
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 1 0 1		0 0 0 1		0 0 1 1
0 1 1 1		0 0 1 1		0 0 1 1
1 0 1 0		1 1 1 0		1 1 1 1
1 1 0 1		1 1 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1
1 1 1 1		1 1 1 1		1 1 1 1

6(A1|L2) Find the output state variables with respect to the current ISS.

The only state variables with values which change between the second ISS and the corresponding final stable states are {c,d}.

7(A2|L2) The set of output state variables is not empty.

8(A3|L2) Find the deterministic subsystems with respect to the second ISS.

The only deterministic subsystems described by at least one output state variable and one output state variable from the subsystems used in the last update, are {b,c,$\underline{c}$} and {b,d,$\underline{d}$}. Notice that while {a,c,$\underline{c}$} and {a,d,$\underline{d}$} are still deterministic subsystems, they are not described by an output state variable from the subsystems used to create the second ISS (i.e. they are not described by state variable b).

9(A4|L2) Find subsets of the deterministic subsystems for ISS update.

The subsets of this set of deterministic subsystems are
{{b,c,$\underline{c}$},{b,d,$\underline{d}$}} {{b,c,$\underline{c}$}} {{b,d,$\underline{d}$}}

10(A5|L2) Update the current ISS using one subset of the set of deterministic subsystems, and call the Decompose() procedure.

The second ISS will be eventually updated using all the sets found in step 9. The first set selected is {{b,c,$\underline{c}$},{b,d,$\underline{d}$}}. The third ISS created by this update is as shown below.

Second ISS		Third ISS		Corresponding final stable states
a b c d	-->	**a b c d**	-->	**a b c d**
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 0 0 1		0 0 1 1		0 0 1 1
0 0 1 1		0 0 1 1		0 0 1 1
1 1 1 0		1 1 1 1		1 1 1 1
1 1 1 1		1 1 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1
1 1 1 1		1 1 1 1		1 1 1 1

11(A1|L3) Find the outputs with respect to the current ISS.

There are no output state variables.

12(A2|L3) Since there are no output state variables, output a deterministic decomposition.

The sets of subsystems used to transform the first ISS into a stable states defines a decomposition. The second ISS was formed using {a,b}. The third was formed using {b,c,c} and {b,d,d}. Therefore, the first discovered decomposition is therefore

2: {b,c,c} {b,d,d}
1: {a,b}

This invocation of the Decompose() procedure is now complete. Execution will continue by updating the second ISS using the next subset of the set of deterministic subsystems identified at Step 9.

{The example is completed in Appendix H.}

A full list of the decompositions produced by the decomposition algorithm is shown below.

Decomposition #1
2: {b,c,c} {b,d,d}
1: {a,b}

Decomposition #2
2: {b,d,d}
1: {a,b} {a,c,c}

Decomposition #3
2: {b,c,c}
1: {a,b} {a,d,d}

Decomposition #4
1: {a,b} {a,c,c} {a,d,d}

Each of these decompositions represents a different view of the same system. The suitability of a particular decomposition will depend upon such considerations as:

1) Which state variables is the analyst interested in? (i.e. What is the goal of the system?), and
2) What maintenance changes are anticipated? (Maintenance considerations will be discussed in the next chapter.)

Decomposition #1 allows state variable "a" to be hidden with respect to outputs "c" and "d". An analyst interested in state variables "c" and "d" need only be concerned with the view of the system illustrated in first part of Figure 10. The arrows indicate value dependencies. For example, the arrow between "a" and "c" means that the final value of "c" depends on the value of "a".

Decompositions #2 and #3 cannot hide any information from an analyst interested in "c" and "d". In neither decomposition is "a" hidden with respect to both "c" and "d". For example, decomposition #2 yields the view shown in the centre of Figure 10 with respect to "c" and "d".

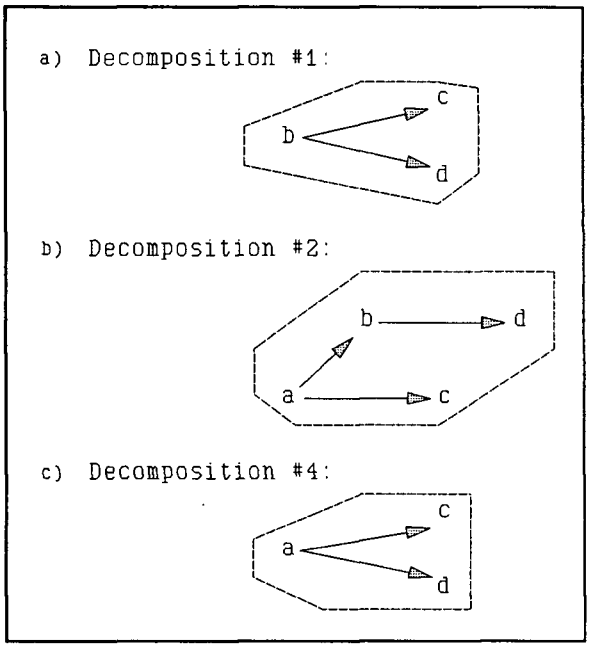


Figure 11: Three possible views of the four lights system.

Decomposition #4 shows that the final values of state variables "b", "c", and "d" can be calculated concurrently if the initial values of {a}, {a,c}, and {a,d} are known. It is also the decompsition inherent in the sublaws. This decomposition yields the view shown in the bottom of Figure 10 with respect to "c" and "d". In this case, state variable "b" is hidden with respect to "c" and "d". It is not immediately clear whether decomposition #1 or #4 is superior.

3.5.3. Importance of the External Event Space

Proper explication of the events which may act upon the system from the environment is crucial. For example, consider the following simple example. The system is intended to model the addition of two continuous quantities "a" and "b". State variable "c" contains the result of the addition. All three state variables are modelled as having only two values: zero and positive.

Stability Conditions:

a	b	c
pos	-	pos
-	pos	pos
0	0	0

Corrective Actions:

Conditions			Actions
a	b	-->	c
pos	-		pos
-	pos		pos
0	0		0

The only intuitively reasonable decomposition for this system is as follows:

1: {a,b,<u>c</u>}

However, if only one external event is defined as

Set a = pos

the specifications analysis tools will yield the following decomposition:

1: {a,c,<u>c</u>}

The defined external event is not sufficient to force the system to exhibit all of its dynamic properties. As a result, knowledge of the initial values of "a" and "c", as well as the value of "a" after the application of the external event, is sufficient to predict the final value of "c". Therefore, {a,c,<u>c</u>} is a deterministic subsystem. Defining external events

1. Set a = pos
2. Set a = 0

will yield the intuitively expected decomposition[54].

The four light example exhibits similar behaviour. If the only defined external event is

Set a = 1

the specifications analysis tools will identify two possible decompositions.

Decomposition 1:
1: {a,b} {a,d,d}

Decomposition 2:
2: {b,d,d}
1: {a,b}

The rules whose responsibility it is to set the value of state variable "c" are never activated. The value of "c" is never changed, therefore "c" cannot be identified as an output state variable.

To help ensure that the defined external events are sufficient to force the system to exhibit all behaviour implied by the defined state variable values, they should cause the affected state variables to assume all of their defined values. The specifications analysis tools perform a test to ensure that this is so. If the test fails, a warning message is issued to the analyst. This heuristic is based on the assumption that if an analyst defines several values for a state variable affected by an external event, he or she is interested in seeing that state variable assume each of these values as a result of external events[55].

---

[54] Defining further external events to alter the value of state variable "b" has no effect on the generated decomposition. No additional decomposition information is provided by such an event.

[55] State variables are not allowed to change their values twice during any response to an external event. Therefore, the values of state variables affected by external events can only be set by external events. The reason for this rule is described in this chapter under the heading "Intermediate State Variables".

3.4.4. Decomposition of the Payroll System

The simple "interactive" payroll system described in the previous chapter decomposes as shown below. To save space, variable names have been abbreviated as indicated. There are seven possible decompositions[56].

State Variable Abbreviations

hours = hours worked	pay_r = pay rate
emp_p = employee position	emp_t = employee type
sales = sales	base = base pay
com = commissions	over = over time pay
total_pay = total pay	

Payroll System Decompositions

Decomposition #1

1: {emp_t,emp_p,hours,over} {emp_t,emp_p,pay_r,hours,sales,total_pay}
{pay_r,hours,base} {emp_t,emp_p,sales,com}

Decomposition #2

2: {emp_t,emp_p,pay_r,hours,com,total_pay}

1: {emp_t,emp_p,hours,over} {pay_r,hours,base} {emp_t,emp_p,sales,com}

Decomposition #3

2: {emp_t,emp_p,hours,sales,base,total_pay}

1: {emp_t,emp_p,hours,over} {pay_r,hours,base} {emp_t,emp_p,sales,com}

Decomposition #4

2: {emp_t,emp_p,hours,base,com,total_pay}

1: {emp_t,emp_p,hours,over} {pay_r,hours,base} {emp_t,emp_p,sales,com}

---

[56] Careful examination of these decompositions will reveal that #1 through #6 may be deduced from #7 by simple substitutions of state variables. This issue will be addressed in Chapter 4.

Decomposition #5

2: {emp_t,emp_p,sales,base,over,total_pay}

1: {emp_t,emp_p,hours,over} {pay_r,hours,base} {emp_t,emp_p,sales,com}

Decomposition #6

2: {pay_r,hours,over,com,total_pay}

1: {emp_t,emp_p,hours,over} {pay_r,hours,base} {emp_t,emp_p,sales,com}

Decomposition #7

2: {base,over,com,total_pay}

1: {emp_t,emp_p,hours,over} {pay_r,hours,base} {emp_t,emp_p,sales,com}

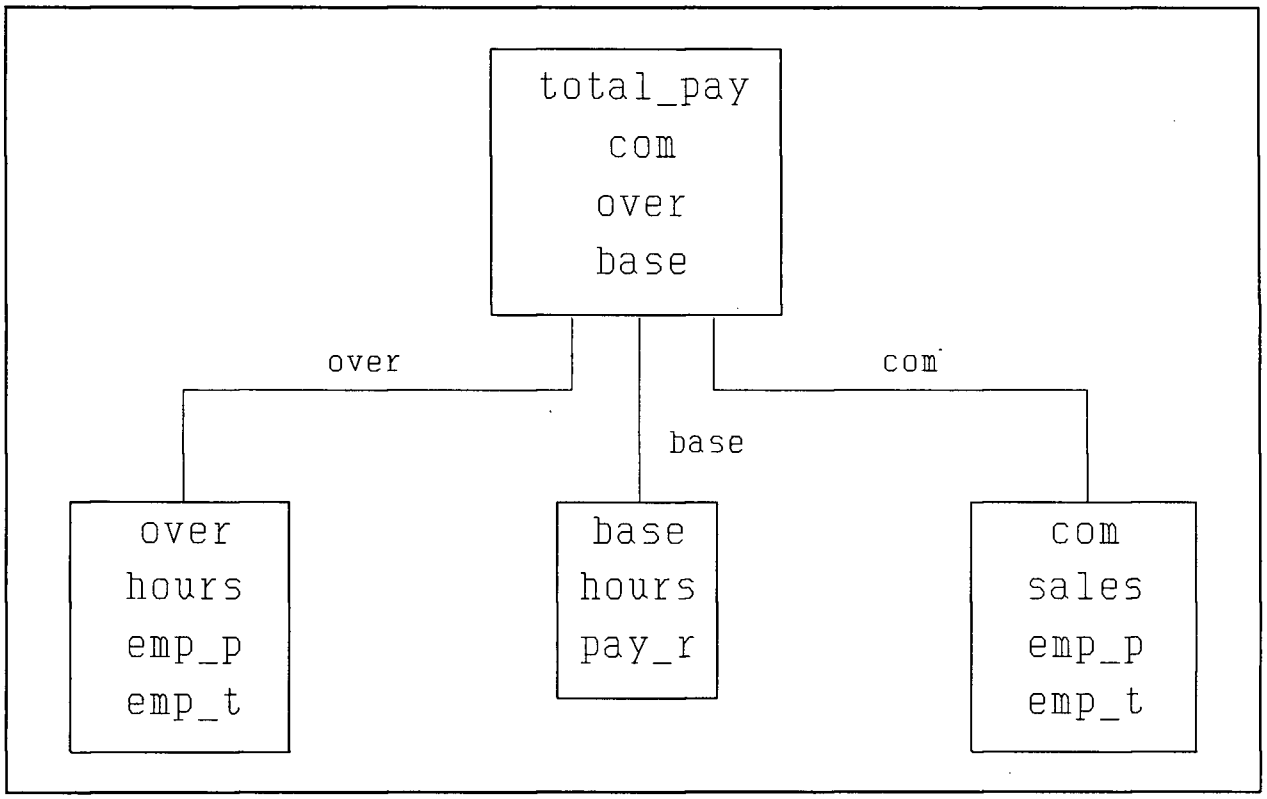


Figure 12: A diagrammatic representation of Decomposition #7 for the "interactive" payroll system. This is the decomposition reflected in the sublaws.

Although it is used in the system model, the "benefits" state variable does not appear in any of the decompositions of this system. Since it has only one possible value, its value cannot change. Therefore, "benefits" is not an output state variable. Also, it is not included in the calculation of any other state variable included in the model. Therefore, it does not appear as an input

state variable in any deterministic subsystem. However, suppose that for some reason "total pay" were to include "benefits". The value of the state variable "total_pay" would be dependent on the value of "benefits", and the model would have to be modified to reflect this dependency. For example, if the "benefits" state variable could have values "0" and "nz", the rules describing the calculation of "total pay" would have to be modified as shown below. Italicized text indicates the changes to the model described in Appendix 6.

```
/* calculate total pay */
dynamic("calculate total pay",
        [v(base,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(over,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(com,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(ben,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(base,"0"),v(over,"0"),v(com,"0"),v(ben,"0")],
        [v(total_pay,"0")]).
```

The specifications analysis tools would now suggest decompositions which included "benefits" as an input to the "total pay" subsystem.

The "batch" payroll system, described in the previous chapter, decomposes as above. However, the "batch" model gives rise to many decompositions which are not generated using the "interactive" model[57]. The specifications analysis tools produce a total of 168 decompositions. All suggested decompositions of the "batch" payroll system have been included as Appendix I. Most are a direct result of the batch orientation. The state variable "end" is not the only state

[57] A "benefits" subsystem is included in the "batch" model. As shown in the appendix, this subsystem is independent of all other subsystems and is not responsible for the increase in the number of alternative decompositions.

variable which may be used to indicate the end of the period. Any output state variable which has had its value calculated may be used to trigger the calculation of another output state variable. For example, consider the following decomposition:

Decomposition #2

2: {pay_rate,hours,benefits,base}

1: {end,benefits} {end,emp_t,emp_p,sales,com}
{end,emp_t,emp_p,hours,over}
{end,emp_t,emp_p,pay_rate,hours,sales,total_pay}

The calculation of all output state variables at level 1 is triggered by the value of the state variable "end". This fact is indicated by the inclusion of "end" in each subsystem at that level. The subsystem at level 2 indicates, as expected, that "base pay" may be calculated from "pay rate" and "hours worked". However, the calculation of "base pay" is triggered by the calculation of "benefits". As soon as the value of "benefits" becomes non-zero, "base pay" is calculated. Generation of decompositions of this form is not considered to be an error.

3.5.5. Intermediate State Variables

Consider a modification of the "interactive" payroll system. Assume the company makes some changes in its payroll policy (Wand and Weber, 1989).

1. Both office staff and sales employees are entitled to both overtime pay and sales commissions.
2. An office employee cannot receive more in commissions than in overtime.
3. A sales employee cannot receive more in overtime than in commissions.

An analyst might be tempted to define a system which, after calculating both commissions and overtime, modifies these amounts to reflect the restrictions resulting from changes 2 and 3. The sublaws could still pass the tests for local completeness and consistency. Response paths could be determined and the first ISS could be created. The decomposition algorithm could be applied, but none of the resulting decompositions would indicate the calculation of the

intermediate values for "overtime" and "commissions". That is, no decompositions of the following form would be found. (Notice that "overtime" and "commissions" are output state variables at two different levels of the decomposition.)

n+2: {...,commissions,overtime,total_pay}
n+1: {employee_type,commissions,overtime,commissions,overtime}
n: {...,hours,overtime} {...,sales,commissions}

Instead, subsystems at levels n and n+1 would be combined together as shown:

m+1: {...,commissions,overtime,total_pay}
m: {...,hours,sales,employee_type,commissions,overtime}

This will happen for the following reason. When an update is performed on an ISS, the state variables in each of the updating subsystems are set to their final values. There is no way to identify the intermediate values of "overtime" and "commissions" which would have been calculated at level n. Only by updating "commissions" and "overtime" to their intermediate values could the two-step nature of the calculation be discovered. Unfortunately, knowledge of these intermediate values is not contained in the information input to the decomposition algorithm. The only information available to the algorithm is the first system relation (i.e. the list of initial unstable states and their corresponding final stable states).

This observation may be stated more generally.

> The specifications analysis tools will never suggest a decomposition where a state variable is an output state variable at more than one level of a system.

If an analyst wishes to show the "multiple-step" nature of a calculation, he or she must identify INTERMEDIATE STATE VARIABLES. In the above example, such a state variable might be called "additional_payments". Suppose "employee type", "commissions" and "overtime" are used to calculate "additional payments". Also suppose "additional payments" is input to the subsystem calculating "total pay". The algorithm would identify a decomposition with the following form.

n+2: {...,additional_payments,total_pay}

n+1: {employee_type,commissions,overtime,additional_payments}

n: {...,hours,overtime} {...,sales,commissions}

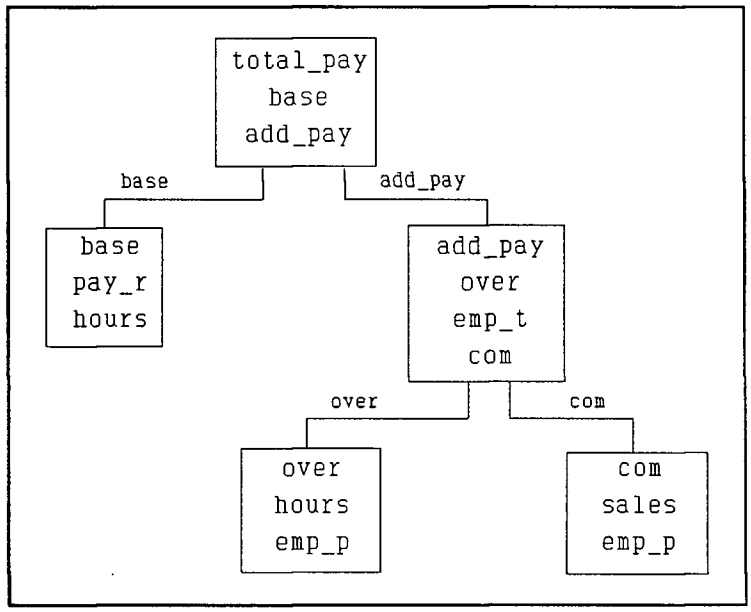


Figure 13: A diagrammatic representation of Decomposition #27 for the modified payroll system. This is the decomposition reflected in the sublaws.

The required use of intermediate state variables is not a restriction on the generality of SELMA. In fact, it could be argued that the commissions and overtime amounts before and after the restrictions are applied are not the same properties of the system. That is, "commissions before restrictions" is not the same state variable as "commissions after restrictions". Perhaps an analyst wishing to model them as the same state variable is actually making a mistake. This mistake might be caused by thinking about the system in procedural rather than sublaw oriented terms. Thus the required use of intermediate state variables can be seen as a kind of semantic integrity check. That is, if none of the decompositions suggested by the tools exhibit the structure intuitively expected by the analyst, some state variables may be serving dual roles and additional state variables may be required.

A listing of the formal model for the modified payroll system has been included as Appendix J. A total of 48 decompositions are suggested by the specifications analysis tools. These are also included as Appendix K. The decomposition matching the structure of the sublaws is shown in diagrammatic form in Figure 13. The fact that so many decompositions are generated highlights the need for additional heuristics to reduce the selection task faced by a designer. Some additional heuristics will be discussed in the next chapter.

3.6. Conclusions

System decomposition can be performed by considering the manner in which the values of the state variables describing the system change under the influence of external events. A theory of decomposition embodying the concepts of emergent and hidden system state variables has been developed. An algorithm for decomposing systems utilizing Wand and Weber's requirement for deterministic decompositions has been described.

The goal of this theory of decomposition is quite different from the formalisms of HOS (Hamilton and Zeldin, 1976) and Mili et al. (1986). These formalisms focus on ensuring that given subsystems are combined in a consistent manner. They are not concerned with the identification of subsystems. Computerized tools implementing their ideas would be "passive" in nature. That is, the tools would merely test the consistency of given decompositions. Myers (1978) and Yourdon and Constantine (1979) were concerned with developing a methodology for actively finding deterministic subsystems. Similarly, the specifications analysis tools are "active" in the sense that they can suggest decompositions for a system. However, while the techniques of Myers and Yourdon and Constantine are informal and depend to a great extent on human judgement, the algorithm used by the specifications analysis tools is derived from a theory of decomposition and may be completely automated.

Two of the three basic forms of decomposition identified in Chapter 1 are supported by the theory. Wand and Weber's requirement is used in conjunction with several heuristics to identify subsystems which are candidates for parallel decomposition. The processes[58] associated with the subsystems at any level of a decomposition may be executed in parallel. No subsystem will have an associated process which depends on the output of another subsystem at the same level. The update is the essence of sequential decomposition. The ISS formed by an update using certain subsystems, represents the states of the system after the processes associated with those subsystems have been completed. Parallel decomposition may be performed following the construction of the first ISS or after any update. A possible interpretation of the third generic form of decomposition, namely conditional decomposition, is discussed in the Chapter 5.

---

[58] The relational form (ie. initial/final state pairs) of these processes could be obtained directly from the system relation with which the subsystem is associated.

The decompositions generated by the specifications analysis tools could provide a basis for either analysis of the system or design of some artifact intended to represent the system (as in the case of a computerized information system). For analysis, the subsystems identified by the specifications analysis tools will be guaranteed to behave in a deterministic way. This will reduce the cognitive load required to comprehend the operation of the entire system. For design, the decomposition can provide the basis of a hierarchy of program modules as required by structured programming. As well, a deterministic subsystem identified by the specifications analysis tools could be easily implemented as an object in an object-oriented programming system. The state variables describing the subsystem would comprise the state vector of an object type. The processes or methods encapsulated with this state vector could be described by a sublaw specifying the relationships between state variables.

The example systems considered in this section were quite small. It is likely that larger systems will give rise to even greater numbers of deterministic decompositions. The next chapter suggests a "ranking" heuristic which could be used to present the analyst with the "best" decompositions first. Another heuristic for reducing the size of the decomposition search space is also suggested.

Chapter 4: System Complexity, Maintenance, and Goals

4.1. General

The last chapter showed how the internal structure of a system may be discovered given only the states resulting from the action of external events. This internal structure is found through the use of a decomposition algorithm based on a number of heuristics. The heuristics serve to limit the number of "possible" decompositions which must be considered by the analyst. However, as illustrated by the simple payroll system examples, these heuristics still allow a large number of decompositions. Some method of ranking these decompositions is required so that only the best need be presented to the analyst.

To this point, all subsystems produced by the specifications analysis tools have been considered to be equally suitable as bases for the construction or understanding of a system. For example, the subsystems

{emp_p,emp_t,hours,pay_r,sales,total_pay}

and

{base,add_pay,total_pay}

where hours = hours worked
pay_r = pay rate
emp_p = employee position
emp_t = employee type
sales = amount of sales
base = base pay
total_pay = total pay
add_pay = additional payments

suggested for the modified payroll system, are considered to describe equally suitable modules for the calculation of total pay. The first subsystem suggests calculation of total pay given only the initial inputs (or the state variables affected by external events), whereas the second suggests making use of the intermediate values: base pay and additional payments. Most analysts would agree that implementation (or understanding) of the first subsystem would be more difficult than the second. The subsystem calculating total pay from initial inputs would be more complicated than the subsystem utilizing the intermediate values. Therefore, the second subsystem likely describes a superior subsystem

in that its complexity is less than the first. Unfortunately, the intermediate values of base pay and additional payments must be calculated before the second subsystem may begin calculation of total pay. These lower-level subsystems might increase the complexity of the system beyond the complexity of a calculation from initial inputs. A quantitative measure of decomposition complexity is required so that different candidate decompositions may be compared and ranked.

This chapter is primarily concerned with the selection of such a complexity measure. As there is no general consensus on the meaning of the term "complexity", the chapter will begin with a discussion of some necessary characteristics for a measure of complexity suitable for use with systems modelled using SELMA. The final selection will be rationalized by tracing the logical development of the measure beginning with Ashby's (1956) definition of system "variety". Variety will be modified to provide some additional desirable properties. After a logarithmic transformation, the modified variety measure is identical to "entropy" as defined by Shannon (1948). While entropy will be shown to be unsuitable as a measure of complexity, its problems can be overcome with a simple modification. This modification was first made by Hellerman (1972). He called the resulting measure "computational work"[59]. Computational work has been adopted as the measure of complexity for this research. In summary, rationalization of the complexity measure shall consist of four major stages:

1. Ashby's Variety
2. Modified Variety
3. Shannon's Entropy
4. Hellerman's Computational Work

---

[59] Hellerman's choice of the label "computational work" is in many ways unfortunate. His measure does not reflect the number of machine operations required to perform a calculation. This sort of machine work would be highly implementation dependent. Hellerman uses a multiplication subroutine as an example. The subroutine could calculate 38 * 73 by adding 73 to itself 38 times. However, there are easier ways to perform multiplication which require far less machine work. Hellerman was interested in finding a measure of the difficulty of a calculation which would be implementation independent. He also notes that, in the Computer Science literature, complexity is a quantity which varies directly with work, "and so may be identified, loosely, with it" (Hellerman, 1972, p. 439).

Application of the complexity measure will be demonstrated using the modified payroll system example of the previous chapter. The complexity of anticipated future changes to a system (or system maintenance) are expected to influence design. These influences will be illustrated using the modification of the payroll system described in the previous chapter. The chapter will close with the definition of another heuristic for pruning the decomposition search tree. This heuristic uses the measure of complexity and depends upon knowledge of the system's purpose or goal.

4.2. Complexity

The Oxford Dictionary defines something as complex if it "consists of parts". Most people would agree that something is complex if it is made up of many parts. A block of ice is usually not considered to be a complex object, whereas a space shuttle is very complex. Thus "many-partedness" does seem to be an essential ingredient for complex things. However, a mountain need not be considered to be a complex object even though it consists of a very large number of individual pieces of rock, and the block of ice would be a complex object if the motions of individual electrons and nuclei were considered. Clearly, although many-partedness is important, it must be many-partedness at the level of abstraction where the behaviour of interest is manifested. That is, if we are only concerned with the static behaviour of mountains, they are indeed simple things. However, if we are interested in patterns of erosion, such as landslides, or even geological uplift, then mountains become fairly complex systems of interacting strata and faults. Similarly, if we are interested in the gross (or emergent) properties of a block of ice, the block may be treated as a simple thing. But if we are concerned with "lower-level" properties of ice, such molecular bonding via electron sharing, the same block must be regarded as a complex system. Therefore, the following necessary criterion for a definition of complexity is proposed.

Complexity must be related to the behaviour of a system.

Any acceptable definition must recognize that complexity is related to the dynamics of the system[60].

4.2.1. Variety

The first step in the rationalization of a measure of complexity will be variety. Ashby (1956) notes that most systems of interest have outputs. He defined variety to be the number of different output states exhibited by a system. For example, consider the following two subsystems from the modified payroll system. A table of input and output states for each subsystem is provided.

Subsystem #1: {hours,pay_r,base}

> Base pay will only be non-zero (abbreviated "nz") if the pay rate is non-zero and the hours worked is non-zero (i.e. Hours worked is either less than the limit for regular hours "reg", or sufficient for overtime pay "ot").

Inputs		Output
hours	**pay_r**	**base**
0	0	0
0	nz	0
reg	0	0
reg	nz	nz
ot	0	0
ot	nz	nz

[60] Notice that this requirement is somewhat at odds with the common usage of the term "complex". Many people would consider an assembly consisting of two parts to be more complex than an assembly consisting of only one part. Further, they might continue to support this ranking even if the assemblies exhibited no behaviour other than simple "existence". The notion of complexity, as presented in this research, is more restrictive in that it does not address this sort of "static complexity". It will be argued that only "dynamic complexity" is important in assessing the quality of a decomposition.

Subsystem #2: {emp_p,sales,com}

Commissions will only be non-zero if the employee has a regular position (abbreviated "r") as opposed to a management position (abbreviated "m") and some sales have been made.

Inputs		Output
emp_p	**sales**	**com**
r	0	0
r	nz	nz
m	0	0
m	nz	0

The variety of both subsystems is 2 since both base pay and commissions may exhibit two distinct values.

Variety is at least similar to complexity. It seems intuitively reasonable to expect a system exhibiting a large number of output states to be more complex than one which shows only a small number of output states. However, complexity appears to be a function of more than just output states. Consider the following possible partial implementations of the base pay and commissions subsystems:

```
procedure base_pay(hours,pay_r,base);
begin
case pay_r of
0:    base := 0;
nz:   case hours of
      0:    base := 0;
      reg:  base := nz;
      ot:   base := nz;
      endcase;
endcase;
end;

procedure commissions(emp_p,sales,com);
begin
case sales of
```

```
0:      com := 0;
nz:     case emp_p of
        m:      com := 0;
        r:      com := nz;
        endcase;
endcase;
end;
```

The base pay calculation procedure is slightly longer than the one for commissions because there are more input states to consider. Because there are three possible values for hours worked and two for pay rate, the base pay subsystem has six input states. The commissions subsystem has only four input states. This suggests that a measure of complexity which is not only a function of output states, but of input states as well, is required. Such a measure will form the next step in the development of a measure of system complexity.

4.2.2. Modified Variety

Each of a deterministic system's input states will lead to one and only one output state. The probability of observing a particular output state is equal to the probability of observing any of the input states leading to that output state. Variety can be modified to be a function of the probabilities of observing each output state. Thus the modified measure would be a function of both input and output states. For systems modelled using SELMA, the probability of observing a particular output state is determined by the frequencies of the external events. The analyst could be asked to estimate these frequencies. They are, after all, likely required to facilitate implementation-level decisions relating to such things as data storage location or file access method. However, for purposes of analysis and design, an analyst is not concerned with the probability of an external event, only with understanding or designing the system's response to that event. For example, a computer program must contain routines to handle all anticipated inputs. The fact that a particular input may occur more often than another does not influence the difficulty of the code written to handle that input. Therefore, for purposes of analysis, it shall be assumed that the probability of observing each external event is the same. For subsystems, this is the same as assuming the

probabilities of observing all input states are equal. Therefore, for this research, the probability p_i of observing a given output state O_i will be defined as I_i/I where I_i is the number of input states leading to O_i and I is the total number of input states.

$$p_i = I_i / I$$

There are many possible ways of incorporating output state probabilities into a modified measure of variety. However, for consistency, the modified measure should yield the same value as Ashby's variety when probabilities do not matter (i.e. when they are all equal). It should also satisfy a somewhat less intuitive requirement. By definition, if the probability of observing output state O_1 is less than that of observing output state O_2, there are fewer input states leading to O_1 than to O_2. As illustrated below, this means that fewer decisions must be made before moving the system to state O_1 than to O_2. Therefore, when two systems exhibit the same input and output states, the system in which the output probabilities are most unequal will be the least complicated. That is, modified variety should be a maximum when the probabilities of observing each output are the same. Consider two simple systems with four input and two output states.

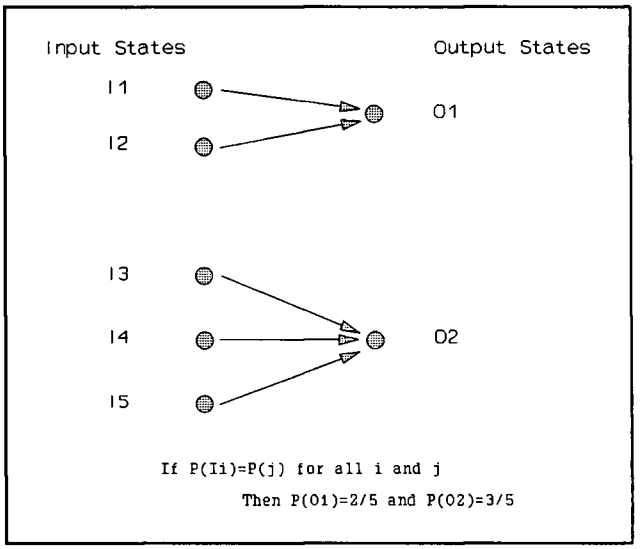


Figure 14: Probabilities of observing output states given equal input state probabilities.

System 1: {a,b,c} Probabilities of observing each output state are NOT equal.

Inputs		Output
a	**b**	**c**
0	0	0
0	1	0
1	0	0
1	1	1

System 2: {d,e,f} Probabilities of observing each output state are equal.

Inputs		Output
d	e	f
0	0	1
0	1	0
1	0	0
1	1	1

Possible implementations for these systems are as follows:

```
procedure c(a,b,c);
begin
      case a of
      0:    c := 0;
      1:    case b of
            0:    c := 0;
            1:    c := 1;
            endcase;
      endcase;
end;

procedure f(d,e,f);
begin
      case d of
      0:    case e of
            0:    f := 1;
            1:    f := 0:
      1:    case e of
            0:    f := 0;
            1:    f := 1;
            endcase;
      endcase;
end;
```

In the first system the probabilities of observing outputs of 0 and 1 are 0.75 and 0.25 respectively. In the second system the probabilities are both 0.50. The implementation for the second system is slightly longer (or more complex) than the one for the first.

One measure exhibiting both of the above properties is as follows:

$$\text{Modified Variety} = \prod_{i=1}^{n} (1/p_i)^{p_i}$$

where

n is the number of output states

p_i is the probability of observing output i

and

$p_i = I_i/I$

where

I_i is the number of input states leading to output i

I is the total number of input states

If all the I_i's are the same, all the p_i's will be equal to $1/n$ and

$$\text{Modified Variety} = \prod_{i=1}^{n} n^{1/n}$$

$$= n$$

$$= \text{Variety}$$

as desired.

In the base pay subsystem there are 6 different input states. Two of these states lead to a non-zero output state and 4 lead to output states of zero. Therefore the probabilities of observing output states of non-zero and zero are 0.33 and 0.66 respectively. Therefore, the modified variety of the base pay subsystem is 1.89 ($= (1/0.33)^{0.33} + (1/0.66)^{0.66}$). The modified variety of the commissions subsystem may be similarly calculated to be 1.76. This implies that

the commissions subsystem is in some sense less complex than the base pay subsystem, but both are less complex than a subsystem which would produce the values "nz" and "0" with equal probability. Because there are two output states, the modified variety of such a system would be 2.00.

At this point a digression is in order. In the above implementations, the only language primitives assumed were a selection structure in the form of "case" and an assignment operator in the form of ":=". Different languages are likely to have different primitives. For example, most languages have a multiplication operator. Such an operator would greatly simplify calculation of base pay since it is merely the product of hours and pay rate. The case structures could be eliminated. However, a multiplication operator would not much simplify the calculation of commissions as a selection depending on employee position is required. That is, no commissions are calculated for management employees. As the complexity measure is to be used to help select a decomposition for use as a basis for system implementation, the primitives of the implementation language are obviously important. It is possible to imagine a fourth-generation language which provides a primitive for the calculation of total pay given the initial inputs of employee position, employee type, hours, pay rate and sales. If this language was to be used for implementation of the payroll system, software written using any of the more detailed decompositions would likely be more complex than software written for the monolithic subsystem.

Also consider a system with the following inputs and output.

Inputs		Output
a	**b**	**c**
1	1	2
1	2	3

This system has a modified variety of 2. Now consider another system.

Inputs		Output
d	**e**	**f**
1	1	2
1	2	3
2	1	3
2	2	4

This system has a modified variety of 2.83. Yet both systems could be implemented identically using a simple addition primitive. But is addition really simple? Decimal addition requires the use of a 100 entry look-up table (0+0=0,0+1=1...9+9=18) and a set of rules for "carrying" (or "borrowing" in the case of negative addition or subtraction). Of course, addition in binary is simpler than addition in decimal but is still a non-trivial exercise. In the case of the base pay subsystem, multiplication was suggested to be a simple operation. In fact, not too many years ago some publishers were able to make a profit selling large look-up tables of logarithms which could be used in conjunction with the addition look-up table and rules (hopefully contained within the user's brain) to simplify multiplication. Modified variety provides a measure of the basic difficulty of a procedure. It is independent of whatever language primitives will be available during implementation. It is often argued that implementation issues, such as language selection, should not be considered during the early stages of systems analysis. It is these early stages that SELMA is designed to support. In fact, mathematical operations such as addition and multiplication are not likely to appear in the early stages of systems analysis but are more likely to be found in later stages where the procedures are developed to calculate emergent state variables. The payroll systems used as examples here are quite "low-level" in their focus. That is, the actual procedures used to calculate total pay are likely to be of interest only in the later stages of the analysis of an entire personnel and accounting system. This is not to say that SELMA is not applicable to such a low-level system. Rather, it is the complexity heuristic which is of questionable use at levels of analysis close to implementation because of the variety of different implementation primitives available.

Back to the discussion of variety. It would also be nice if the modified variety of a system formed by merging two independent subsystems could be found by combining the modified varieties of the subsystems in some simple way. In fact, as shown in Appendix M, the modified variety of such a system is simply the product of the modified varieties of the subsystems. Consider the system formed by merging the total pay and commissions subsystems:

{emp_p,hours,pay_r,sales,base,com}

The input and corresponding output states for this system are as follows:

Inputs				Outputs	
emp_p	**hours**	**pay_r**	**sales**	**base**	**com**
r	0	0	0	0	0
r	0	0	nz	0	nz
r	0	nz	0	0	0
r	0	nz	nz	0	nz
r	reg	0	0	0	0
r	reg	0	nz	0	nz
r	reg	nz	0	nz	0
r	reg	nz	nz	nz	nz
r	ot	0	0	0	0
r	ot	0	nz	0	nz
r	ot	nz	0	nz	0
r	ot	nz	nz	nz	nz
m	0	0	0	0	0
m	0	0	nz	0	0
m	0	nz	0	0	0
m	0	nz	nz	0	0
m	reg	0	0	0	0
m	reg	0	nz	0	0
m	reg	nz	0	nz	0
m	reg	nz	nz	nz	0
m	ot	0	0	0	0
m	ot	0	nz	0	0
m	ot	nz	0	nz	0
m	ot	nz	nz	nz	0

The variety of this combined system is

$$(24/12)^{12/24} * (24/4)^{4/24} * (24/6)^{6/24} * (24/2)^{2/24} = 3.32$$

which is also equal to the product of the varieties of the original subsystems (1.89 * 1.76 = 3.33) ignoring some round-off error. This result can be easily generalized to systems formed by merging more than two independent subsystems.

It was proven that the modified variety of a system formed by merging independent subsystems could be found by multiplying the modified varieties of the components. However, since multiplication is not as easily visualized as

addition[61], a logarithmic transformation of the modified variety measure is commonly used. This transformed measure is called entropy.

4.2.3. Entropy

Shannon (1948) was the first to propose a definition of information entropy[62] although Ashby did not suggest the notion of variety until several years later. Shannon was looking for a measure H of the "degree of choice or uncertainty" in the selection or occurrence of an output state which would be a function of the probabilities of observing each output state $p_1, p_2, \ldots, p_n$. He also wanted the measure to have a number of desirable properties (Shannon, 1948, pp. 392-393).

1. H should be continuous in the p_i.
2. If all the p_i's are equal, $p_i = 1/n$, then H should be a monotonically increasing function of n. With equally likely events there is more choice, or uncertainty, when there are more possible events.
3. If a choice can be broken down into two successive choices, the original H should be the weighted sum of the individual values of H.

He concluded that the only H satisfying all of these conditions is of the form[63].

$$H = \sum_{i=1}^{n} p_i * \log(1/p_i)$$

where again

---

[61] It is relatively easy to visualize the result of adding two things to a collection of four. It is much harder to visualize two things multiplied by three. Many people will visualize the multiplication as a series of additions. Addition is seen to be more intuitive than multiplication. Entropy is an additive measure while modified variety is multiplicative. Therefore, entropy is considered superior.

[62] For the remainder of this research, "information entropy" will be referred to as simply "entropy".

[63] The formula for H may be multiplied by, or added to, a constant and still possess the required properties.

n is the number of output states

p_i is the probability of observing output i

The base of the logarithm determines the units of entropy. The usual base is 2, and the units are bits. Logarithms in this document are always to base 2, although the actual units of entropy are irrelevant to this research.

Entropy is equal to the logarithm of the modified variety introduced in the previous section.

$$H = \log(\text{Modified Variety})$$

Since modified variety was introduced as a possible measure of system complexity, Shannon's "degree of choice", or entropy, of a system is also a possible measure of complexity.

In Shannon's work the p_i's were given. Here it is assumed that the probabilities of observing any input state is the same and that each p_i may be calculated as follows:

$$p_i = I_i/I$$

where

I_i is the number of input states leading to output i

I is the total number of input states

Shannon also noted that H has other properties which make it a reasonable measure of choice (pp. 394-395):

1. H = 0 if and only if all the p_i's but one are zero, this one having the value 1. That is, a system with only one output state has zero entropy.

2. Suppose there are two subsystems A and B with m and n output states respectively. Let p_{ij} be the probability of the joint occurrence of i as the output of the first subsystem and j as the output of the second. The system C formed by merging the two subsystems is

$$H_C = \sum_{i,j} p_{ij} * \log(1/p_{ij})$$

while

$$H_A = \sum_{i,j} p_{ij} * \log(1 / \sum_{i} p_{ij})$$

$$H_B = \sum_{i,j} p_{ij} * \log(1 / \sum_{j} p_{ij})$$

and it is easily shown that

$$H_C \leq H_A + H_B$$

Note: The previous examples of merged subsystems were for independent subsystems where $p_{ij} = p_i * p_j$ so that $H_C = H_A + H_B$.

Unfortunately, as a measure of complexity in the sense of coding or understanding difficulty, entropy is flawed. The second property seems to run counter to the heuristic in Chapter 3 which suggested that subsystems be kept as small as possible. The entropy of a merged process can be smaller than the sum of the entropies of the component processes. This is one undesirable property of entropy as a measure of system complexity. There is another. Consider the following systems and possible implementations:

{a,b}

Input	Output
a	b
1	1
2	2

```
procedure b(a,b);
begin
      case a of
      1:    b := 1;
      2:    b := 2;
```

```
      endcase;
end;
```

and

{c,d}

Input	Output
c	d
1	1
2	2
3	1
4	2

```
procedure d(c,d);
begin
      case c of
      1:    d := 1;
      2:    d := 2;
      3:    d := 1;
      4:    d := 2;
      endcase;
end;
```

The entropy of {a,b} is 1, but the entropy of {c,d} is also 1 despite the fact that its implementation is twice as long. This is a result of the fact that entropy is based on the probability of observing a given output. It is not dependent on the absolute number of input states which give rise to those outputs, but only on their ratios.

Solutions to both of these undesirable properties of entropy were suggested by Hellerman (1972). His measure has been selected as the estimate of system complexity for this research.

4.2.4. Computational Work

Hellerman (1972) was interested in estimating the amount of work done by a process independent of its implementation. His measure is equal to the amount of information stored in the look-up table implementation of a process. Look-up tables are a list of input and corresponding output states, and have been used to describe the dynamics of the systems discussed in this chapter. To determine the amount of information in a look-up table, Hellerman suggests performing an experiment. First the table is implemented in a computer memory by utilizing the concept of a DOMAIN CLASS. A domain class is the set of input states which map into a single output state. If there are N output states, there are N domain classes. If there are I input states, the look-up table may be implemented in a computer memory consisting of I locations by placing the output value corresponding to the jth input state in the jth location. An arbitrary memory location may then be selected and its contents examined. If I_i is the number of input states leading to the ith output state, the particular contents found in the selected memory location occur in I_i locations. Therefore, its probability of selection was I_i/I. According to information theory, the selection provided $\log(I/I_i)$ bits of information. The total information which may be extracted from the memory is then

$$\sum_{i=1}^{N} I_i * \log(I/I_i)$$

This is the total amount of information stored in the memory or the total information required by the process. Hellerman called this quantity the **computational work (W)** and it is equal to the number of input states multiplied by the entropy of the process.

$$W = \sum_{i=1}^{N} I_i * \log(I/I_i) = I*H$$

He also notes that, in the computer science literature, complexity is a quantity that varies directly with work and "so may be identified, loosely, with

it" (p. 439). In this research, an absolute value for the complexity of a process is not required. The measure need only provide relative levels of complexity, and be additive[64]. As noted earlier, the selection of "computational work" as the name for this quantity is perhaps unfortunate as its value is independent of any particular computer implementation. For the purposes of this research, "computational work (W)" shall be renamed "complexity (C)".

$$C = \sum_{i=1}^{N} I_i * \log(I/I_i)$$

This formulation of complexity avoids the two problems noted for entropy. The complexity of a system formed by merging several subsystems will always be greater than or equal to the sum of the complexities of the component subsystems. That is, if A and B are subsystems with complexities C_A and C_B respectively, and D is the system, with complexity C_D, formed by merging A and B, the following will be true.

$$C_D \geq C_A + C_B$$

In fact, Hellerman notes that if A and B have no inputs state in common, the complexity of C is given by

$$C_D = I_B * C_A + I_A * C_B$$

where I_A and I_B are the numbers of input states of subsystems A and B respectively (p. 442). Therefore, the heuristic calling for small subsystems can be justified from the standpoint of reducing overall complexity. The second

---

[64] Therefore, if A, B and D are processes with complexities C_A, C_B, and C_D respectively, and

$C_A > C_B$

then

$C_A + C_D > C_B + C_D$.

This property is possessed by both entropy and computational work.

problem, relating to entropy's reliance on only the proportions of input states leading to each final state, is also solved. Recall the two systems {a,b} and {c,d} described above. The complexity of {a,b} is equal to 2 (I = 2, H = 1). The complexity of {c,d} is twice this amount or 4 (I = 4, H = 1). This difference in complexity is intuitively reasonable when the possible implementations (given earlier) are considered.

4.2.5. States or State Variables?

In software cost estimation, a common input to module complexity calculations is the number of input variables (Halstead, 1977; Albrecht, 1979; Bailey and Basili, 1981; Rubin, 1983). If the sort of complexity being estimated in these calculations is the same as that described in this chapter, such a practice can only be justified if the number of variables is monotonically related to the number of input states. Perhaps, on average, this will be close to the truth, but it is only correct when all input variables have the same degree of interdependence and the same number of possible values. For example, if there are three input variables with 2 possible values each, and if there is no relationship relating the variables to each other, the number of input state is $2^3 = 8$. If another similar variable is added, the number of states would become 16, and so on. However, if a fourth variable with three possible values is added, the number of input states would become 24. Therefore, the number of input states to a module need not be monotonically related to the number of input variables, and a basic assumption of software cost estimation techniques is shown to be questionable.

4.3. Heuristic Guided Search

A measure of system complexity was required so that the decompositions generated by the algorithm of Chapter 3 might be presented to the analyst in a meaningful order[65]. Alternative decompositions will be presented in order of increasing complexity. The complexity of a decomposition is defined as being

[65] As indicated by a footnote in Chapter 3, there is no suggestion that the algorithm and ranking heuristic (ie. complexity) described here are the "best". It is possible that more efficient algorithms and more appropriate heuristics exist. This section is intended to show that automated decomposition and some sort of meaningful ranking of alternatives is possible.

equal to the sum of the complexities of its constituent subsystems at all levels of the decomposition[66].

The algorithm performs updates on an intermediate state space (ISS) using every possible subset of the subsystems which were deterministic with respect to that ISS. The complexity measure can provide the basis for a heuristic to select the subset most likely to lead to a "high quality" decomposition, where "high quality" is defined as low complexity. For example, suppose the modified payroll system has been updated once producing the partial decomposition shown below. Complexities of individual subsystems are shown following a "|".

State variable abbreviations:

hours = hours worked
pay_r = rate of pay
emp_p = employee position
sales = amount of sales
com = commission pay
over = overtime pay

1: {hours,pay_r,base}|5.51 {emp_p,sales,com}|3.25 {emp_p,hours,over}|3.90

This is not a full decomposition in that the second ISS, formed by the update at level 1, still contains unstable states. In particular, the state variables representing additional payments and total pay have not been updated to reflect their final values. The subsystems which are deterministic with respect to the second ISS (and which satisfy the other heuristics presented in Chapter 3) are

{base,com,emp_t,over,total_pay}|12.98
{base,emp_p,emp_t,over,sales,total_pay}|22.04
{base,com,emp_p,emp_t,hours,total_pay}|29.61
{com,emp_t,over,add_pay}|8.00
{emp_p,emp_t,over,sales,add_pay}|9.71

[66] Notice that the sum of the subsystem complexities is only the lower limit to Hellerman's complexity of the system. However, one of the important reasons for decomposing a system is to avoid having to visualize the entire system at once. An analyst deals with individual subsystems at each level of the decomposition. Therefore, the sum of the subsystem complexities is a reasonable estimate of the overall effort required to understand the system.

{com,emp_p,emp_t,hours,add_pay}|15.34

where emp_t = employee type
total_pay = total pay
add_pay = additional payments

A good search heuristic should indicate the subset of this set of subsystems which is most likely to lead to the lowest-complexity decomposition[67]. There are 15 subsets[68]. For each subset, the specifications analysis tools determine the minimum and maximum possible decomposition complexities, where that subset comprises level 2. That is, each possible update has, associated with it, a minimum and a maximum possible decomposition complexity. These minimum and maximum complexities are based on information already obtained during the search. The minimum possible decomposition complexity is equal to the sum of the subsystem complexities at all lower levels plus the total complexity of the subsystems used for update at the current level. In other words, the minimum possible complexity is determined by assuming that all potential higher-level subsystems have zero complexity. The maximum possible decomposition complexity is equal to the minimum possible complexity plus the complexities of the least complex subsystems known so far which can determine the final values of any remaining output state variables. The next update will be performed using the subset with the lowest associated minimum complexity. Minimum and maximum complexities can be used together to "prune" the search tree. For example, consider an update using the subset

{{base,com,emp_t,over,total_pay}|12.98,{com,emp_t,over,add_pay}|8.00}.

The total complexity of level 1 is 12.66 (= 5.51 + 3.25 + 3.90). Therefore, the minimum possible complexity of any decomposition arising from this update is 33.64 (= 12.66 + 8.00 + 12.66). After this update there will be no remaining output state variables. That is, in the third intermediate state space created

---

[67] This sort of heuristic search is sometimes called the "best bud" method (Sandewall, 1971).

[68] Recall that no two deterministic subsystems chosen for use in an update operation may contain the same output state variable. Therefore, there are only 3 * 3 + 6 = 15 possible update subsets.

by this update, all state variables will have reached their final values. Therefore, for this update subset, the minimum and maximum possible complexities are equal. As another example, consider an update using the subset

{{com,emp_t,over,add_pay}|8.00}

The minimum possible complexity of any decomposition arising from this update is 20.66 (= 8.00 + 12.66). The state variable "total_pay" will still be an output with respect to the third intermediate state space created by this update. The lowest-complexity deterministic subsystem discovered thus far which can calculate the final value of "total_pay" is

{base,com,emp_t,over,total_pay}|12.98

Therefore, the maximum possible complexity arising from this update is 33.64 (= 12.98 + 20.66). Minimum and maximum complexities for all possible update subsets are listed below. To reduce the size of the table, subsystems are coded as follows:

Subsystem	Complexity	Code
{base,com,emp_t,over,total_pay}	12.98	A
{base,emp_p,emp_t,over,sales,total_pay}	22.04	B
{base,com,emp_p,emp_t,hours,total_pay}	29.61	C
{com,emp_t,over,add_pay}	8.00	D
{emp_p,emp_t,over,sales,add_pay}	9.71	E
{com,emp_p,emp_t,hours,add_pay}	15.34	F

Update Subset	Minimum Possible Complexity	Maximum Possible Complexity[69]
{A}	25.63	33.63
{B}	34.70	42.70
{C}	42.27	50.27

[69] Minimum and maximum possible decomposition complexities will be the same whenever the update subset contains all of the remaining output state variables. For example, the subset {A,D} contains both "total_pay" and "add_pay". Therefore, the minimum and maximum possible decomposition complexities following an update operation using {A,D} are equal (33.64).

{D}	20.66	33.64
{E}	22.37	35.35
{F}	28.00	40.98
{A,D}	33.64	33.64
{A,E}	35.35	35.35
{A,F}	40.98	40.98
{B,D}	42.70	42.70
{B,E}	44.41	44.41
{B,F}	50.04	50.04
{C,D}	50.27	50.27
{C,E}	51.98	51.98
{C,F}	57.61	57.61

The update leading to the smallest minimum possible complexity uses the subsystem {over,emp_t,sales,add_pay} (i.e. update subset {D}). If this update is performed, the partial decomposition becomes

2: {com,emp_t,over,add_pay}|8.00
1: {hours,pay_r,base}|5.51 {emp_p,sales,com}|3.25 {emp_p,hours,over}|3.90

The only subsystems which are deterministic with respect to the third ISS (and satisfies the other heuristics of Chapter 3) are

{base,add_pay,total_pay}|3.25
and
{add_pay,hours,pay_r,total_pay}|11.02.

The minimum and maximum possible complexities associated with an update using the first subsystem are both 23.91. If the second subsystem is used, they are both 31.68. The first subsystem is selected for the next update. Thus the first decomposition reached, starting from the given level 1, is as follows:

3: {base,add_pay,total_pay}|3.25
2: {com,emp_t,over,add_pay}|8.00
1: {hours,pay_r,base}|5.51 {emp_p,sales,com}|3.25 {emp_p,hours,over}|3.90

The complexity of this decomposition is equal to the sum of the complexities of the individual subsystems or 23.91 (= 5.51 + 3.25 + 3.90 + 8.00 + 3.25). This is, in fact, the lowest-complexity decomposition of the modified payroll system. Alternative decompositions, with the same subsystems at level 1, can be found by performing updates using the other subsets of subsystems which were deterministic with respect to the second and third intermediate state spaces. These alternative subsets will be selected in order of increasing minimum possible complexity.

Using subsystem complexity to guide the search for decompositions with low complexity is not quite as straightforward as it appears. The above example started from a given set of subsystems at level 1, or a given second ISS. In fact, the specifications analysis tools, when applied to the modified payroll system suggest a large number of different possible level 1's. The following subsystems are all minimal (i.e. described by as small a number of state variables as possible) and deterministic with respect to the first ISS:

1. {hours,pay_r,base}
2. {emp_p,sales,com}
3. {emp_p,hours,over}
4. {hours,emp_p,emp_t,sales,add_pay}
5. {hours,emp_p,emp_t,pay_r,sales,total_pay}

With these five subsystems there are 31 subsets which might be used to form the second ISS. That is, there are 31 possible sets of level 1 subsystems, or 31 possible partial decompositions resulting from an analysis of the first ISS. The subset of subsystems, selected for the first update in the above illustration was

{{hours,pay_r,base},{emp_p,sales,com},{emp_p,hours,over}}.

Such an update eventually leads to the decomposition with the lowest-complexity, but this subset does not have the lowest minimum possible complexity at level 1. While decompositions with lower complexities are generally suggested first, there will often be exceptions.

It was mentioned earlier that maximum and minimum complexity information could be used together to "prune" the search tree[70]. If the analyst is able to specify an upper bound to the complexity of a decomposition he or she is interested in, some possible updates need never be performed. The specifications analysis tools allow the analyst to input the maximum percentage difference between the minimum complexity decomposition and any other suggested decompositions. If the percentage difference between the minimum complexity associated with a possible update and the maximum complexity associated with the lowest minimum complexity found so far is greater than the specified maximum percentage difference, that update will never be performed[71]. For example, recall from the above illustration, the possible update using the subsystem

{add_pay,hours,pay_r,total_pay}|11.02

The minimum possible complexity associated with the above subsystem is 31.68. The smallest minimum, and associated maximum, possible complexity found thus far were both 23.91. The percentage difference is 32% (= [31.68 - 23.91] / 23.91). If the analyst had specified 20% as the maximum percentage difference, this possible update would never be performed. Of course, should an analyst wish to see all possible decompositions irrespective of complexity, he or she can simply enter a very large number as the maximum allowed percentage difference.

4.4. Maintenance

In this research, SYSTEM MAINTENANCE refers to any changes to a system after implementation[72]. It will be shown that when system maintenance is

---

[70] Such pruning cannot result in the loss of the lowest complexity decomposition. The algorithm will still find the "optimal" decomposition of the system.

[71] This is a modified form of SSS* minimax search (Charniak and McDermott, 1985, pp. 286-290). The concept of "maximum allowable percentage difference" is added because the complexity measure is imperfect and, as will be shown in Chapter 6, other decompositions can help to identify shortcomings of the system model. In other words, higher-complexity decompositions can sometimes serve a useful purpose.

[72] In common usage, the term "maintenance" does not include enhancements to a system. The Oxford Dictionary defines "maintenance" as "being maintained", and "maintain" as "cause to continue". However, in keeping with the terminology

considered, the optimal (or lowest-complexity) decomposition for a system may change. It will be assumed that all future changes to a system may be defined to the degree of detail present in the system model[73]. This is a fairly strong assumption. In many cases, changes cannot be anticipated. In such cases, the best the analyst can do is select a low complexity decomposition for the original system ignoring possible changes.

Parnas (1972) and Myers (1977) suggest that the quality of a decomposition may be assessed by observing its behaviour in the face of maintenance changes. They assume the best decompositions will limit the effects of a change to a small number of subsystems. In this section, a framework for classifying maintenance changes will be developed. A technique for assessing the impact of maintenance on a given decomposition will also be proposed. It will be shown that in some cases it is best to construct parts of the original system with maintenance in mind, while in others it is best to ignore maintenance during initial construction, and to create entirely new subsystems when the maintenance must be done.

Because it is based on a limited number of basic constructs, SELMA provides a unique framework for classifying possible changes to an existing system. All possible changes to a system model can be categorized as follows[74]:

1. Changes to sublaws
   - add a sublaw
   - delete a sublaw
2. Changes to the set of external events
   - add an event
   - delete an event

As shall be shown, a change may cause a subsystem in a given decomposition to be no longer deterministic or no longer minimal (i.e. the subsystem is now

---

in the field of information systems, system maintenance shall include any change to a system including possible enhancements.

[73] That is, changes to the functional relationships between state variables, defined in the original system model, must be known before the original system is implemented.

[74] Modification of a sublaw or an event can be accomplished by deleting the old version and adding the new.

described by more state variables than are required to predict its behaviour). The above are SIMPLE CHANGES. A real MAINTENANCE OPERATION is likely to consist of several simple changes.

4.4.1. Changes to Sublaws

Consider the sublaw for calculation of base pay in the payroll system.

Stability Conditions:

base	hours	pay_r
0	-	0
0	0	-
nz	regular	nz
nz	overtime	nz

Corrective Actions:

Conditions		Actions
hours	pay_r -->	base
0	-	0
-	0	0
regular	nz	nz
overtime	nz	nz

where

0	=	zero
nz	=	not zero
regular	=	less than that required for overtime pay
overtime	=	sufficient for overtime pay
-	=	any value, or "don't care"

The analyst must specify each line of the above sublaw. During maintenance, any change to the sublaw can be represented as a sequence of additions and/or deletions of individual lines or rules. Therefore, at the lowest level, an analyst does not add and delete sublaws. Rather, he or she adds and deletes rules.

A change to a rule may or may not introduce new state variables to the system. Changes which do not introduce new state variables will be considered first. With respect to a given decomposition, the state variables affected by such a change will be

1. contained within a single subsystem, or

2. not contained within a single subsystem.

If the system after the change is still locally complete and consistent, the addition of a rule where all the state variables covered by the rule are included in a single subsystem will not affect the decomposition. For example, consider the effect, on the subsystem {hours,pay_r,base}, of adding the following rule to the base pay sublaw.

Corrective Actions:

Conditions			Actions
hours	**pay_r**	**-->**	**base**
nz	unknown		unknown

The addition of such a rule merely results in altering the functional relationship between the input and output state variables of the subsystem.

On the other hand, if the rule is added which covers state variables not found in any single subsystem, the subsystem may no longer be deterministic. For example, consider the addition the following rule to the base pay sublaw.

Corrective Actions:

Conditions				Actions
hours	**emp_p**	**pay_r**	**-->**	**base**
overtime	management	-		unknown

Hours and pay rate are no longer sufficient to determine the value of base pay. Knowledge of the employee's position is also required. Therefore, the subsystem {hours,pay_r,base} would be no longer deterministic.

Rules may also be deleted from a sublaw. When a rule is deleted, one or more of the subsystems in a decomposition may no longer be required. For example, consider the commissions sublaw from the original payroll system as shown below.

Stability Conditions:

emp_p	**emp_t**	**com**	**sales**
regular	sales	nz	nz
-	-	0	0
-	office	0	-
management	-	0	-

Corrective Actions			
Conditions			Actions
emp_p	emp_t	sales -->	com
regular	sales	nz	nz
-	-	0	0

If the first corrective action rule were deleted, employee position and employee type would no longer be required to determine the final value of commissions. Therefore, the subsystem {emp_p,emp_t,sales,com} would no longer be minimal[75].

When a rule in a system model is deleted, the state variables used in the rule may or may not be contained in a single subsystem of the given decomposition. That is, there is no reason to suppose that the sublaws specified in the system model will match exactly the sublaws describing the subsystems produced by the specifications analysis tools. The input to the tools is simply the first ISS (Intermediate State Space) of the system and the corresponding final stable states (i.e. the first system relation). The algorithm has no direct "knowledge" of the rules specified by the analyst. However, the effects of deleting a rule spanning more than one subsystem in the given decomposition will be the same as described above. One or more subsystems may be no longer required because some state variable is no longer an output, or some subsystem may no longer be minimal.

There is one more way in which a sublaw may be altered. A rule may be added which contains a state variable not previously used to describe the system. This was the case when the additional payments state variable was added to form the modified payroll system. The new state variable will be either an input state variable (i.e. its value does not change between the first ISS and the final stable states) or an output state variable. If it is an input state variable, it will simply be added to some subsystem or subsystems in the decomposition. For example, decomposition #27 of Appendix K shows "add_pay" added as an input state variable to the subsystem responsible for calculating the value of "total_pay". If it is an output state variable, it may be added as an output to an existing subsystem or a new subsystem may be formed to

---

[75] In fact, if the first corrective action rule is deleted, the subsystem will not only no longer be minimal, it will no longer be required. After such a maintenance operation, the value of commissions will never change. That is commissions is no longer an output state variable, and the subsystem whose responsibility it was to calculate commissions will no longer be required.

determine its final value. For example, decomposition #1 of Appendix K shows the creation of the subsystem (com,emp_t,over,add_pay) to determine the value of "add_pay". In either case the basic structure of the decomposition will not be altered.

4.4.2. Changes to External Events

The effects of changes to external events are similar to those of sublaws. After all, external events are functionally quite similar to sublaws. In fact external events can be thought of as sublaws with activation conditions located in the system's environment (and, therefore, not included in the system specification) and actions affecting state variables within the system.

If an external event affecting an existing state variable is added to a system, some new states may be added to the first ISS (since the first ISS is essentially the cross product of the stable state space of the system and the set of external events). As was shown in Chapter 3 under "Importance of the External Event Space", these new states may represent system behaviours which were not evident under the smaller set of external events. New subsystems may appear (because state variables which were previously constant may become outputs) and other subsystems may no longer be deterministic (because new behaviours may be exhibited.

If an external event is deleted from the system, some system behaviours may no longer be exhibited. This means that some subsystems of the given decomposition may disappear (because some state variables are no longer outputs), or some systems may no longer be minimal (because fewer input state variables are required to determine the final values of the output state variables).

Prediction of the effect of adding an external event which affects a state variable not previously used to describe the system, is trivial. No changes to the decomposition are expected. The system can only respond to events through the activation of rules. Since no rules mentioning the affected state variable exist, adding such an event can not affect system behaviour. Changes in behaviour will only occur if rules are added to respond to the new state variable. The effects of such changes were described earlier.

4.4.3. Implications for Design

When a system designer has knowledge of planned maintenance operations, he or she must decide whether to construct a system which can support these changes, or modify the initial system at a later date. This decision can be simplified by considering the possible effects a maintenance operation. The possible effects of changes to a system model are summarized in Table II.

Table II: Possible effects of simple changes to a system model.

Change		Effect
Sublaws:		
1.	add rule covering one subsystem	change form of sublaw associated with some subsystem
2.	add rule covering more than one subsystem	some subsystems may no longer be deterministic
3.	add rule with new state variables	expand existing subsystems or create new ones
4.	delete rule covering one subsystem	some subsystems may no longer be minimal
5.	delete rule covering more than one subsystem	some subsystems may no longer be minimal
Events:		
6.	add event affecting existing state variable	some subsystems may no longer be deterministic new subsystem may be added
7.	add event affecting new state variable	none, unless new rules are added
8.	delete event	some subsystems may no longer be minimal

A maintenance operation which results only in the removal of some subsystems will require very little maintenance effort. However, if the maintenance requires the addition of new subsystems, the inclusion of new state variables in old subsystems, or a change in the relationship between inputs and outputs of a subsystem, a great deal of effort may be required. Notice that the serious effects all occur when a rule or event is added to a subsystem. The problem of maintenance becomes one of identifying these serious effects when the initial system is designed. Merely identifying the simple changes involved in a maintenance operation, and looking up their effects in the above table, is not sufficient to predict specific effects. No change is certain to have an effect and the extent of effects which do occur cannot be easily determined. There is,

however, a way to identify all specific effects. The analyst must construct models for three systems[76]:

1. A model of the initial system.
2. A model of the modified system.
3. A single model describing the behaviours of both the initial and modified systems. This model will include a state variable to distinguish between the two versions of the system for use in subsystems where calculations are performed differently after the modification. Use of such a state variable is illustrated in Appendix N.

Decompositions produced for the combined system will contain only subsystems which behave deterministically with respect to the behaviours exhibited by both the initial and modified systems. The analyst must then compare decompositions for all three models and decide how the initial system should be implemented. Subsystems which are deterministic with respect to the behaviours of both systems could be implemented initially, or subsystems which are only deterministic for the first system could be implemented and then reconstructed when the maintenance operation is actually performed. For example, consider the initial and modified payroll systems. The original payroll system described in Chapter 2 was modified in Chapter 3 to reflect the following changes in company policy:

1. Both office staff and sales employees are entitled to both overtime pay and sales commissions.
2. An office employee cannot receive more in commissions than in overtime.
3. A sales employee cannot receive more in overtime than in commissions.

Appendix N contains a model for a system which will exhibit the behaviours of both payroll systems. The following is the lowest-complexity decomposition[77] for the combined system. Complexities are noted beside each subsystem. The

---

[76] The actual effort required to construct the three models is not likely to be prohibitive. Except when major maintenance changes are expected, the models will probably be quite similar.

[77] The complexity of a decomposition is equal to the sum of the complexities of the individual subsystems.

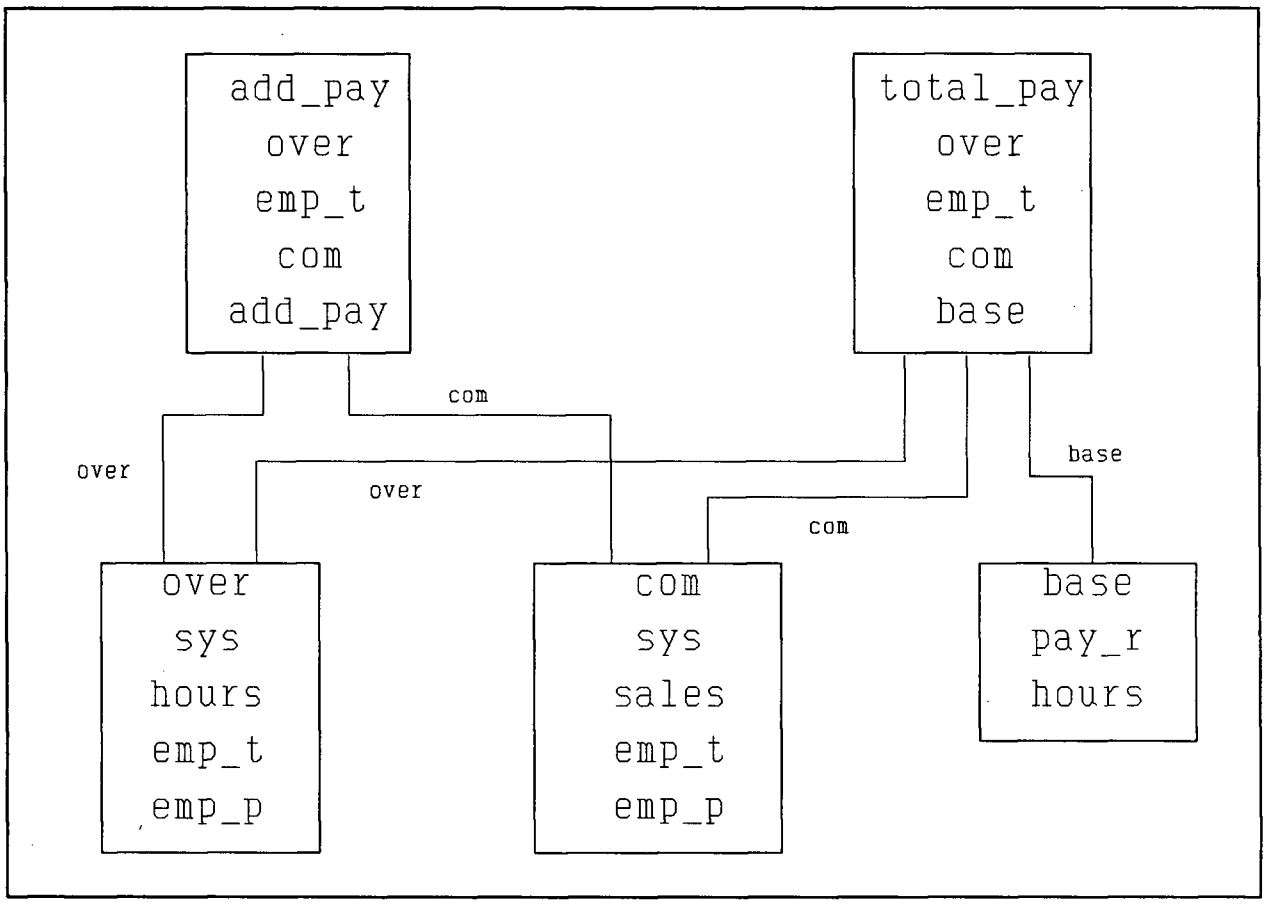


Figure 15: Lowest-complexity decomposition for the combined payroll system.

decomposition is shown using the diagrammatic format in Figure 14.

Lowest-Complexity Decomposition for the Combined Payroll System:

2: {add_pay,com,emp_t,over,add_pay}|19.02
{base,com,emp_t,over,total_pay}|12.98

1: {hours,pay_r,base}|5.51 {emp_p,emp_t,sales,sys,com}|11.14
{emp_p,emp_t,hours,sys,over}|13.05

where

emp_t	=	employee type (sales or office)
emp_p	=	employee position (management or regular)
hours	=	hours worked
sales	=	sales
over	=	overtime pay
com	=	commissions
add_pay	=	additional payments

total_pay = total pay

and "sys" is a state variable representing the version of the system. This state variable is used to avoid problems which might arise if rules for the two versions of the system conflict with each other. For example, in the commissions subsystem {emp_p,emp_t,sales,sys,com} there are two different ways to calculate commissions. The value of the "sys" state variable is used to determine which set of rules is to be activated. Notice that this decomposition is structurally similar to the lowest-complexity decomposition for the initial payroll system.

Lowest-Complexity Decomposition for the Initial Payroll System:

2: {base,com,over,total_pay}|3.90

1: {hours,pay_r,base}|5.51 {emp_p,emp_t,sales,com}|4.35
{emp_p,emp_t,hours,over}|4.97

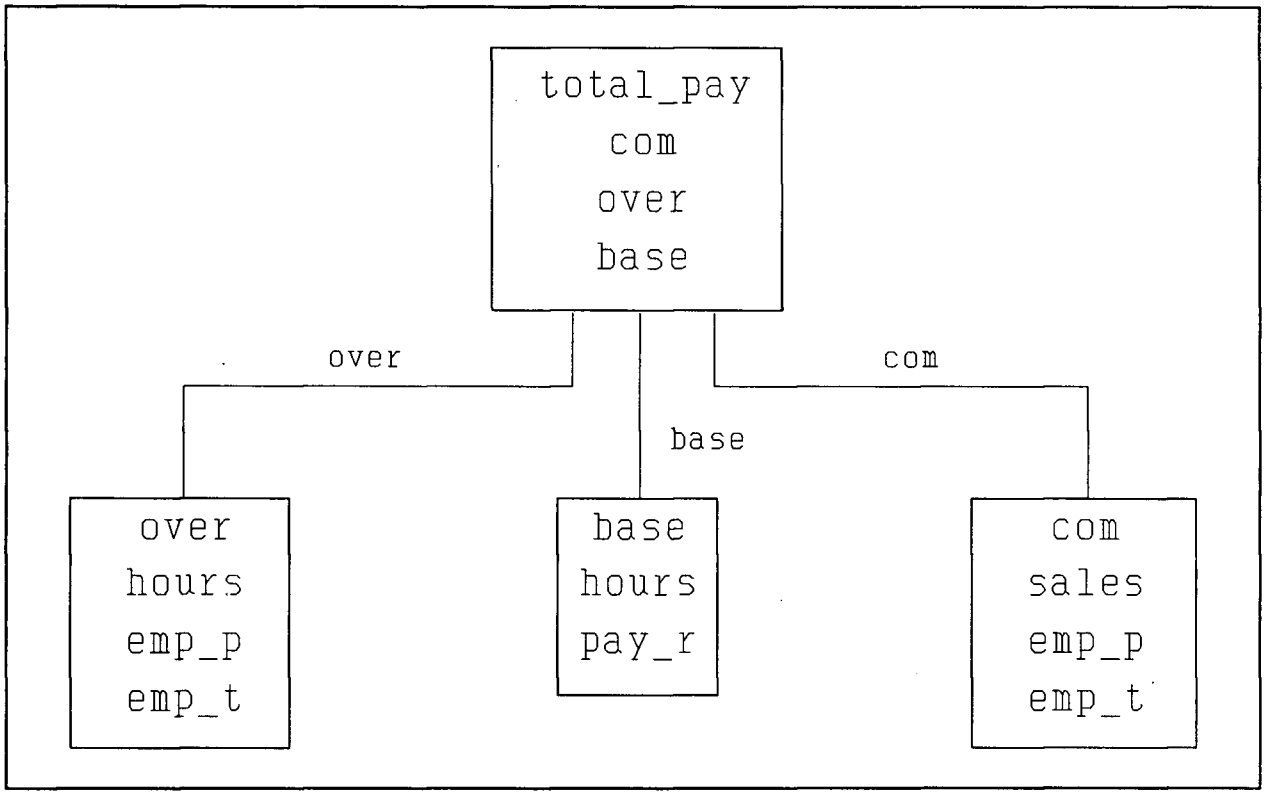


Figure 16: Lowest-complexity decomposition for the initial payroll system.

This decomposition is shown using the diagrammatic format in Figure 16. A subsystem for additional payments has been added and the "sys" state variable

is included in two subsystems to show that the behaviour of these subsystems depends on the version of the system. The lowest-complexity decomposition for the modified payroll system is shown below. Figure 17 displays this decomposition using the diagrammatic format.

Lowest-Complexity Decomposition for the Modified Payroll System:

3: {base,add_pay,total_pay}|3.25

2: {com,over,emp_t,add_pay}|8.00

1: {hours,pay_r,base}|5.51 {emp_p,sales,com}|3.25 {emp_p,hours,over}|3.90

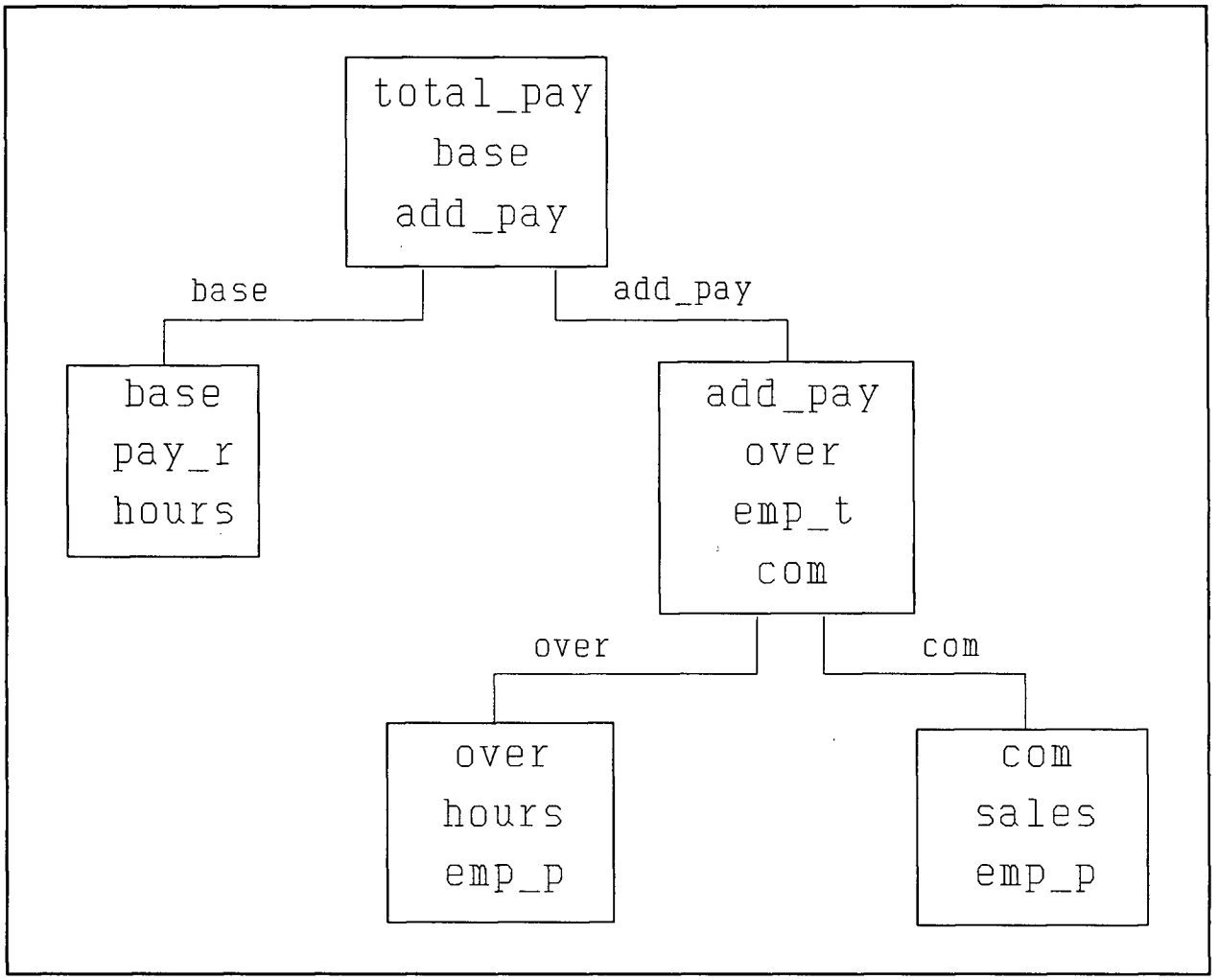


Figure 17: Lowest-complexity decomposition for the modified payroll system.

The decomposition of the combined system reveals three surprising aspects of the modification:

1. The lowest-complexity decomposition for the combined system places calculations of additional payments and total pay at the same levels. That is, additional payments does not become an input to the total pay calculation as in the modified system.
2. The calculation of total pay in the modified system does not require information as to the version of the system.
3. The calculation of additional payments in the combined system is more complex than in the modified system.

The first observation shows that the structure of the initial system is in some sense "dominant". The modification does not require major changes to the composition of any subsystem. Indeed, system version information is not even required in order to calculate total pay. This fact results from a relationship between the initial system's method of calculating commissions and overtime pay, and the modified system's method of calculating additional payments and total pay. This relationship was unlikely to be foreseen intuitively, and is described in detail in Appendix O[78]. The calculation of additional payments is more complex in the combined system, because the model specifically insisted that the calculation NOT be performed if the "sys" state variable indicated the initial system. This additional decision increased the complexity. Also notice that the "sys" state variable is not explicitly required by the subsystem. If the incoming value of additional payments is "not calculated" then the initial system is being simulated and the final value should also be "not calculated". The additional payments state variable is both an input and an output with respect to this subsystem.

Design decisions must be made subsystem by subsystem. The analyst needs to decide whether it is more complicated to construct a subsystem which will not require changes during maintenance, than it is to construct initial and modified subsystems. That is, if C_i, C_m and C_c are the complexities of a subsystem in the

[78] Briefly, the input states which would lead to an error in the initial system's calculation of "total_pay", if the modified system's method were used, simply cannot occur. This allows the combined system to use the same method of calculating "total_pay" for both versions of the system.

initial system and the corresponding subsystems in the modified and combined systems, the following decision rule applies[79]:

IF $C_c > C_i + C_m$ THEN

construct a new subsystem during maintenance

ELSE

construct the combined subsystem initially

After examining the decompositions of the three systems the following design decisions might be made:

1. The complexities of the commissions and overtime pay subsystems in the combined system are greater than the sums of their complexities in the initial and modified systems ($11.14 > 3.25 + 4.35$ and $13.05 > 3.90 + 4.97$). Therefore, it will be simpler to reconstruct these subsystems when the maintenance is performed than to initially construct subsystems which will not require changes.
2. Assuming the analyst is interested in knowing the value of additional payments:
   a. Additional Payments: The subsystem is simpler in the modified system than in the combined system ($8.00 < 19.02$). It is also not required in the initial system. Therefore, this subsystem should be constructed during maintenance.
   b. Total Pay: The sum of the complexities of the initial subsystem and the complexity of the subsystem in the second decomposition of the modified system is less than the complexity of the subsystem in the combined system ($12.98 > 3.90 + 3.25$). Therefore, this subsystem should be reconstructed during maintenance to take advantage of the new additional payments state variable.

---

[79] Other factors are likely to be important in determining the optimal maintenance strategy. For example, since the available implementation language primitives can affect subsystem implementation difficulty, they may also influence the selection of a maintenance strategy. Identification of other such factors is a possible subject for future research. The fact that other important factors undoubtedly exist means that the simple decision rule should not be automated. Intervention by the system designer must be allowed.

3. Assuming the analyst is NOT interested in knowing the value of additional payments:

   a. Additional Payments: This subsystem should never be constructed.

   b. Total Pay: The subsystem in the combined system is less complex than the sum of the initial subsystem's complexity and the complexity of the total pay and additional payments subsystems in the modified system (12.98 < 3.25 + 8.00 + 3.90). Therefore, the combined subsystem should be constructed initially.

It should be noted that the above discussion applies to single maintenance operations only. In reality, a system is likely to undergo a series of such operations before it is finally discarded. This scenario might be diagrammed as follows:

$$\sigma_1 \xrightarrow{m_1} \sigma_2 \xrightarrow{m_2} \sigma_3 \xrightarrow{m_3} \ldots \xrightarrow{m_{n-1}} \sigma_n$$

where σ_m is the mth version of the system resulting from the m-1th maintenance operation. It is possible that σ_3 might be more easily constructed by modifying σ_1 than by modifying σ_2. In this case, the design chosen for σ_1 would be affected by m_1 and m_2, but the design for σ_2 would not be influenced by m_2. In general, the problem of finding an optimal set of system designs and changes could be quite complex.

4.5. The System Goal

As shown in Chapter 3, the same system model can have several alternative decompositions. However, not all of them may be equally acceptable to the analyst in that state variable emergence and hiding varies between alternatives. In this section, this notion is formalized through the concept of a SYSTEM GOAL.

The existence of a goal is one of the distinguishing features of an artificial system (Simon, 1981, p. 8)[80]. A system designer creates a system to fulfil some goal. This is the raison d'être for artificial systems. Definition

[80] According to Simon, artificial systems are deliberately created by man. They are qualitatively different from systems resulting from "natural" forces (eg. biological systems formed through natural selection).

of the system goal is a very important part of systems analysis and design. The system's goal, as perceived by the analyst or envisioned by the designer, influences level of abstraction and selection of boundaries for the system model[81]. The notion of a system goal is readily supported by SELMA. When an analyst creates a system model, he or she is likely to define two different kinds of state variables. Some variables will be indispensable, others will be dispensable. Indispensable state variables represent the essential properties of the system (e.g. in the modified payroll system "total pay" is likely to be indispensable). Such state variables will be examined by the user or by other systems which refer to the system being modelled. Dispensable state variables are defined by the analyst to simplify creation of the system model (e.g. in the modified payroll system, "additional payments" may have been added merely to facilitate definition of the sublaw describing the computation of "total pay"). The indispensable state variables define the purpose or goal[82] of the system as perceived by the analyst.

Definition: Goal State Variable

> Any state variable which the analyst requires to be included in a decomposition is called a GOAL STATE VARIABLE.

---

[81] A system model consists of state variables and values, external events, and sublaws. The state variables selected for inclusion in a model will be determined by the system goal. For example, a model created to analyze or describe the financial efficiency of a point-of-sales terminal system is unlikely to include state variables representing the work schedule of the terminal operator. These state variables are probably irrelevant with respect to the stated goal. Similarly, state variables describing the operation of the individual electronic and mechanical components of the terminal will not be included. Thus the system goal influences both the system boundaries and level of abstraction found in the system model.

[82] An analyst may find it more convenient to visualize a system as having a set of subgoals. However, definition of subgoals would imply that some form of decomposition has already been performed by the analyst. To avoid prejudicing the operation of the specification analysis tools, the analyst is asked to provide only the highest-level goal of the system. If the system has more than one high-level goal, the sets of state variables describing these goals must be merged.

Definition: System Goal

The set of all the goal state variables of a system is called the SYSTEM GOAL.

Notice that, as defined here, a goal is not an inherent property of a system. Rather, a goal is a function of both a system and the analyst's expectations for that system[83].

The definition of goal state variables can influence system decomposition and hence system design. Sometimes, if a state variable is not part of the system goal, subsystems which determine its value may be dropped from a decomposition. If a subsystem is dropped, the complexity of the decomposition will be reduced.

For example, consider the suggested decompositions of the modified payroll system listed in Appendix K. There are forty-eight decompositions, all satisfying the heuristics defined in Chapter 3. The intuitive decomposition is #27. All other decompositions are seemingly trivial transformations of decomposition #27. The transformations being simple substitutions. For example, consider decompositions #27 and #1 (both are shown in diagrammatic form in Figure 18).

Decomposition #27:

3: {add_pay,base,total_pay}

2: {com,emp_t,over,add_pay}

1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #1

2: {com,emp_t,over,add_pay} {base,com,emp_t,over,total_pay}

1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

where

---

[83] The specifications analysis tools allow the analyst to include a predicate of the form: system_goal(SVList) where SVList is a list of state variables which must be included in all suggested decompositions. If no goal state variables are defined, it is assumed that all state variables are indispensable.

emp_t	=	employee type (sales or office)
emp_p	=	employee position (management or regular)
hours	=	hours worked
sales	=	sales
over	=	overtime pay
com	=	commissions
add_pay	=	additional payments
total_pay	=	total pay

There are two basic differences between these decompositions:

1. the subsystems responsible for calculating the final value of total pay utilize different input and constant state variables, and
2. the output state variables at the top level are different.

In the total pay subsystem of decomposition #1, the additional payments state variable "add_pay", has been replaced by the state variables representing the values of commissions, employee type and overtime. Since it is already known from decomposition #27 that {com,emp_t,over,add_pay} is a deterministic subsystem, this would seem to be a trivial substitution. Moreover, since increasing the number of state variables in a subsystem can never decrease that subsystem's complexity, it would seem to be a useless substitution. However, such a substitution does reduce the system's dependence on the emergent state variable "add_pay". In decomposition #1, the total pay subsystem no longer requires knowledge of the additional payments state variable. If the subsystem responsible for the calculation of the final value of "add_pay" is dropped from the decomposition, total complexity will be reduced from 33.64 to 25.64. This is still greater than the 23.91 complexity of decomposition #27. Thus, in this case, the substitution may not be useful.

In general, the total complexity of a decomposition may be reduced if the subsystems responsible for the calculation of state variables removed by substitution are dropped. Such a course of action may only be justified if the analyst is not interested in knowing the final values of the removed state

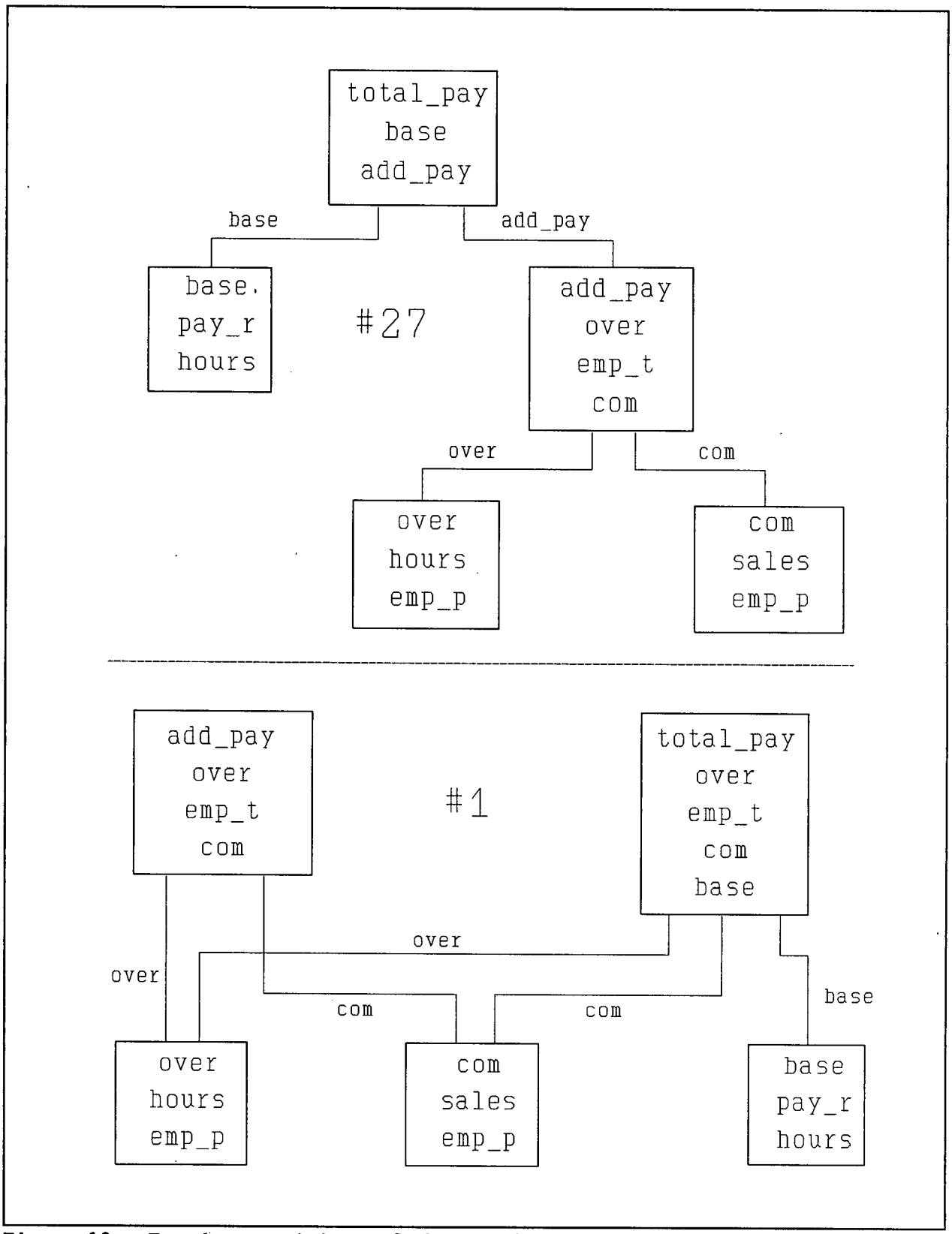


Figure 18: Two decompositions of the modified payroll system.

variables[84]. No reduction of complexity is possible in the payroll systems. However, the four-lights system does present such an opportunity. Two of the suggested decompositions for this system were as follows: Subsystem complexities are as indicated.

2: {b,c,$\underline{c}$}|2.75 {b,d,$\underline{d}$}|2.75
1: {a,$\underline{b}$}|2.00

and

1: {a,$\underline{b}$}|2.00 {a,c,$\underline{c}$}|2.75 {a,d,$\underline{d}$}|2.75

The complexity of both of these decompositions is 7.51 (There is some round-off error in the complexities of the individual subsystems). Therefore, there is no clear advantage in making substitutions for "b" in the subsystems which determine the values of "c" and "d". However, if "b" is not part of the system goal, the decomposition

1: {a,c,$\underline{c}$}|2.75 {a,d,$\underline{d}$}|2.75

with complexity 5.51 becomes a viable alternative decomposition. If complexity had not been reduced, as was the case for the overtime pay substitution in the modified payroll system, there would have been no need to suggest this alternative to the analyst. This suggests the following heuristic:

Heuristic 6: Avoid useless substitutions

Do not suggest decompositions formed by state variable substitutions unless

1. the substitution allows the removal of a subsystem where a non-goal state variable is an output, and

---

[84] That the analyst may not be interested in knowing the final values of all state variables was suggested earlier. In the previous section, different system designs were recommended depending on whether he or she was interested in knowing the final value of additional payments. Also, in Chapter 3, system views were found which "hid" different state variables depending on which state variables were of interest to the analyst.

2. after removal of the subsystem, the complexity of the decomposition has been reduced.

This means that in the case of the four light system, if "b", "c", and "d" are all specified as goal state variables, only one decomposition will be suggested to the analyst.

Decomposition #1:
2: {b,c} {b,d}
1: {a,b}

If "b" is not specified as a goal state variable an additional decomposition will be suggested.

Decomposition #2:
1: {a,c} {a,d}

4.6. Conclusions

This chapter has presented an intuitively justifiable measure of complexity. Rationalization of the measure was combined with the examination of four possible measures of complexity:

1. Ashby's Variety
2. Modified Variety
3. Shannon's Entropy
4. Hellerman's Computational Work

Hellerman's measure of computational work was finally selected for use in this research. This measure is particularly well suited for use with SELMA. Through the use of this measure, the decompositions suggested by the specifications analysis tools may be presented to the analyst in a meaningful order. It should be noted that the complexity measure can be used only to guide the search so that low-complexity decompositions are found relatively early in the search. That is, the full set of possible decompositions can be found by specifying a very

large maximum allowable percentage difference between the least and most complex decompositions.

While the measure was first suggested as a means to guide the search for decompositions so that they might be presented to the analyst is some meaningful order, it has proved itself useful in other ways as well. Its quantitative nature has supported the detailed analysis of maintenance operations. Use of the complexity measure in conjunction with the decomposition algorithm allows a system designer to select a decomposition which will reduce the total effort required for initial implementation and maintenance[85].

It was noted that the same system model can have several alternative decompositions, but not all of them may be equally acceptable to the analyst. Alternative decompositions will hide different state variables. Definition of the system goal was recognized as an important part of systems analysis and design. The goal, as perceived by an analyst, influences both the system boundaries and the level of abstraction of the system model. SELMA allows the analyst to explicitly define the system goal, so as to distinguish between indispensable state variables, which are used to define the goal, and those state variables created merely to facilitate creation of the model by simplifying the specification of sublaws. The complexity measure, coupled with system goal information, can be used to reject many alternative decompositions which would otherwise have been presented to the analyst by the specifications analysis tools.

---

[85] Future maintenance must be predictable, but need only be known to the degree of detail represented in the system model.

Chapter 5: Conditional Decomposition

5.1. Introduction

Three basic forms of decomposition were identified in Chapter 1: parallel, sequential, and conditional. Chapter 3 showed how parallel and sequential decomposition can be automated provided that the system to be decomposed has been specified using SELMA. Automation of conditional decomposition will be described in this chapter.

Recall that parallel decomposition involves identifying subsystems which behave deterministically with respect to some intermediate state space. Subsystems which are deterministic with respect to a given system relation (i.e. initial system states with their corresponding final stable states) may perform their functions at the same time (or in parallel[86]), so long as the system is in one of the states of the relevant intermediate state space. Sequential decomposition, on the other hand, was not so much a matter of identifying deterministic subsystems, but of arranging the subsystems found by parallel decomposition into a meaningful sequence of levels. This sequence had to satisfy a number of heuristics and showed how each system relation associated with a deterministic subsystem might be created. For example, consider the following decomposition of the modified payroll system.

3: {add_pay,base,total_pay}
2: {com,emp_t,over,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

where hours = hours worked
pay_r = pay rate
emp_p = employee position
emp_t = employee type
sales = amount of sales
base = base pay
com = commissions
over = over time pay
total_pay = total pay
add_pay = additional payments

This means that the subsystems {hours,pay_r,base}, {emp_p,sales,com}, and {emp_p,hours,over} at level 1 behave deterministically with respect to the first

[86] Actually, such deterministic subsystems may perform their functions in any order. Simultaneity is not required.

system relation. The first system relation results from the action of all external events on every stable state of the system. Therefore, these subsystems are deterministic with respect to all anticipated responses of the system caused by interaction with its environment. The subsystem {com,emp_t,over,add_pay} at level 2 also exhibits deterministic behaviour, but only with respect to the system relation created when the subsystems at level 1 have performed their functions. In the model these functions cause changes in the values of the output state variables of each subsystem (in this case, state variables "base", "com", and "over") in each initial state of the system relation. The end result of the actions of the subsystems at level 1 is the second system relation. The second system relation differs from the first system relation only in the values of the output state variables of the subsystems at level 1. Similar observations can be made for level 3. Parallel decomposition identifies deterministic subsystems at each level. Sequential decomposition identifies the levels themselves.

Before it may be automated, conditional decomposition must be clearly defined. The definition adopted for this research is essentially that of the "alternation statement rule" from Mili, et al. (1986). This rule is described in detail in Appendix C. Briefly, it depends on finding two relations R_1 and R_2 such that

a) $R = R_1 \cup R_2$, and

b) $domain(R_1) \cap domain(R_2) = \{\}$

where R is the system relation[87] describing the behaviour of the system. The alternation statement rule is used to decompose a program specification into two conditionally executed specifications. The programmer is required to find some predicate t(s), where t(s) is true when $s \in domain(R_1)$ and false when $s \notin domain(R_2)$, which can be used to split the original relation into two non-intersecting parts. Conditional decomposition, therefore, involves partitioning

[87] Mili et. al.'s system relations are similar to the system relations defined in Chapter 3. The domains of their relations consist of initial states of the system. However, the domains of the relations used in this research need not contain only initial states. They may contain system states where the values of some state variables have been changed by the actions of some subsystems. That is, the domains of the system relations used in this research may contain intermediate states.

a system relation into two (or more) parts. The major problem lies in deciding where to place the partitions[88].

5.2. Conditional Decomposition Basics

Before entering into an involved discussion of the partitioning problem, it may be best to consider a simple example. The system relation for a system described by four state variables ("a", "b", "sw", and "c") is shown below. The state variable sw is intended to represent the position of an SPDT (single pole/double throw) switch which makes a connection between "a" and "c" or between "b" and "c" as illustrated in Figure 19.

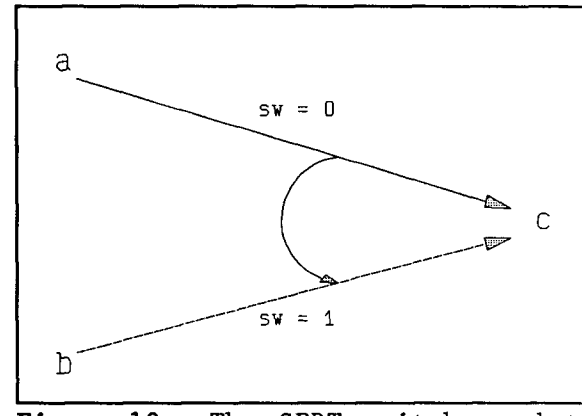


Figure 19: The SPDT switch used to illustrate conditional decomposition.

Intermediate State Space					Corresponding Final Stable States			
a	**b**	**sw**	**c**	---->	**a**	**b**	**sw**	**c**
1	-	0	-		1	-	0	1
0	-	0	-		0	-	0	0
-	1	1	-		-	1	1	1
-	0	1	-		-	0	1	0

where "-" means "don't care" or "any value". There is only one parallel/sequential decomposition of this system.

1: (a,b,sw,c)

That is, the system may not be decomposed into smaller subsystems using only parallel and sequential techniques. However, the system may be conditionally

---

[88] It is always possible to produce a trivial partitioning of the system relation by creating a dummy state variable with value "1" for some states and "0" for the rest. However, an analyst is unlikely to create such an arbitrary state variable, and the specifications analysis tools use only the state variables included in the system model in the search for possible conditional decompositions.

decomposed. If the above relation is partitioned using the conditions sw = 0 and sw = 1, the following two smaller relations result.

sw = 0:

Intermediate State Space					Corresponding Final Stable States			
a	**b**	**sw**	**c**	---->	**a**	**b**	**sw**	**c**
1	-	0	-		1	-	0	1
0	-	0	-		0	-	0	0

sw = 1:

Intermediate State Space					Corresponding Final Stable States			
a	**b**	**sw**	**c**	---->	**a**	**b**	**sw**	**c**
-	1	1	-		-	1	1	1
-	0	1	-		-	0	1	0

When the value of "sw" is 0, one deterministic subsystem satisfying all the heuristics of the previous chapter is {a,c}. State variable "b" can be any value for each value of "c" and state variable "sw" is a constant providing no information, therefore, neither is required to determine the value of "c". Similarly, when the value of "sw" is 1, {b,c} is a deterministic subsystem.

Conditional decompositions will be expressed using the following syntax:

$$[\text{CondSVs} = \text{CondVals}_1]\text{Subsystems}_1 \ \ldots \ [\text{CondSVs} = \text{CondVals}_p]\text{Subsystems}_p$$

where

CondSVs = the CONDITIONAL STATE VARIABLES, or the set of state variables which are tested to partition the system relation.

CondVals_i = a set of sets of values of the conditional state variables.

Subsystems_i = a set of subsystems[89] which are deterministic with

---

[89] This is a set of subsystems because partitioning of the system relation may allow further parallel decomposition of the system being decomposed. For example, the subsystem {i,j,k,l} might conditionally decompose to
1: [{i} = {{0}}]{{i,k},{j,l}} [{i} = {{1}}]{{j,k,l}}

respect to the part of the partition identified by the condition CondSVs = $CondVals_i$. Such subsystems will be referred to as CONDITIONAL SUBSYSTEMS.

For example, the conditional decomposition for the SPDT switch system may be expressed as follows:

[{sw} = {{0}}]{{a,c}}|2.00 [{sw} = {{1}}]{{b,c}}|2.00

The complexity of {a,b,sw,c} is 8.00. As indicated following the |, the complexities of {a,c} and {b,c} are both 2.00. Therefore, the complexity of the original system has been reduced (by a factor of 2) through conditional decomposition.

The above syntax requires several layers of bracketing. In order to improve the readability of conditional decompositions, brackets will be dropped whenever possible so long as the meaning is preserved. The conditional decomposition of the SPDT switch system can be simplified to the following:

[sw = 0]{a,c} [sw = 1]{b,c}

Conditional decompositions may also be presented diagrammatically as shown in Figure 20.

5.3. Heuristics

Conditional decomposition of the SPDT switch system was trivial. A glance at the system relation sufficed to identify "sw" as a suitable conditional state variable, and discovery of the conditional subsystems quickly followed. In most cases things will not be so simple. In fact, had the rows of the system relation been randomly rearranged, conditional decomposition of even this system would have required some effort. While simple, the example did illustrate the general procedure to be followed when conditionally decomposing a system.

---

indicating that the final values of the output state variables need not necessarily be determined together.

Conditional Decomposition Procedure:

1. Perform parallel/sequential decomposition.
2. Select a subsystem[90] for further conditional decomposition.
3. Select a state variable.
4. Partition the system relation on the basis of the values of this state variable.
5. Find the subsystems which behave deterministically with respect to each part of the partition.

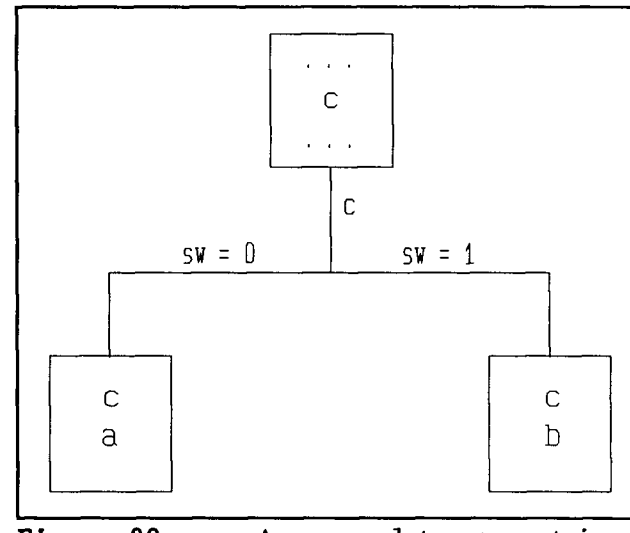


Figure 20: An alternative representation for conditional decomposition.

As was the case for parallel/sequential decomposition, given only the deterministic subsystem requirement, this procedure could lead to an extremely large number of conditional decompositions. After all, in the above example, when the value of "sw" is 1, {a}, {sw}, {a,sw,c} as well as {a,c} are all deterministic subsystems. Clearly, some heuristics to limit the search for conditional decompositions are required. The heuristics of Chapter 3 which dealt with individual subsystems are applicable here. The others were concerned with arranging subsystems in a level structure and are not useful for conditional decomposition.

Conditional Heuristic #1: Outputs Required

Each conditional subsystem must be described by at least one output state variable.

---

[90] There is no fundamental distinction between a system and a subsystem. A subsystem of a system σ_1 is a system σ_2 where the remainder of σ_1 is in the environment of σ_2. Conditional decomposition may be applied to both systems and subsystems in exactly the same fashion. If parallel/sequential decomposition is applied to a system which may not be decomposed either in parallel or sequentially, only one deterministic subsystem will be found (as happened in the case of the SPDT switch system). This subsystem will be equal to the system.

The rationale for this heuristic is the same as for parallel/sequential decomposition. Basically, subsystems without outputs are not very interesting.

Conditional Heuristic #2: Must be Small

> Each conditional subsystem must not be described by any state variable which is not required to ensure deterministic behaviour.

Again, the rationale for this heuristic is the same as for parallel/sequential decomposition. An analyst is interested in knowing the minimal amount of information (in the form of state variable values) necessary to perform some task.

Conditional Heuristic #3: Must be Different

> The set of state variables describing conditional subsystems must differ by at least one state variable.

Consider the following conditional decomposition. Some redundant brackets have been removed for clarity.

[h = 0]{j,k} [h = 1]{j,k}

Such a structure does not provide any information beyond the fact that {j,k} is deterministic with respect to the entire first system relation[91]. This could be more succinctly represented using the simpler parallel/sequential syntax.

1: {j,k}

While it is important to know what conditional decomposition is, it is equally important to know what it is not. This heuristic implies that some conditional decompositions, which would be considered by Mili et al. using their alternation statement rule, will not be considered here. Conditional

---

[91] The Mili et. al. requirement that $R = R_1 \cup R_2$ implies that "h" has no values other than "0" and "1".

decomposition will not find alternative functional forms for the same subsystem. For example, suppose that total pay (total_pay) is a function of hours worked (hours) and the pay rate (pay_r). Also suppose that the employee receives 1.5 times his or her regular pay for each hour in excess of 40. Such a situation is easily coded using an IF/THEN/ELSE structure.

```
IF hours≤40 THEN total_pay := hours*pay_r
            ELSE total_pay := pay_r*(1.5*hours-20);
```

This use of an IF/THEN/ELSE structure is not the sort of conditional decomposition being described here. The subsystem describing both the THEN and ELSE parts of the structure is {hours,pay_r,total_pay}. Therefore, a partition of the system relation using hours worked as the conditional state variable would be rejected because of Conditional Heuristic #3. This is an important difference between the sort of decomposition embodied by Mili's alternation statement rule and conditional decomposition. Mili et al. do not explicitly consider state variables in their decompositions. They look only at the system relation resulting from a partition. They would see a partition using the rule hours ≤ 40 as useful because it allows different program implementations for the THEN and ELSE portions of the structure. Whether a partition allows different program implementations is determined by the primitives available in a given language. If a language primitive to calculate total pay directly from any values of hours worked and pay rate existed, partitioning on the basis of hours worked would not lead to different implementations, and Mili et al. would not consider such a partition useful. As argued in the previous chapter, this research is not concerned with available language primitives, and as such is primarily useful at a fairly high level of analysis.

Conditional Heuristic #4: Same Conditional State Variables

> The conditional state variables associated with each conditional subsystem must be the same.

This heuristic helps to ensure that there is no overlap between the parts of the system relation associated with each conditional subsystem. Consider the following:

$[h = 0]\{j,\underline{k}\}$ $[i = 0]\{i,\underline{k}\}$

There is no reason why such a decomposition should be rejected. As long as "i" cannot be 0 when "h" is 0 and vice versa, this decomposition will not violate the Mili et al. condition of non-intersecting domains. However, in this case the decomposition could be replaced by

$[h = 0]\{j,\underline{k}\}$ $[h \neq 0]\{i,\underline{k}\}$.

The non-intersecting nature of such a decomposition is far more apparent, and is preferred.

Conditional Heuristic #6: Complexity may not Increase

> The total complexity of the conditional subsystems may not exceed the complexity of the subsystem being decomposed.

There is no point in suggesting a conditional decomposition which is more difficult to understand or build than the original system. An example of a conditional decomposition which increases the complexity of the system is given near the end of this chapter.

The next heuristic cannot be intuitively justified. It is introduced solely to keep the problem of conditional decomposition computationally tractable.

Conditional Heuristic #7: Single Conditional State Variables

> The set of conditional state variables used to partition a system relation may have no more than one member.

This means that conditional decompositions such as

$[\{x,y\} = \{\{0,0\},\{1,1\}\}]\{j,\underline{k}\}$ $[\{x,y\} = \{\{0,1\},\{1,0\}\}]\{i,\underline{k}\}$

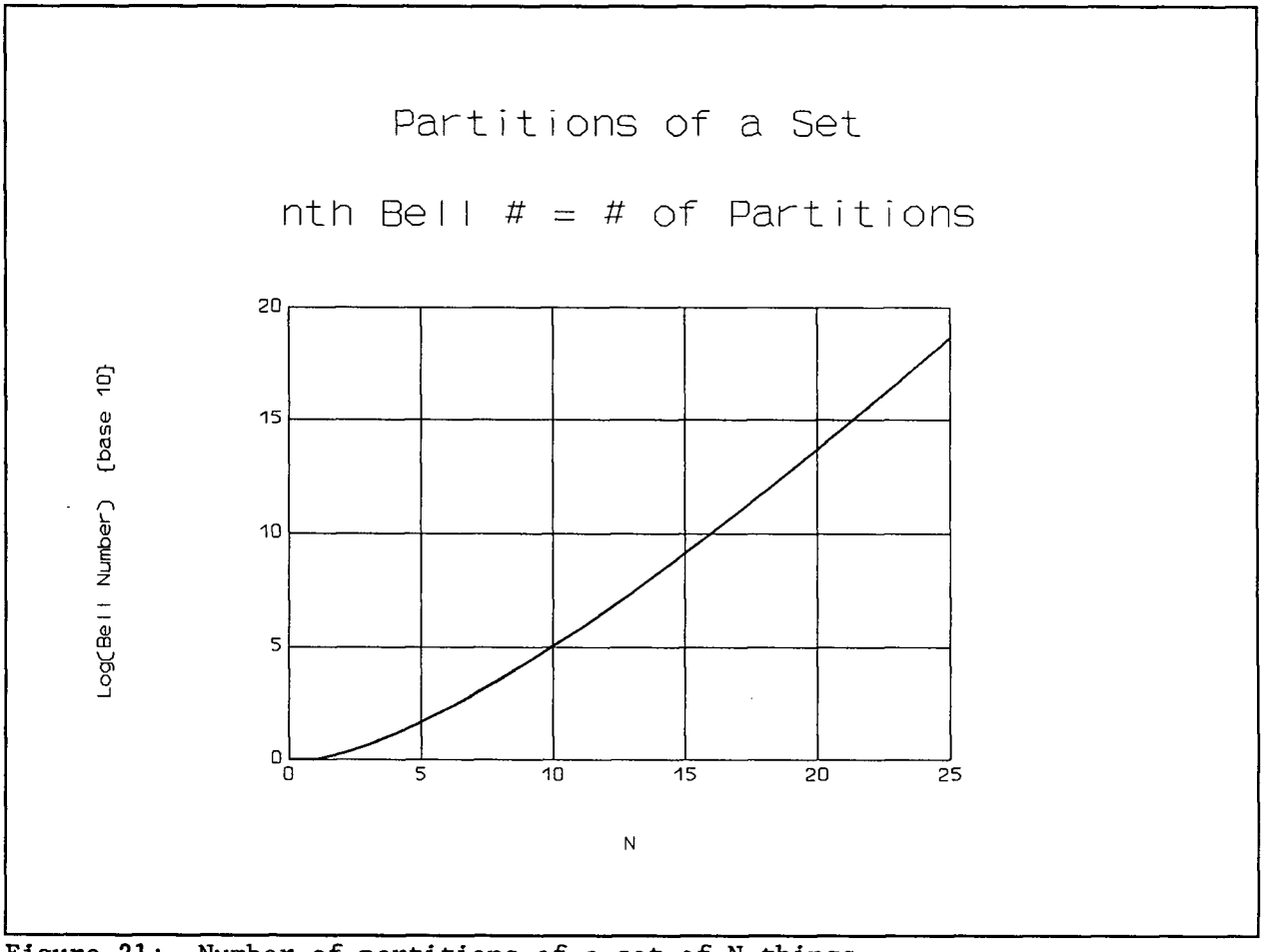


Figure 21: Number of partitions of a set of N things.

will not be considered. The major computational problem with conditional decomposition lies in testing all possible partitions of the system relation with respect to the values of the conditional state variables. The number of partitions of a set[92] increases dramatically as the number of elements in the set increases (see Figure 21). Experience gained during this research has shown that most systems may be described using state variables with between two and five

---

[92] The number of partitions of a set containing N elements where all N elements occur in one and only one part (also called a "class" or "block") is called the "Nth Bell number". Bell numbers are given by the following recurrence relation (Krishnamurthy, 1986, pp. 16 and 22).

$$B(0) = 1$$

$$B(N+1) = \sum_{K=0}^{N} C_{N,K} * B(K)$$

where $C_{N,K}$ is the number of combinations of N things taken K at a time.

values[93]. A state variable with five values leads to only 52 partitions of the system relation. This number of partitions can be easily examined for subsystems meeting the other heuristics. On the other hand, if more than one conditional state variable is allowed, the number of partitions quickly becomes unmanageable. As shown below, conditional decomposition of a trivial system described by only three state variables with two values each would require testing of 4184 partitions.

State Variable	Values
a	0,1
b	0,1
c	0,1

Conditional State Variables	Partitioning Values	Partitions to Test[94]
a	0,1	1
b	0,1	1
c	0,1	1
ab	{0,0},{0,1},{1,0},{1,1}	14
ac	{0,0},{0,1},{1,0},{1,1}	14
bc	{0,0},{0,1},{1,0},{1,1}	14
abc	{0,0,0},{0,0,1},{0,1,0},{0,1,1}, {1,0,0},{1,0,1},{1,1,0},{1,1,1}	4139
	total partitions to test:	4184

note: If certain combinations of state variable values can never occur together, the total number of partitions to test can be reduced. For

---

[93] Recall that continuous real-world variables are modelled using ranges. The system responds to all values in a given range in similar ways.

[94] The number of partitions which must be tested is one less than the number of possible partitions. The partition which consists of only one part, where that part is the set to be partitioned need not be tested. Since the system relation would not be split in this case, examining such a partition would be equivalent to testing whether the original system is deterministic. It is assumed that all systems to which conditional decomposition is to be applied are already known to be deterministic.

example, if the system relation contains no state where a=0 and b=0, the total number of partitions to test drops to 1+1+1+4+14+14+202 = 237. Thus, the number of partitions shown in the above table is an upper limit only.

The effect of restricting partitions to those involving only one conditional state variable is not as serious as might be first imagined. The analyst can always decide to further conditionally decompose a conditional subsystem. The effect of this is essentially the same as partitioning with more than one conditional state variable. For example, suppose the following was suggested as a conditional decomposition of a system described by state variables h, i, j, and k:

$[h = 0]\{i,j,\underline{k}\}$ $[h = 1]\{i,\underline{k}\}$

Now suppose the analyst suspects that the first conditional subsystem can be further decomposed. The conditional decomposition procedure can be applied again to the subsystem $\{i,j,\underline{k}\}$. This might result in the decomposition

$[i = 0]\{i,j,\underline{k}\}$ $[i = 1]\{j,\underline{k}\}$.

The two levels of conditional decomposition can be combined as follows:

$[\{h,i\} = \{0,0\}]\{i,j,\underline{k}\}$ $[\{h,i\} = \{0,1\}]\{j,\underline{k}\}$ $[h = 1]\{i,\underline{k}\}$

In this case, full search of all possible partitions has been replaced by selective search guided by the judgement of the analyst.

5.4. Using Conditional Decomposition to Test a Model

None of the three major examples developed so far (namely the four light, the payroll and the modified payroll systems) offer any subsystems which are obvious candidates for conditional decomposition. However, the conditional decomposition procedure can help to uncover some modelling errors. This will be demonstrated using the modified payroll system. The error discovered in the modified payroll system is symptomatic of one potential problem with using

qualitative modelling techniques (i.e. state variable ranges) for state variables which represent continuous quantities in the real world.

Recall the description of the modified payroll system presented in Chapter 2:

1. both office staff and sales employees are entitled to both overtime pay and sales commissions,
2. an office employee cannot receive more in commissions than in overtime, and
3. a sales employee cannot receive more in overtime than in commissions.

The specification analysis tools identified the following subsystem capable of determining the value of the additional payments state variable. This state variable was introduced to represent the total overtime and commission pay to which an employee is entitled after these rules have been applied.

2: {com,emp_t,over,add_pay}|8.00

where com = commissions
over = overtime pay
emp_t = employee type (o = office worker, s = sales employee)
add_pay = additional payments

When conditional decomposition is applied to this subsystem an unreasonable suggestion is made by the specifications analysis tools:

[emp_t = o]{over,add_pay}|2.00 [emp_t = s]{com,add_pay}|2.00

In order to calculate additional payments, the amount of commissions, overtime pay and the type of the employee must be available. There is no way that additional payments can be calculated given only the employee type and the amount of overtime pay. Why then, is the above conditional decomposition suggested? The functional form of this subsystem may be represented by the following table:

com	emp_t	over	--->	add_pay
nz	o	nz		nz
0	o	nz		nz
nz	s	nz		nz
nz	s	0		nz
nz	o	0		0
0	o	0		0
0	s	nz		0
0	s	0		0

where 0 = a value of zero
nz = some non-zero value

Notice that the additional payments state variable is modelled with only two values: 0 and non-zero. It is possible to predict whether additional payments is going to be zero or non-zero given only the employee type and either the amount of commissions or overtime pay. That is, if the employee is an office worker and his or her overtime pay is non-zero, then additional payments will be non-zero. On the other hand, if his or her overtime pay is zero, then additional payments will also be zero, since he or she may not make more in commissions than in overtime. A similar argument applies for members of the sales staff. The above table may be rewritten to make this relationship obvious.

com	emp_t	over	--->	add_pay
-	o	nz		nz
-	o	0		0
nz	s	-		nz
0	s	-		0

where "-" = any value or "don't care"

The problem lies in the choice of values for the additional payments state variable. The rules specified in the system model for the calculation of additional payments may be represented in tabular form as shown:

com	emp_t	over	--->	add_pay
nz	-	nz		nz
0	-	0		0
-	o	nz		nz
-	o	0		0
nz	s	-		nz
0	s	-		0

These rules are not concerned merely with determining whether the value of additional payments is zero or non-zero. The rules specify the conditions under which the total of commissions and overtime pay is to be reduced because of the employee's position. For example, if the employee is part of the sales staff and he or she potentially makes more in overtime pay than in commissions, not all of the overtime should actually be paid. This concept of "pay reduction" should be made explicit in the values of the additional payments state variable. The rules could be rewritten as follows:

com	emp_t	over	--->	add_pay
nz	-	nz		nr
0	-	0		0
0	o	nz		nr
nz	o	0		r
nz	s	0		nr
0	s	nz		r

where r = pay has been reduced
nr = no reduction in pay

When parallel/sequential decomposition is applied after such a change, the decompositions are structurally the same as for the original system. The only difference is that the complexity of the additional payments subsystems has been increased to 12.00. However, when conditional decomposition is applied to the additional payments subsystem, the offending conditional decomposition is no longer suggested.

The changed subsystem can also be used to illustrate the need for the conditional decomposition heuristic, which required that total complexity not be increased by conditional decomposition. If total complexity is allowed to increase, the following is suggested by the specifications analysis tools for the changed system:

[add_pay = {0,nr}]{com,emp_t,over,add_pay}|12.00
[add_pay = {r}]{com,over,add_pay}|6.00

That is, if the system relation is split using the initial values of the additional payments state variable, in all cases where the additional pay was previously[95] reduced, only knowledge of commissions and overtime is required to determine the new value of additional payments. The complexities of these two conditionally-activated subsystems are 12.00 and 6.00 respectively. The complexity of the original additional payments subsystem was 12.00. Conditional decomposition increased the overall complexity of the subsystem. Complexity is increased because the additional payments state variable is now an input to the conditionally-decomposed system as well as an output. Since the number of input state variables has increased, so has the number of input states and, therefore, so has the complexity.

Although the above decomposition increases complexity and can, therefore, be rejected, it is worth examining how it could even be a possibility. How can the value of additional payments be calculated without knowing the position of the employee? This decomposition results from the fact that each external event defined in the model changes the value of only one state variable. There are external events which alter the value of the hours worked state variables, and external events which alter the value of the sales state variable, but there are no external events which alter both together. This means that whenever the additional payments subsystem is activated, either the values of commissions or overtime pay is equal to its previous value. This, along with the old dollar value of additional payments and knowledge that pay was previously reduced, is sufficient information to calculate the new value of additional payments. For example, suppose the old values of commissions, overtime pay, and additional payments were as follows:

old commissions	=	$400
old overtime pay	=	$500
old additional payments	=	$800 and pay reduced

[95] That is, before the external event which led to a new value for either "com" or "over".

The employee was obviously a member of the sales staff since additional payments is less than the sum of sales commissions and overtime pay, and overtime pay exceeds sales commissions. Now suppose, an external event alters the value of sales[96] such that commissions are increased to $600. The additional payments subsystem now has access to the following information:

new commissions	=	$600
old overtime pay	=	$500
old additional payments	=	$800 and pay reduced

Since the old value of additional payments is less than twice the old value of overtime pay and pay was reduced, the employee is a salesperson. Therefore, the new value of additional payments should be $1,100 with no pay reduction. If the system model is changed so that both hours worked and amount of sales can change at the same time (as would likely be the case in a batch processing system, where transaction records contained information about both hours and sales), this conditional decomposition will not be suggested. Once again, the sensitivity of decomposition to the defined external events is demonstrated.

5.5. Conclusions

Three basic forms of decomposition were identified in Chapter 1: parallel, sequential, and conditional. Parallel and sequential decomposition were discussed in Chapter 3. This chapter has investigated the remaining basic form of decomposition: conditional decomposition. While the basics of conditional decomposition are adapted from the alternation statement rule of Mili et al. (1986), procedures for actually decomposing a system are original to this research. Two types of conditional decomposition were identified. One type led to different functional forms for a calculation using the same state variables. The other found subsystems described by different sets of state variables. The first type was seen to be primarily useful during the implementation phase of the system development life cycle. While Mili et al. were concerned with both types, the specifications analysis tools deal only with the second.

---

[96] This might happen, for example, if a correction to the amount of sales were entered within the same pay period.

A number of heuristics to limit the search for suitable conditional decompositions have been suggested. Two of these heuristics are derived from those suggested for parallel decomposition, one is justified on the basis of complexity, and another is suggested for reasons of computational efficiency. The remainder follow directly from the meaning of conditional decomposition.

The modified payroll system of Chapter 3 was reexamined, and conditional decomposition was shown to be a useful tool for finding inadequacies in a system model. While the small systems used as examples cannot illustrate useful conditional decomposition, the IFIP system, analyzed in the next chapter, can.

Chapter 6: SELMA Applied

6.1. General

This chapter is intended to show the feasibility of applying the SELMA formalism and specifications analysis tools to a "real" system. The validity of the modelling approach will be assessed by comparing the results to those obtained by using more established systems analysis and design methodologies. In order to do this, each technique must be applied to the same system. Fortunately, there exists a system which has been analyzed by a large number of methodologies. This is the IFIP Working Conference system.

In 1982, the International Federation for Information Processing (IFIP) held a conference intended to provide a comparative review of a number of information system design methodologies. In order to facilitate comparison, a single test case was provided. The proponents of each methodology then produced a specification for an information system designed to solve the problem presented in the case. The problem was to design an information system to support an IFIP Working Group Conference. The information system was to support several activities of the Program Committee and the Organizing Committee (Olle, 1982, pp. 8-9). The case is described with greater detail in Appendix P.

Activities of the Program Committee to be supported:

1. Preparing a list to whom the call for papers is to be sent.
2. Registering the letters of intent received in response to the call.
3. Registering the contributed papers on receipt.
4. Distributing the papers among those undertaking the refereeing.
5. Collecting the referees' reports and selecting the papers for inclusion in the program.
6. Grouping selected papers into sessions for presentation and selecting a chairman for each session.

Activities of the Organizing Committee to be supported:

1. Preparing a list of people to invite to the Conference.
2. Issuing priority invitations to National Representatives, Working Group members and members of associated working groups.
3. Ensuring all authors of each selected paper receive an invitation.

4. Ensuring authors of rejected papers receive an invitation.
5. Avoiding sending duplicate invitations to any individual.
6. Registering acceptance of invitations.
7. Generating a final list of attendees.

SELMA has been applied to the IFIP Working Conference system. The application technique is comprised of five major steps:

Step 1: State variable identification
Step 2: External event identification
Step 3: Sublaw identification
Step 4: Consistency and completeness testing
Step 5: Decomposition

Full application of these steps to the IFIP Working Conference system is far too lengthy to be demonstrated in this chapter. The reader would be overcome by details. The identification of three state variables, one external event, and one sublaw will be described here. These examples were selected to show some interesting aspects of SELMA and to suggest the flavour of its application to a real system. Construction of the entire model is described in Appendices Q, R, and S.

The specifications analysis tools were used to verify the consistency and completeness of the resulting system model. Many errors were made during the construction of the model; however, only one consistency and one completeness error will be described in detail. The intent is to illustrate to the reader the process by which a complete and consistent model may be constructed, but not to overwhelm him or her with details. The errors also illustrate the usefulness of the specifications analysis tools for ensuring model integrity.

The specifications analysis tools suggest three decompositions for the IFIP Working Conference system. As will be discussed later, the differences result from the limited amount of system information incorporated in the model. One decomposition will be selected for comparison to the decompositions produced by Jackson System Development (JSD) (McNeile, 1982, pp. 225-246) and Active and Passive Component Modelling (ACM/PCM) (Brodie and Silva, 1982, pp. 41-91). JSD and ACM/PCM were selected for comparison with SELMA for a number of reasons:

1. Both JSD and ACM/PCM have been used to solve the Working Conference problem.
2. JSD is notable for its explicit focus on real-world modelling and simulation. In particular it provides some guidelines for the selection of suitable "communicating sequential processes" or entities. These entities are similar to the objects of Object-Oriented Programming. As will be described later in this chapter, SELMA decompositions may be used to identify objects. It will be interesting to see how closely the objects automatically identified by the specifications analysis tools match those identified by JSD.
3. ACM/PCM carefully distinguishes between static and dynamic system modelling. This separation of static and dynamic behaviour, or of data and programs, is common to many methodologies. SELMA makes no such distinction. It will be argued that the separation of system statics and dynamics is not only unnecessary, but may even lead to specification errors.
4. ACM/PCM is typical of many of the system development techniques which depend on object hierarchies representing "is-a" and "part-of" relationships.
5. ACM/PCM uses condition and action statements to describe dynamic behaviour. These statements are similar to sublaws.

It should be clarified from the outset that SELMA is <u>not</u> intended as a replacement for either JSD or ACM/PCM. Both JSD and ACM/PCM support detailed system design down to the implementation level. The SELMA methodology does not do this. SELMA is intended for use at a relatively high level of abstraction during the real-world modelling phase of the systems analysis and design process. When used with the specifications analysis tools, SELMA can provide automated system verification and decomposition. In ACM/PCM the system's decomposition is a function of the objects selected for inclusion in the specification. No advice is given on how to make the selections. JSD provides a number of rules to aid in object (or entity) identification, but they would be very difficult to automate. These rules will be examined later in this chapter. It will be shown that the subsystems suggested by the specifications analysis tools can be used to form objects which will satisfy all of the JSD rules. Thus SELMA is

seen as a possible addition to existing systems analysis and design methodologies, rather than as a methodology in itself.

6.2. Applying SELMA

The five major steps for applying SELMA may be diagrammed as in Figure 22.

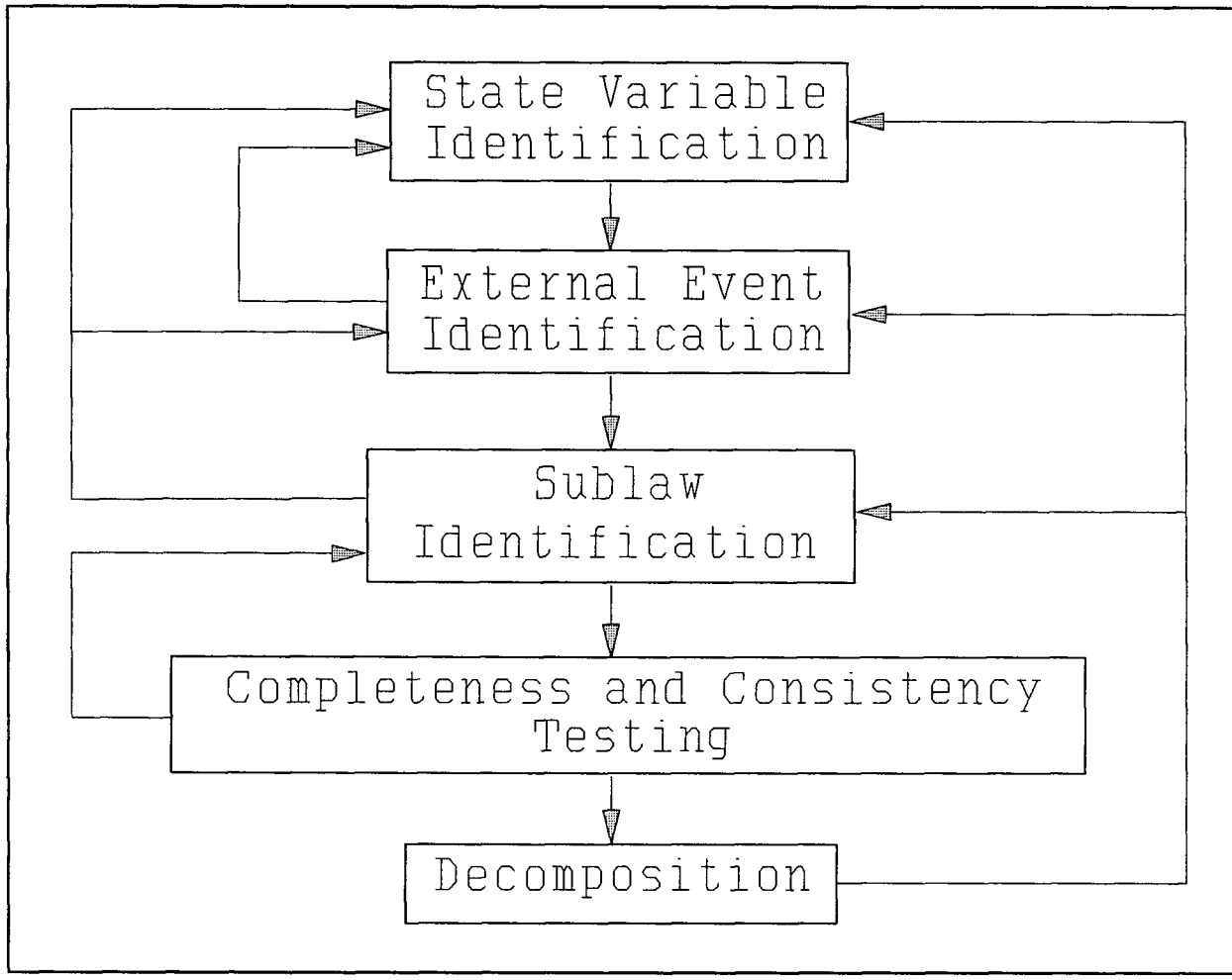


Figure 22: Block diagram of the States, Events, and Laws Modelling Approach (SELMA).

Note that these steps need not be performed sequentially. That is, it is quite likely that while an analyst is identifying sublaws, he or she may decide that another state variable is required or that an external event has been missed. Also, should the model fail the tests for local consistency and completeness, changes to state variables, external events, and/or sublaws will be required.

Finally, as was illustrated in the last chapter[97], if the decompositions suggested by the tools are not considered reasonable by the analyst, changes to the model may be required. A brief description of each step is provided below. Detailed examples of the activities performed during each step will follow. Construction of the IFIP Working Conference model[98] is described in full in Appendices Q, R, and S.

Step 1: State variable identification

State variable identification is accomplished through a detailed examination of the system functions (these correspond to the activities listed above and in Appendix P for the IFIP Working Conference Problem). The system functions (or requirements) are combined with the analyst's knowledge of system behaviour to identify those properties which should be represented in the information system[99].

Step 2: External event identification

External events are found by examining each state variable identified in Step 1, and deciding whether its value is determined by the system itself or the environment. External events are defined for each state variable directly affected by the environment.

---

[97] The specifications analysis tools suggested a conditional decomposition of the additional payments subsystem of the modified payroll system which conflicted with reality. Changes to the sublaw describing the calculation of additional payments were required.

[98] In order to illustrate the utility of tests for completeness and consistency, the model constructed in Appendices Q, R, and S contains several errors.

[99] Strictly speaking, SELMA state variables represent properties of the system. They do not have to be tied to any particular things (ie. objects or entities) in the real world. However, it is likely to be difficult for most analysts to visualize a property of the system, as opposed to a property of some thing. There is no harm in visualizing a system as consisting of some set of things before deciding on relevant properties. For example, when analyzing the IFIP Working Conference problem, an analyst may wish to visualize people and papers before deciding on specific properties such as Group membership or paper quality. But it must be remembered that these things are merely a first approximation to a decomposition of the system. The specifications analysis tools will identify deterministic groups of state variables (ie. subsystems) which can provide the basis for identifying a system's things. There is no reason to expect that an analyst's initial list of things will always be the same as that derived from use of the tools.

Step 3: Sublaw identification

Every state variable not directly affected by the environment will be included in at least one sublaw. The analyst consults his or her knowledge of system dynamics to construct rules describing the relationships between state variables.

Step 4: Consistency and completeness testing

The specifications analysis tools are used to automatically test the model for local completeness and consistency. Operation of the system is simulated to ensure that each stable state, when acted on by an external event, can be transformed to one and only one stable state by the defined sublaws. Local completeness and consistency were formally defined in Chapter 2.

Step 5: Decomposition

All three forms of decomposition are automatically performed by the specifications analysis tools. Parallel sequential decomposition is used to find sets of deterministic subsystems and the time ordering of their activation. For example, parallel sequential decomposition of the modified payroll system yielded the following:

3: {base,add_pay,total_pay}
2: {com,emp_t,over,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

This decomposition indicated, among other things, that calculations for base pay ("base"), sales commission ("com") and overtime pay ("over") may be performed in parallel, and that they must be performed before total pay may be determined. Conditional decomposition provides additional flexibility in the time ordering of subsystem activations. For example, suppose that additional payments ("add_pay") were only calculated for office employees ("emp_t" = "office"). The tools would suggest the following conditional decomposition of the additional payments subsystem.

[emp_t = office]{com,over,add_pay} [emp_t = sales]{add_pay}

This indicates that, in the case of sales employees, additional payments may be calculated immediately (it will be zero). There is no need to wait for sales commission and overtime pay to be determined. All forms of decomposition will help the analyst to identify modelling errors should suggestions conflict with his or her understanding of the system.

The five steps of the SELMA methodology will now be applied to the IFIP Working Conference Problem.

6.2.1. State Variable Identification

The first stage in the process of information systems analysis and design involves building a model of the real world (see Figure 6). Naturally, no analyst would attempt to model everything in the real world. He or she will only model those parts which are to be reflected in the implemented information system. To identify these parts, the analyst must determine the functionality of the system. That is: what is the information system supposed to provide? In SELMA, the portion of the real world to be modelled is delineated by the state variables chosen to represent those properties of the real system required to support the functions to be provided by the information system. The IFIP Working Conference information system is required to support a number of activities. These were listed earlier in this chapter and are repeated in Appendix P. Consider the first activity of the Programme Committee.

Activity: Preparing a list to whom the call for papers is to be sent.

This activity suggests that one property of the real world, with which the information system will be concerned, should indicate whether a particular person is to be invited to submit a paper to the Conference. This property, or state variable, will be called "pap_inv" (for "invited paper"). Invitations to submit papers are always sent to National Representatives, Working Group members, and members of associated working groups. A state variable indicating whether a person is in <u>any</u> of these categories will be called "grp_mem" (for "group member"). Individuals in each category are treated the same with respect to all of the activities which the information system is to support. Therefore, to

avoid unnecessary complexity only one state variable is used. Individuals not in any of the above categories could also be invited to submit a paper. The state variable "ext_inv" (for "external invitation") will be used to indicate whether this is the case. Each of these state variables will have two values "y" and "n" (for "yes" and "no") to indicate whether a person has been invited to submit a paper, is a group member, or will be invited to submit a paper regardless of group membership. Notice that state variables describing the list itself are not properly a part of the system being modelled[100]. The list is an artifact of the implemented information system and need not be included in model of the real world.

Many more state variables were identified by examining the other required system functions. These are described in Appendix Q. A list of the IFIP Working Conference state variables will be provided after the identification of an external event is illustrated.

6.2.2. External Event Identification

In SELMA, external events affect a system by altering the values of state variables. The values of other state variables may be changed by the system itself in response to an external event. Such secondary changes are called internal events. During this step, the analyst is primarily concerned with external events. Internal events are considered when system sublaws are defined. Each of the above state variables must be examined to decide whether its value is set by an external event. One will be examined here. The others are considered in Appendix R.

The state variable "del_acc", as identified in Appendix Q, is used to represent whether a delegate has accepted an invitation to attend the Conference. Whether a person accepts an invitation is beyond the influence of the system. Therefore, acceptance of an invitation must be modelled using external events. However, the state variable "del_acc" was to be used to generate a list of conference attendees. This implies that attendance at the Conference is entirely decided by factors external to the system. This is not the case. Mere acceptance of an invitation is not a sufficient condition for registration at

---

[100] For example, state variables describing the list itself might include list currency or length. If some activities of the Committees required these properties, they would have to be included in the model.

the Conference. The delegate must also have been invited[101]. A state variable, in addition to those identified in Appendix Q, is required to indicate whether the delegate has actually been registered. This state variable will be called "del_reg" (for "registered delegate" and will have the values "y" and "n" (for "yes", the delegate is registered and "no", he or she is not). The value of "del_reg" is not directly affected by external events, but is determined solely by the values of "inv" and "del_acc". Also note, the activity "generating a final list of attendees" will require the examination of the state variable "del_reg", instead of "del_acc" as suggested in Appendix Q.

The state variables identified through examination of required system functions and determination of external events are listed below. The defined values have the following meanings.

y = yes
n = no
acc = accept
rej = reject
n/a = not applicable

State Variable Name	Values	Description
grp_mem	y,n	Whether a person is a member of the Working Group.
ext_inv	y,n	Whether an invitation to submit a paper should have been issued to a person by the Programme Committee regardless of Group membership.
pap_prom	y,n	Whether a person has promised to submit a paper to the Working Conference.
pap_sub	y,n	Whether a person has submitted a paper for review to the Working Conference.
ret_ref	y,n	Whether a paper has been returned to the Programme Committee by the referees.

[101] It is conceivable that some person might return an invitation which was not sent to him. Perhaps it was obtained from a colleague. Details like this one were not included in the "first draft" model of the IFIP Working Conference system. They were added in order to make the model complete and consistent. For clarity, not all errors made in the "first draft" are described here.

suit	y,n	Whether a paper is suitable for inclusion in the Conference
chair	y,n	Whether a chairman has been assigned to a session by the Programme Committee.
del_acc	y,n	Whether a person has accepted an invitation from the Organizing Committee to attend the Conference.
pap_inv	y,n	Whether a person has been invited to submit a paper to the Programme Committee for consideration.
sent_ref	y,n	Whether a paper has been sent to the referees by the Programme Committee.
ref_dec	acc,rej,n/a	The referee's decision as to the suitability of a paper for inclusion in the conference.
pap_dec	acc,rej,n/a	The Programme Committee's decision as to the suitability of a paper for inclusion in the conference.
sess_ass	y,n	Whether a paper has been assigned to a session by the Programme Committee.
inv	y,n	Whether a person has been invited to attend the Conference by the Organizing Committee.
del_reg	y,n	Whether the person has been registered to attend the conference.

6.2.3. Sublaw Identification

The easiest way to define sublaws is to consider each state variable individually. The sublaws involving all of the state variables listed above are developed in Appendix S. Only the sublaw governing the decision to invite a person to attend the conference will be considered here. A person will be invited if one of the following conditions is met:

1. He or she is a Working Group member.

2. He or she has submitted a paper that has been accepted, rejected, or not yet returned by the referees[102].

Furthermore, no person should be invited twice and no invitation should be cancelled once issued. This last requirement implies that the stability conditions relevant to the state variable "inv" are not very restrictive. A person will not be invited if his or her paper is not considered by the Programme Committee (i.e. "sent_ref" is "n") and he or she is not a Group member. However, an invitation may be (or may have been) sent in any other situation. This sublaw may be expressed in tabular form as shown below. State variables and values are as defined earlier, and "-" means "any value" or don't care".

Sublaw: "Authors of processed papers and group members are invited"

Stability Conditions:

sent_ref	grp_mem	inv
-	-	y
n	n	n

Corrective Actions:

Conditions					Actions
pap_dec	sent_ref	grp_mem	inv	-->	inv
-	-	-	y		y
acc	-	-	n		y
rej	-	-	n		y
-	y	-	n		y
-	-	y	n		y

6.2.4. Consistency and completeness testing

Appendix T contains a listing of the IFIP Working Conference system model in the format required by the specifications analysis tools. There are some differences between this model and the one developed above. The differences reflect changes to the system required to correct errors found during this step. The tools also note that some of the rules included in the model are not required to respond to the defined external events. Each of these rules must be examined to determine whether they are redundant or whether there is a deficiency in the model.

---

[102] Notice that mere submission of a paper does not guarantee a person an invitation to attend the Conference. This is an invited paper conference. No paper will be sent to the referees by the Programme Committee unless it was previously invited.

The time required for testing can be considerably reduced by noticing that two state variables appear to be unrelated to the rest of the system. The state variables "pap_prom" and "chair" are not mentioned together, or with any other state variables, in any sublaw. Therefore, they cannot affect the behaviour of any other state variable. While the values of these state variables are of interest to the Programme Committee, they can be handled by subsystems which are independent of the rest of the system. The system as described above has 352 stable states. If "chair" is removed from the system, there will be 176 stable state, and if "pap_prom" is also removed, there will be only 88 stable states. The number of stable states is halved in each case because both state variables have two values, and they may assume either of these values regardless of the state of the rest of the system. In order to save time testing and decomposing the system, these two state variables will be dropped from the model. Subsystems to handle promised papers and the assignment of chairmen can be constructed independently of the other subsystems which will be suggested by the specifications analysis tools.

When the model is entered, the specifications analysis tools will find it to be inconsistent. If a person who was not a Working Group member and did not submit a paper to the Programme Committee becomes a member, the system can change to two different stable states. The relevant state variables and values are shown below.

State Variable	Initial Stable State	After Event	Final Stable States	
grp_mem	n	y	y	y
pap_sub	n	n	n	n
ref_dec	n/a	n/a	n/a	n/a
sent_ref	n	n	n	n
pap_dec	n/a	n/a	rej	n/a

There is an error in the sublaw which determines the final value of "pap_dec". In Appendix S, it was assumed that the decision to include a paper in the Conference is based solely on the validity of the referees' decision. The referees' decision will not be valid if the paper they judged was not sent to them by the Programme Committee or no paper was submitted. The original tabular form of this sublaw is as follows (as developed in Appendix S):

Original incorrect sublaw[103]:

```
Sublaw:     "Papers are accepted nor rejected"
Stability Conditions:
ref_dec     sent_ref    pap_sub     pap_dec
  acc         y           -           acc
  rej         y           -           rej
  n/a         -           -           rej
   -          -           n           n/a
Corrective Actions:
Conditions                                  Actions
ref_dec     sent_ref    pap_sub     -->     pap_dec
  acc         y           -                   acc
  rej         y           -                   rej
  n/a         -           -                   rej
   -          -           n                   n/a
```

The third rule in both the stability conditions and corrective actions sections of the sublaw must be changed as shown below. The corrected sublaw reflects that fact that if the referees' decision is not applicable, the Programme Committee's decision will be neither accept nor reject.

```
Stability Conditions:
ref_dec     sent_ref    pap_sub     pap_dec
  n/a         -           -           n/a
Corrective Actions:
Conditions                                  Actions
ref_dec     sent_ref    pap_sub     -->     pap_dec
  n/a         -           -                   n/a
```

The model is <u>incomplete</u> with respect to an external event which sets the value of "grp_mem" to "y" (i.e. the person becomes a Working Group member). There is an error in the sublaw responsible for setting the value of "sess_ass". A paper may be submitted and neither accepted nor rejected by the Programme

---

[103] Notice that the stability conditions and the corrective actions parts of this sublaw are nearly identical. A "pap_dec" conditions column containing initial values of "pap_dec" could have been added to the corrective actions, but since the final value of "pap_dec" is independent of its initial value, it is easier to simply leave it out.

As is evident in Appendix S, the structures of the stability conditions and the corrective actions are often very similar. This is to be expected since the stability conditions specify the stable combinations of values for the state variables and the corrective actions specify how to attain those values. While this means the analyst must provide seemingly redundant information, the two parts of a sublaw are not always the same and neither may be left out. For example, the stability conditions and corrective actions of the sublaw for determining the value of "inv" (as described in the previous section) are quite different.

Committee, either because it was never sent to the referees or was never returned. The original tabular form of this sublaw is as follows (as developed in Appendix S):

Original incorrect sublaw:

Sublaw: "Accepted papers are assigned to a session"
Stability Conditions:

pap_dec	pap_sub	sess_ass
acc	-	y
rej	-	n
-	n	n

Corrective Actions:

Conditions			Actions
pap_dec	pap_sub	-->	sess_ass
acc	-		y
rej	-		n
-	n		n

Rules specifying the behaviour of the system, when the Programme Committee's decision on a paper is neither accept nor reject, must be added as shown below. These new rules show that papers which are neither accepted nor rejected are not assigned to a session.

Stability Conditions:

pap_dec	pap_sub	sess_ass
n/a	-	n

Corrective Actions:

Conditions			Actions
pap_dec	pap_sub	-->	sess_ass
n/a	-		n

The model is also <u>incomplete</u> with respect to an external event which sets the value of "pap_sub" to "n" (i.e. no paper is submitted to the Programme Committee). The sublaw responsible for setting the value of "ref_dec" does not specify the action to be taken when a paper is not submitted (and the value of "suit" is therefore "n/a") but is somehow returned by the referees[104]. This sublaw, and the sublaw specifying the relationship between paper submission and suitability, are as follows:

[104] Perhaps a referee accidentally returned a paper destined for another conference.

```
Sublaw:      "Referees either accept or reject"
Stability Conditions:
ret_ref      suit        ref_dec
   y          y            acc
   y          n            rej
   n          -            n/a
Corrective Actions:
Conditions                      Actions
ret_ref      suit        -->    ref_dec
   y          y                   acc
   y          n                   rej
   n          -                   n/a

Sublaw:      "Papers may be suitable of unsuitable"
Stability Conditions:
pap_sub      suit
   y          y
   y          n
   n         n/a
```

Rather than alter the referee decision sublaw, it was decided to drop the value of "n/a" for the state variable "suit". In reality, a paper will be either suitable or unsuitable regardless of whether it is actually submitted to the Programme Committee. The sublaw specifying the relationship between paper submission and suitability was also dropped from the model.

If the above corrections are made, the model will be both locally complete and locally consistent. Appendix T contains a listing of the locally complete and consistent model expressed in the syntax required by the specifications analysis tools. Although this model is complete and consistent, the tools note that two rules are not needed to return the system to a stable state after the action of any external event. One of these rules deals with invitation to the Conference, the other with registration.

```
Unnecessary invitation rule:
Conditions                              Actions
grp_mem      sent_ref     inv    -->    inv
   -            -          y            y

Unnecessary registration rule:
Conditions          Actions
inv          -->    del_reg
 n                     n
```

The analyst should confirm that these rules are indeed redundant, and that no semantic error has been made. The reason the rule which sets the value of "inv"

is never activated is trivial. It is not capable of changing the state of the system. That is, the action state variable "inv" must have the same value before and after rule activation. This does not indicate an error in the system model. There is no theoretical reason why such a rule should not be allowed to fire. The specifications analysis tools simply avoid such rules to save time when determining system response paths to external events. The rule which sets the value of "del_reg" is never activated because state variable "inv" will never be assigned a value of "n" during a system response to an external event. Invitations are never withdrawn. Because states are stable before the application of the external events, if "inv" has the value "n" then "del_reg" will also have the value "n". Therefore, this rule is never required to regain stability and is redundant.

6.2.5. Decomposition

6.2.5.1. Parallel/Sequential Decomposition

Parallel/sequential decomposition as performed by the specifications analysis tools leads to three different decompositions for the IFIP Working Conference system. None of these decompositions is exactly the same as the decomposition inherent in the sublaws. The differences between the suggested decompositions will be discussed first. As shall be shown, these differences can be attributed to a deficient system model. One decomposition will be selected for further analysis and the differences between it and the decomposition inherent in the sublaws will be explained. These differences point to "inefficient" sublaw definitions.

The three decompositions suggested by the tools are listed below and shown in diagrammatic form in Figure 23. They differ only in the subsystems responsible for calculating the values of "pap_dec" and "sess_ass" (i.e. the subsystems which decide whether a paper is accepted by the Programme Committee for inclusion in the Conference, and whether a paper is assigned to a session).

Decomposition #1 Complexity = 30.84

4: {del_acc,inv,del_reg} {ret_ref,sent_ref,sess_ass,pap_dec}

3: {grp_mem,inv,sent_ref,inv} {ref_dec,sent_ref,sess_ass}

2: {pap_inv,pap_sub,sent_ref}

1: {ext_inv,grp_mem,pap_inv} {ret_ref,suit,ref_dec}

Decomposition #2 Complexity = 31.49

4: {del_acc,inv,del_reg}

3: {grp_mem,inv,sent_ref,inv} {ref_dec,sent_ref,pap_dec}
{ref_dec,sent_ref,sess_ass}

2: {pap_inv,pap_sub,sent_ref}

1: {ext_inv,grp_mem,pap_inv} {ret_ref,suit,ref_dec}

Decomposition #3 Complexity = 30.35

4: {del_acc,inv,del_reg} {pap_dec,sess_ass}

3: {grp_mem,inv,sent_ref,inv} {ref_dec,sent_ref,pap_dec}

2: {pap_inv,pap_sub,sent_ref}

1: {ext_inv,grp_mem,pap_inv} {ret_ref,suit,ref_dec}

The complexities of all three decompositions are roughly the same. The first suggestion is somewhat surprising in that it shows that "pap_dec" can be determined as a function of "sess_ass". (The functional forms associated with this subsystem, and the other subsystems discussed below, are listed in Appendix U). While it would be possible to construct a system which functioned this way and still fulfilled all the requirements, an analyst would probably reject any suggestion that papers be assigned to sessions before they are accepted by the Programme Committee. The decomposition would be rejected because there are probably other factors which affect the Programme Committee's acceptance and session assignment decisions in addition to those included in the model. For example, time and space considerations may dictate that an otherwise acceptable paper cannot be included in the conference. In other words, the model does not embody all of the analyst's knowledge pertaining to the dynamics of the system. That there is some missing information is also apparent in the second decomposition. It shows that the values of "pap_dec" and "sess_ass" can be determined in parallel using the same input state variables. In the third decomposition, "sess_ass" is shown as a function of "pap_dec" only. This is

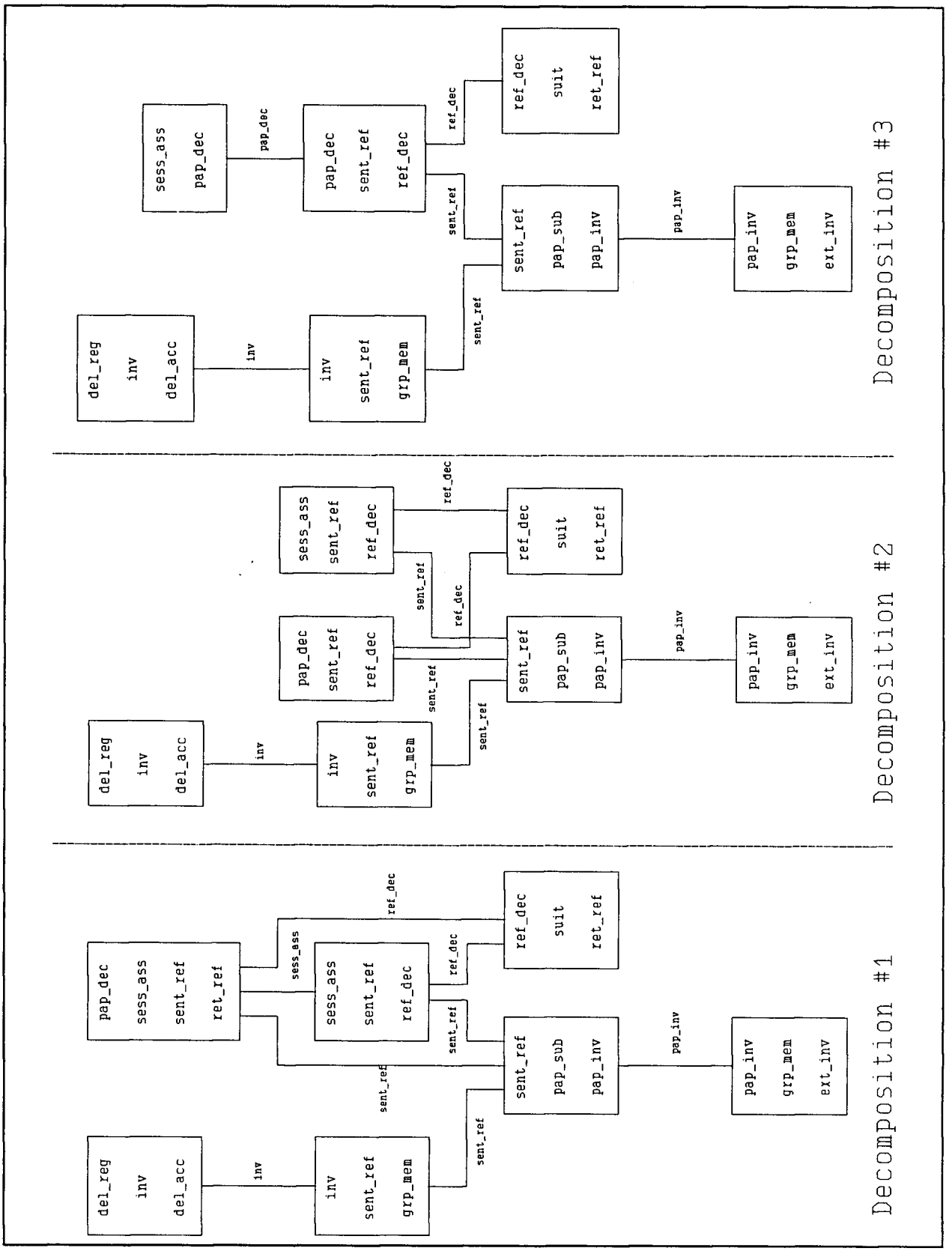


Figure 23: The parallel/sequential decompositions suggested by the specifications analysis tools for the IFIP Working Conference system.

closest to the intuitive sequence of actions within the system. The model could be enhanced to include more of the factors influencing the Programme Committee's decision to accept a paper and assign it to a session. The enhanced model could then be decomposed again to ensure that the suggested decompositions fully agree with the analyst's knowledge of the system dynamics. However, such a detailed analysis would not be appropriate in this chapter. The third decomposition will be selected for further analysis.

In the third decomposition, the subsystems responsible for calculating the values of "pap_inv", "ref_dec", "sent_ref", and "del_reg" are described by the same state variables used to specify the relevant sublaws. However, there are some differences between the subsystems suggested by the tools and those intuitively obvious from the sublaw specifications. As discussed below, these differences point to redundant rules or "over-specification" in the model.

1. The sublaw specifying the calculation of "inv" mentions state variables "pap_dec", "grp_mem", "inv" and "sent_ref", but the subsystem is not described by "pap_dec". The tools recognized that the decision to invite a person to attend the Conference depends only on whether that person is a group member, whether he or she submitted a paper, and whether an invitation was previously issued. The decision to accept or reject the paper is irrelevant as authors of both accepted and rejected papers are invited.
2. The sublaw specifying the calculation of "pap_dec" mentions state variables "ref_dec", "sent_ref", and "pap_sub". However, the tools recognized that the value of "pap_sub" is not required in order to determine the value of "pap_dec". That is, it is not necessary to know explicitly whether a paper was submitted, only whether it was sent to the referees and whether they found it to be suitable.
3. Similarly, the sublaw specifying the calculation of "sess_ass" mentions state variables "pap_dec", and "pap_sub". However, the tools recognized that the value of "pap_sub" is not required in order to determine the value of "sess_ass".

6.2.5.2. Conditional Decomposition

The selected decomposition shows the decision to invite a person to attend the Conference (represented by the value of the state variable "inv") being made at level 3. This results from the fact that in order to make the decision under all possible circumstances it must be known whether a person is a Working Group member (represented by "grp_mem") and whether the paper was accepted for further consideration (represented by "sent_ref"). Such a subsystem is likely to be unacceptable to the analyst as invitations to Working Group members should be sent long before papers are received from either Group members or external invitees. The subsystem {grp_mem,inv,sent_ref,inv} is a candidate for conditional decomposition. When conditional decomposition is applied to the subsystem by the specifications analysis tools, three suggestions are made.

1. [grp_mem = y]{inv} [grp_mem = n]{inv,sent_ref,inv}

This conditional decomposition shows that as long as the person is a Working Group member, there is only one possible value for "inv". An invitation should be sent. However, if the person is not a working group member, the invitation decision must be delayed until a paper is submitted and it is decided whether to send the paper to the referees (i.e. it is accepted for consideration by the Programme Committee).

2. [inv = y]{inv} [inv = n]{grp_mem,sent_ref,inv}

Similarly, this conditional decomposition shows that if an invitation has been sent the final value of "inv" is known. Since invitations are not cancelled, the final value of "inv" will be "y". However, if no invitation is sent, the values of both "grp_mem" and "sent_ref" must be considered.

3. [sent_ref = y]{inv} [sent_ref = n]{grp_mem,inv,inv}

This conditional decomposition shows that if a paper is sent to the referees, there is only one possible value for "inv" (i.e. an invitation will be sent). However, if the paper is not sent to the referees, it is necessary to know

whether the person is a Working Group member before deciding whether to send an invitation.

When choosing between the three conditional decompositions, an analyst would likely favour decompositions which allow decisions to be made at a lower level (i.e. earlier) than that indicated by the parallel/sequential decomposition. For example, the conditional decompositions for the invitation subsystem allow the parallel/sequential level structure to be modified as follows:

4: {del_acc,inv,del_reg} {pap_dec,sess_ass}
3: [grp_mem=n]{inv,sent_ref,inv} {ref_dec,sent_ref,pap_dec}
2: {pap_inv,pap_sub,sent_ref}
1: {ext_inv,grp_mem,pap_inv} {ret_ref,suit,ref_dec} [grp_mem=y]{inv}

or

4: {del_acc,inv,del_reg} {pap_dec,sess_ass}
3: [inv=n]{grp_mem,sent_ref,inv} {ref_dec,sent_ref,pap_dec}
2: {pap_inv,pap_sub,sent_ref}
1: {ext_inv,grp_mem,pap_inv} {ret_ref,suit,ref_dec} [inv=y]{inv}

or

4: {del_acc,inv,del_reg} {pap_dec,sess_ass}
3: [sent_ref=y]{inv}[sent_ref=n]{grp_mem,inv,inv}{ref_dec,sent_ref,pap_dec}
2: {pap_inv,pap_sub,sent_ref}
1: {ext_inv,grp_mem,pap_inv} {ret_ref,suit,ref_dec}

The first alternative shows that under some circumstances the decision to invite a person may be made at level 1 (i.e. if a person is a Group member, he or she should be invited). The second merely affirms the fact that invited people will always be invited (i.e. invitations are not revoked). This decomposition does not allow the original invitation decision to be made any earlier. The third does not allow any decision to be advanced in time since both conditional subsystems are still at level 3. Therefore, the first alternative

will be adopted for purposes of comparison with JSD and ACM/PCM. This decomposition is shown in diagrammatic form in Figure 24.

The complexity of the subsystem {grp_mem,inv,set_ref,inv} is 4.35. The total complexities of each conditional decomposition is 3.25. Therefore, in addition to adding flexibility to the timing of the decision to invite a person, there is significant complexity reduction through conditional decomposition.

6.3. Jackson System Development (JSD)

The results of analysis using SELMA can be compared to those of more established systems analysis techniques. In this section and the next, the objects identified by Jackson System Development (JSD) and Active and Passive Component Modelling (ACM/PCM) for the IFIP Working Conference system will be compared to the hierarchy of subsystems discovered above.

In JSD, "the real world is described in terms of entities, actions they perform or suffer, and the orderings of those actions" (Jackson, 1983, p. 23). The notion of "entity" in JSD a different from that used in most database modelling methods. Jackson suggests that if an analyst were to construct an Entity-Relationship Model (Chen, 1976) of the fact that "a customer is located in a certain town, each town in a certain county, and each county in a certain state", he or she would identify town, country and state as entities. In general, JSD would not. Town, country, and state would not be entities unless they perform or suffer significantly time-ordered actions in the real world. JSD entities are also not identical to the objects of object-oriented programming, but they are similar enough that for the purposes of this research the terms can be used interchangeably.

Jackson notes that "some database development methodologies compensate for the static nature of the database by specifying updating constraints as part of the database definition" (Jackson, 1983, p. 19). On the other hand, JSD begins "by constructing a dynamic model directly, based on a dynamic description of the real world" (p. 19).

JSD provides four general rules for identifying entities (Jackson, 1983, p. 40):

1. An entity is to be included ... if the system will need to produce or use information about it.

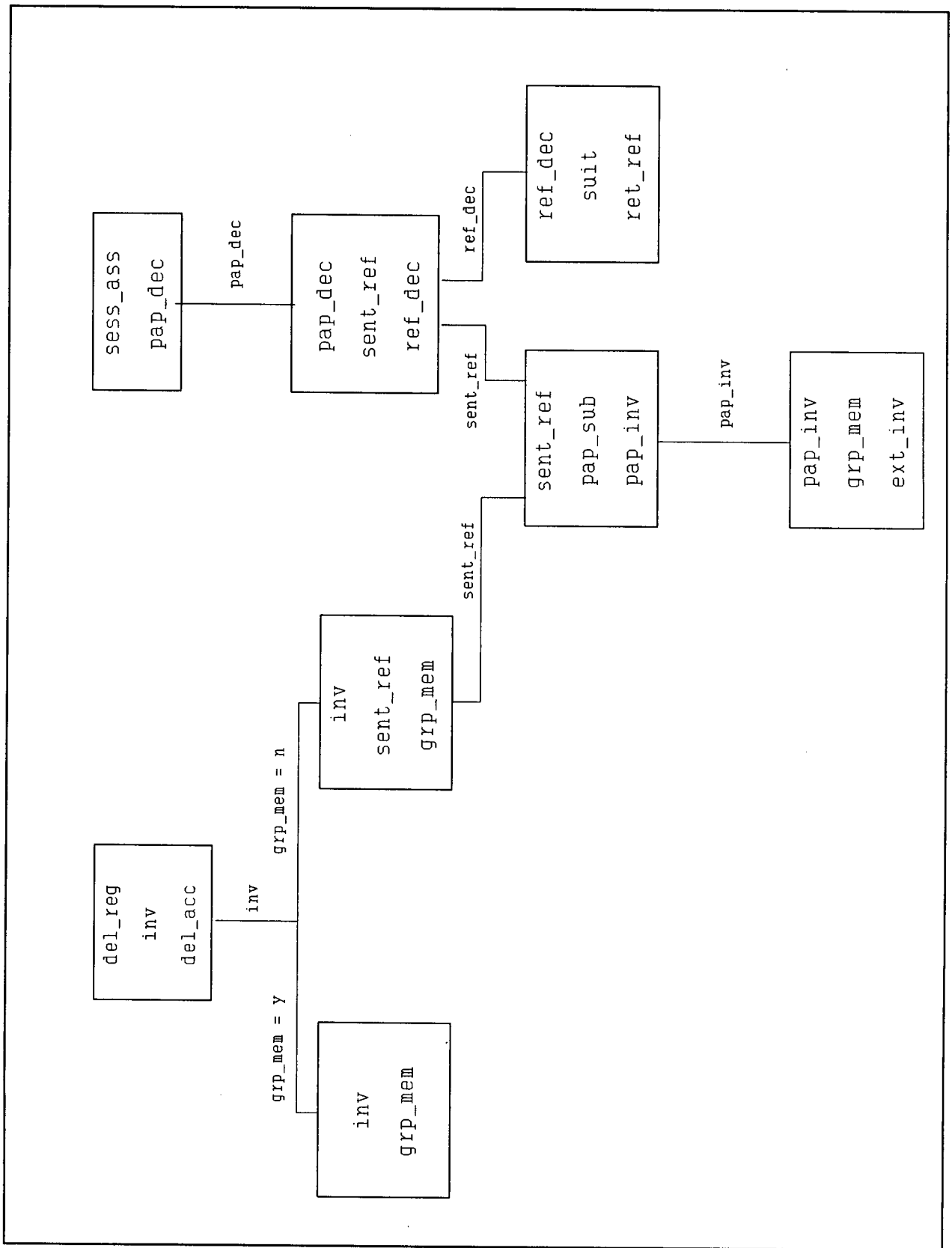


Figure 24: A decomposition for the IFIP Working Conference system incorporating conditional decomposition.

2. An entity must perform or suffer actions, in a significant time ordering. Thus "date" and "age" would not be permissible entities in any plausible system.
3. An entity must exist in the real world outside the system, and must not be merely part of the system itself or a product of the system. Thus "error report" would not be a permissible entity.
4. An entity must be capable of being regarded as an individual, and, if there is more than one entity of a type, of being uniquely named.

The first rule corresponds to the first step in applying SELMA where state variables are identified by considering the functions the implemented information system is to perform. The third rule is identical to the suggestion made when defining the sublaw governing invitations to the IFIP Working Conference, that state variables describing artifacts of the implemented information system not be included in a system model[105]. In the SELMA model of the Working Conference system, no state variables describe the various lists to be produced. It is fairly easy to see whether a potential entity satisfies rules 1 and 3. It is considerably more difficult to apply rules 2 and 4. The phrase "significant time ordering" and the property "capable of being regarded as an individual" are not clearly defined in JSD. This is where the decomposition techniques embodied in the specifications analysis tools can help.

In JSD "time ordering" refers to the sequencing of actions. JSD actions correspond to the external and internal events of SELMA. In a SELMA system model, no ordering of events is given. However, the decompositions suggested by the tools do impose a "significant" sequence on the events. Namely, the subsystems at lower levels in the decomposition must determine the values of their output state variables before those variables are used by higher-level subsystems. Also, it seems reasonable to assume that an individual should be describable by a subsystem satisfying the independence criteria suggested by Wand and Weber (1988) and described in Chapter 3. That is, an individual should embody all the information required in order to predict its behaviour in a given situation or state. Now consider the decomposition of the IFIP Working Conference system suggested by the specifications analysis tools:

---

[105] Unless, of course, the description of the functions to be provided by the information system require such state variables.

4: {del_acc,inv,del_reg} {pap_dec,sess_ass}

3: [grp_mem=n]{inv,sent_ref,inv} {ref_dec,sent_ref,pap_dec}

2: {pap_inv,pap_sub,sent_ref}

1: {ext_inv,grp_mem,pap_inv} {ret_ref,suit,ref_dec} [grp_mem=n]{inv}

Figure 24 more clearly shows the sequence of subsystem activations.

Any subsystem, or collection of subsystems which are activated in sequence, will meet the JSD criteria for suitable entities. For example, entities consisting of the following subsystems could be defined. Interconnections between these entities are illustrated in Figure 25. It is in no way implied that this partition represents a "good" selection of entities.

Entity 1: Invitation and Registration Division
{del_acc,inv,del_reg},[grp_mem=n]{inv,sent_ref,inv}

Entity 2: Paper Requesting and Receiving Division
{pap_inv,pap_sub,sent_ref},{ext_inv,grp_mem,pap_inv} [grp_mem=y]{inv}

Entity 3: Final Programme Committee Activities Division
{ref_dec,sent_ref,pap_dec},{pap_dec,sess_ass}

Entity 4: Referees
{ret_ref,suit,ref_dec}

Communication between entities is in the form of state variable values. Communication is required where an arrow crosses an entity boundary in the above Figure 25. There are, of course, several other ways in which the above subsystems could be grouped into entities.

McNeile (1986) applied JSD to the IFIP Working Conference problem. The identified entities and associated actions were as follows:

Entity Name	Description	Action Name	Description
paper	submitted paper	promise	paper is promised to the Programme Committee

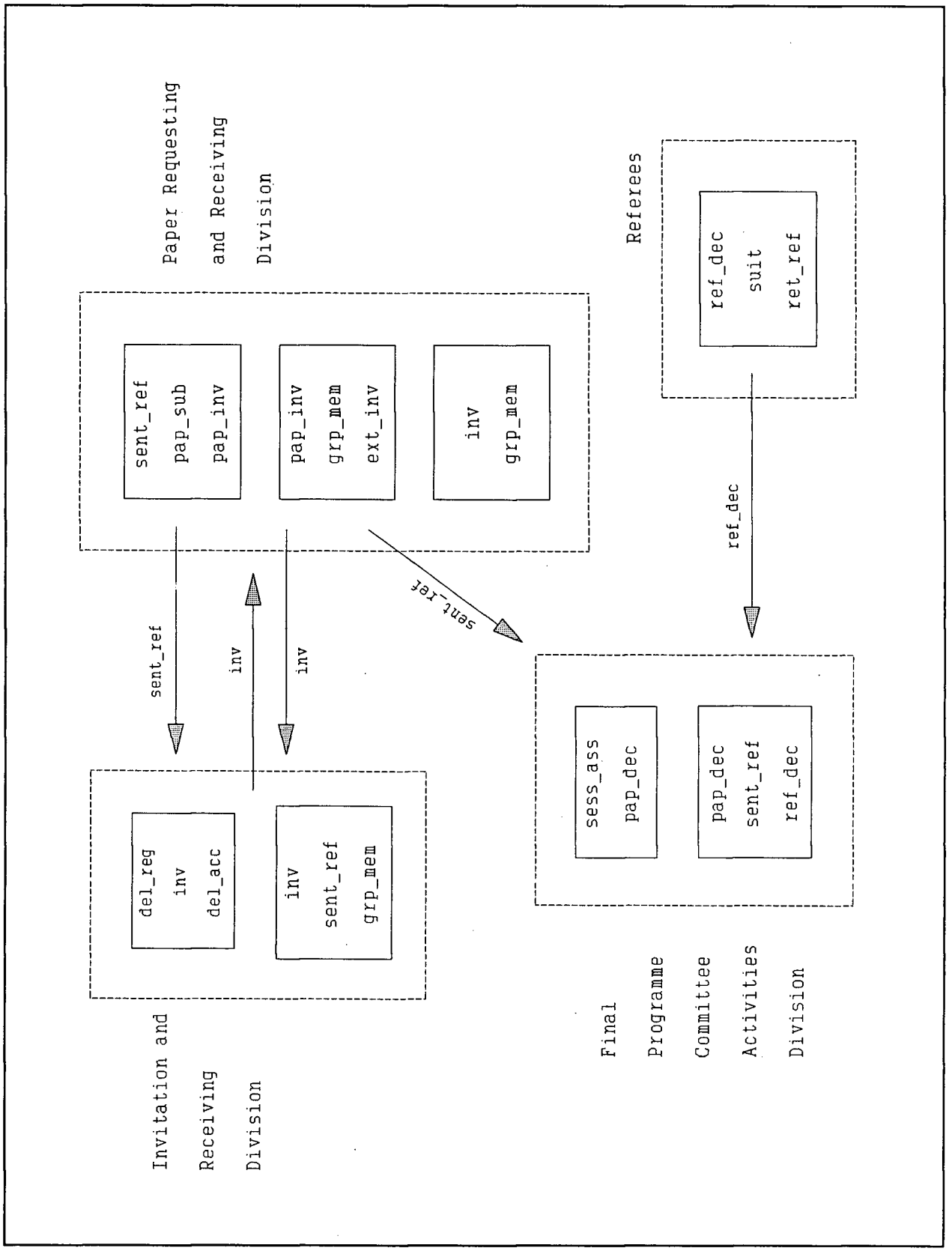


Figure 25: A possible entity structure for the IFIP Working Conference problem based on a decomposition suggested by the specifications analysis tools.

		submit	submit to the Programme Committee
		sendref	send to a referee
		retref	returned from the referee
		reject	Programme Committee rejects the paper
		accept	Programme Committee accepts the paper
		sessdef	paper is allocated to a session
pot_del	potential delegate	invite	invite person to the Conference
		register	delegate registers for Conference
		attend	delegate attends the Conference
group_mem	Working Group member	join	join Group
		leave	leave Group
paper_ref	paper/referee assignment	sendref	send paper to a referee
		retref	paper is returned by the referee

These entities and actions may be combined to form the entity structure diagrams of Figure 26. Entity structure diagrams show the order in which actions are applied to an entity. They are read from left to right. For example, the "paper" entity may be promised, must be submitted, must be sent to one or more referees, will be either rejected or processed following acceptance, and if necessary, this processing will consist of acceptance followed by assignment to a session. The paper/referee entity is a special entity created to model the fact that a paper can be sent to more than one referee. In the SELMA model, papers are sent to referees and returned by them. The actions relating to individual referees are not modelled. The list of actions corresponds to the external and internal events (represented by the calculation of output state variables) identified in the SELMA model. There are a few minor differences resulting from different interpretations of the problem description. McNeile does not model the initial call for papers. It appears that any submitted paper will be considered by the Programme Committee. In the SELMA model, only invited papers are considered. McNeile also models actual attendance at the Conference, while this is considered beyond the scope of the SELMA model. This difference likely results from different interpretations of the "Generating a final list of attendees" activity of the Organizing Committee.

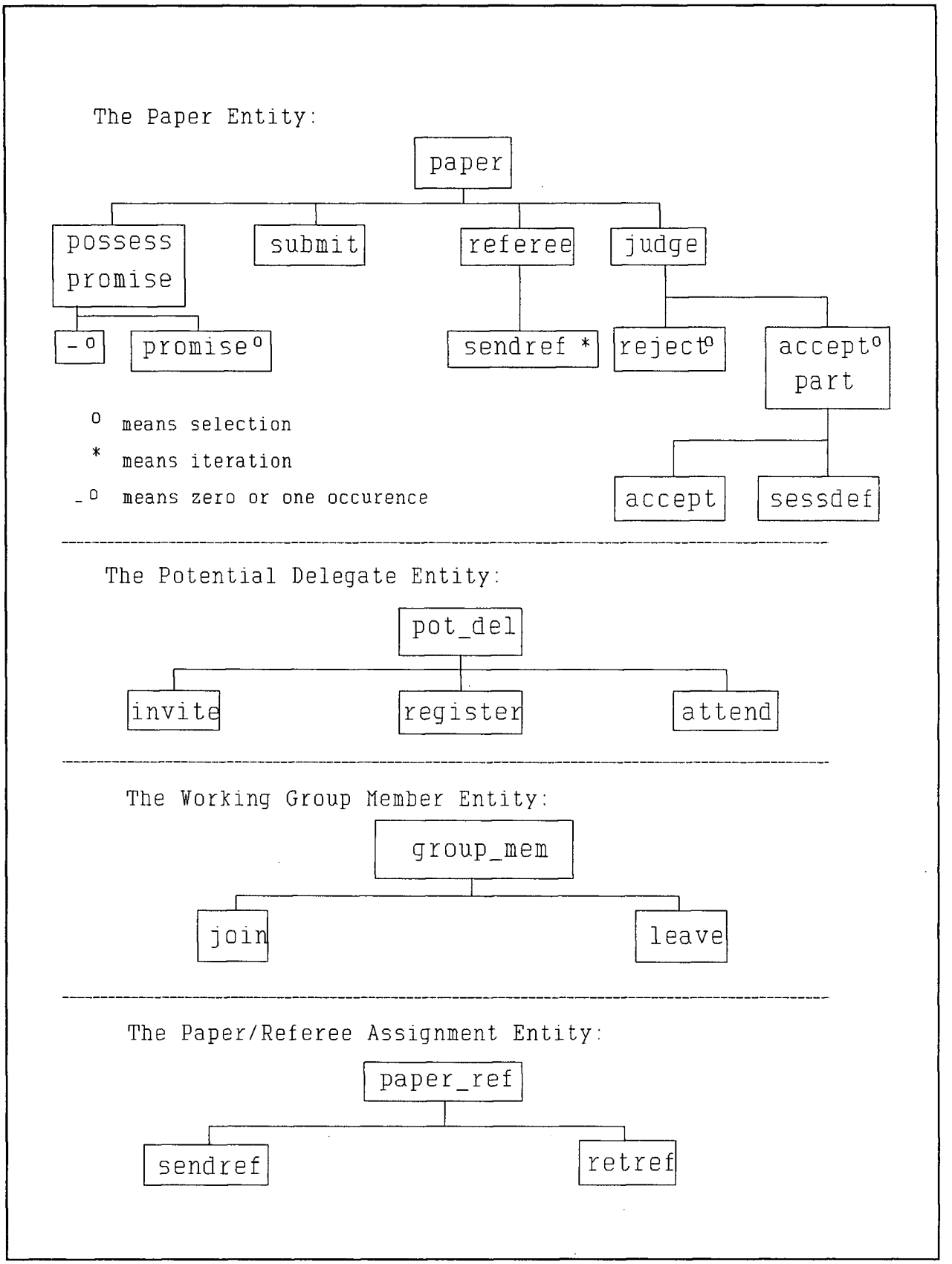


Figure 26: JSD entity structure diagrams for the IFIP Working Conference Problem.

McNeile's working group member entity is represented by a single state variable in the SELMA model. The state variable "grp_mem" is used to indicate whether a person is a member of the Working Group. External events are defined to model the joining and leaving actions. McNeile's remaining two entities correspond to the partition of the subsystems shown in Figure 27. Notice that there is much less communication (in the form of transferred state variable values) between the entities of this subsystem partition than between those of the arbitrary partition suggested earlier. One could say that there is less coupling and greater cohesion in this partition[106]. The sequence of subsystem activations inside each SELMA entity closely resembles the order of the sequential actions in each JSD entity. The delegate registration and invitation entity (comprised of the "del_reg" and "inv" subsystems) shows invitation preceding registration in much the same way as is shown be the JSD "pot_del" entity. McNeile has chosen to model attendance as well. The SELMA paper entity shows paper invitation ("pap_inv") followed by referee judgement ("sent_ref" and "ref_dec") followed by judging ("pap_dec") followed by session assignment ("sess_ass"). This is very similar to the sequence of actions in the JSD paper entity. Notice, however, that McNeile has included a paper promising action in his paper entity. The SELMA decomposition has no corresponding subsystem because the "pap_prom" state variable was dropped from the model for efficiency reasons. There were no interactions between the "pap_prom" state variable and any other state variable. The single state variable subsystem {pap_prom} could be appended to the SELMA paper entity but its activation may occur at any time (i.e. no other subsystem requires the value of "pap_prom"). The JSD entity insists that if a paper is promised at all, it must be promised before it is submitted. This is intuitively reasonable. But what happens if the paper is received by the Programme Committee before the promise arrives?[107] The JSD entity, as shown, simply cannot handle such a situation. The SELMA entity notes that the order of these events is irrelevant.

While the differences between the JSD and SELMA solutions are interesting, what is of primary importance is that the decomposition suggested by the specifications analysis tools maps easily into the entity decomposition produced

---

[106] Perhaps entity identification can be viewed as the decomposition of the system of subsystems identified by the specifications analysis tools.

[107] Perhaps there was a problem with the mail system.

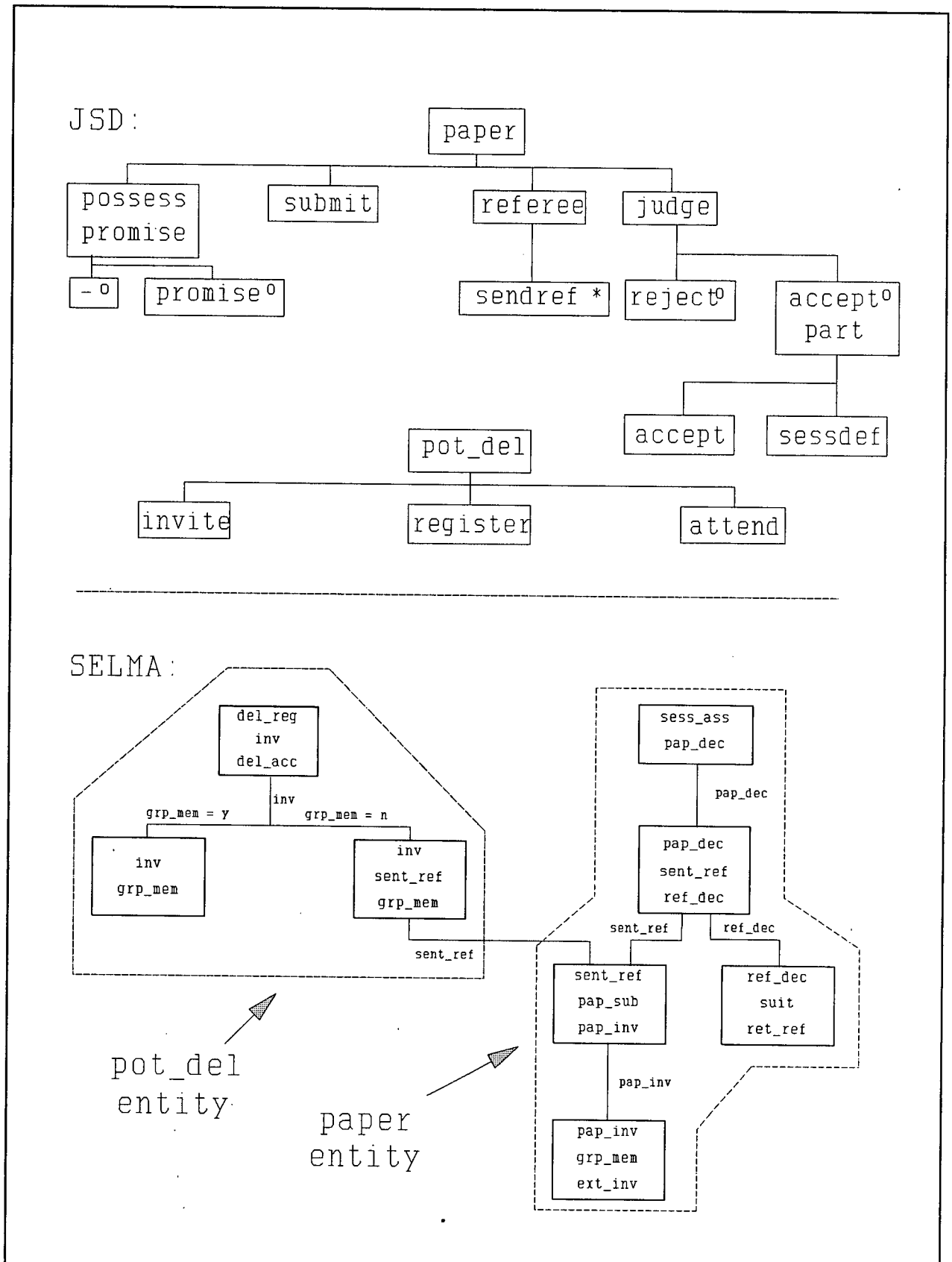


Figure 27: Mapping the JSD solution into the SELMA solution of the IFIP Working Conference problem.

by at least one practitioner of JSD. Also note that the specifications analysis tools were able to perform decomposition automatically, while JSD required substantial human judgement.

6.4. Active and Passive Component Modelling (ACM/PCM)

The real-world modelling aspects of ACM/PCM will be briefly discussed and compared to JSD and SELMA. Brodie and Silva's (1982) solution to the IFIP Working Conference problem will be described. There are considerable differences between the ACM/PCM solution and those of JSD and SELMA. The reasons for these differences will be explained and the superiority of JSD and SELMA for high-level, real-world modelling will be demonstrated.

ACM/PCM is notable for its extensive use of abstraction[108] which is claimed to result in "equal emphasis on the integrity of structural and behavioural properties of an application... Structure refers to states and static properties. Behaviour refers to state transitions and dynamic properties" (Brodie and Silva, 1982, pp. 41). It is also claimed that "both structural and behavioural properties of objects can be designed in isolation..." (p. 41). In ACM/PCM, structural properties are equivalent to static properties, and include decomposition. This is almost the antithesis of both JSD and SELMA. As noted earlier, JSD begins "by constructing a dynamic model directly, based on a dynamic description of the real world" (Jackson, 1983, p. 19). SELMA channels most of an analyst's effort into defining the behavioural properties of a system. It is these behavioural properties which determine the structure of the system. In SELMA, static properties consist of only the list of state variables[109] included in the system model. It will be interesting to see how the objects discovered by ACM/PCM compare with those of found in the previous sections.

Unfortunately ACM/PCM does not provide much guidance for the identification of objects. The first few steps in the methodology are described below (Brodie and Silva, 1982, p. 50).

---

[108] Abstraction is defined as "the suppression of some detail in order to emphasize more appropriate detail" (Brodie and Silva, 1982, p. 41).

[109] and their associated values.

A. CONCEPTUAL MODELLING

A.1 CONCEPTUAL MODELLING OF STRUCTURE

A.1.1. Structure Design

A.1.1.1. Object identification

Identify and name the basic objects to be represented.

A.1.1.2. Object classification:

Classify each object as either permanent or temporary and independent or dependent[110].

A.1.1.3. Construct individual object schemes:

Considering permanent objects first, apply the aggregation / decomposition, generalization / specialization and association / membership to each identified object. Consider first the independent and then the dependent objects. This step results in identifying the basic objects.

...

Aggregation / decomposition, generalization / specialization, and association / membership refer to "part-of", "is-a", and "member-of" relationships between objects, respectively. No further advice pertaining to object identification is given. As a result of performing step A.1.1.1. the following objects were identified for the IFIP Working Conference problem (Brodie and Silva, 1982, pp. 57-58).

Object Name	Description
accepted	paper accepted for presentation at the Conference
assigned_to_session	paper that is enroled for presentation in one session
attendee	a person who will be present at the Conference
author	person who submitted a paper

[110] An object is "permanent" if it cannot be created or destroyed by the system, and "independent" if its existence does not depend on the existence of some other object. These classifications are not relevant to the discussion here and will not be considered further.

cfp_divulgation	technical publisher or association who will get Conference announcements for publication in their magazines
cfp_invitee	a person who will get an invitation to submit a technical paper to the Conference
conference_invitee	person who got an invitation to attend the Conference
dropped	paper withdrawn from the Conference by the author
intended_attendee	person who sent a registration to attend the Conference
letter_of_intent	a correspondence that indicates that an individual or more wishes to present one or more papers at this Conference
national_representative	IFIP national representative
not_assigned to a session	accepted paper that is currently not enroled in any session
oc_chairperson	organizing committee chairperson
oc_member	person that belongs to the Working Conference organizing committee
paper	a technical report submitted to be presented at the Conference
paper_referee_assignment	refers to transferring a paper to a referee for evaluation, consists of person, comments, date of assignment, deadline, date comments received, and paper
pc_chairperson	program committee chairperson
pc_member	person involved in the Working Conference paper selection
referee	appointed person to review submitted papers
rejected	technical paper not accepted by the Programme Committee
session	a technical meeting where technical papers are presented to people attending the Conference
session_chairperson	individual that will be chairing the session
subscriber	person who sent a letter showing interest in presenting a paper

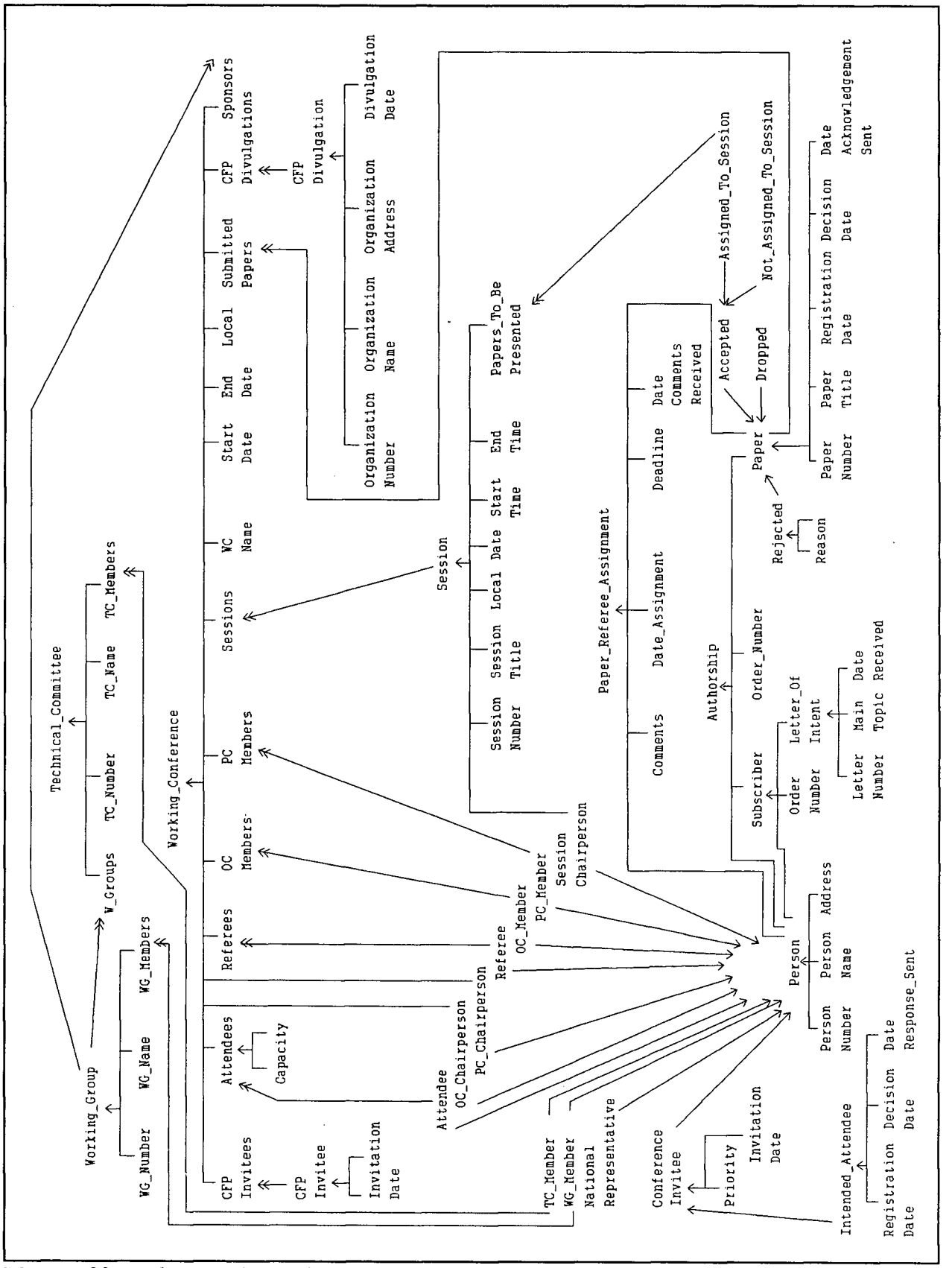


Figure 28: The ACM/PCM object scheme for the IFIP Working Conference problem (from Brodie and Silva, 1982, p. 61).

tc_member	representative of IFIP member society to a Programme Committee
technical_committee	IFIP Programme Committee
wg_member	working group member
working_conference	an IFIP technical conference to discuss and debate a specific issue

Obviously, the ACM/PCM model begins with more detail and has a broader scope than either the JSD or SELMA models. The scope is greater because Brodie and Silva have dealt with membership of the various committees and advertising. Also, the inclusion of a working conference object implies the model can handle multiple conferences. The static relationships between these objects (and some others) are diagrammed in Figure 28. In Figure 29, the scope of the model has been reduced to facilitate comparison with JSD and SELMA.

The decomposition illustrated in Figure 29 is quite different from those provided by either JSD or SELMA. The two central objects are "person" and "paper". "Paper" was included in the initial list of objects. "Person" was added because "wg_member", "intended_attendee", etc. are "naturally modelled as being categories of an object PERSON, which is an aggregate of name, address and a unique number (system's internal key)" (Brodie and Silva, 1982, p. 59). Instead of using "author" as a category of "person", the authors decided to create an aggregate called "authorship" to show that a relationship may exist between a person and a paper. Thus, instead of showing that "author" is-a "person", they show that "person" is part-of an "authorship". It is not immediately obvious how to compare this object hierarchy with those of JSD and SELMA.

JSD identified (and with appropriate partitioning of the suggested subsystems, SELMA could identify) two major entities or objects: grp_mem and paper. "Paper" is certainly also identified by ACM/PCM, as are many subcategories of "paper", namely, rejected papers, accepted papers, papers assigned to sessions, and papers not assigned to sessions. In JSD these subcategories are represented by the values of some of the attributes of the "paper" entity. For example, a rejected paper could be represented by a paper with the value "reject" assigned to the attribute "paper_decision". The same technique is used in SELMA except that attributes of an entity are called state variables of a subsystem. However, "pot_del" (i.e. a potential delegate to the

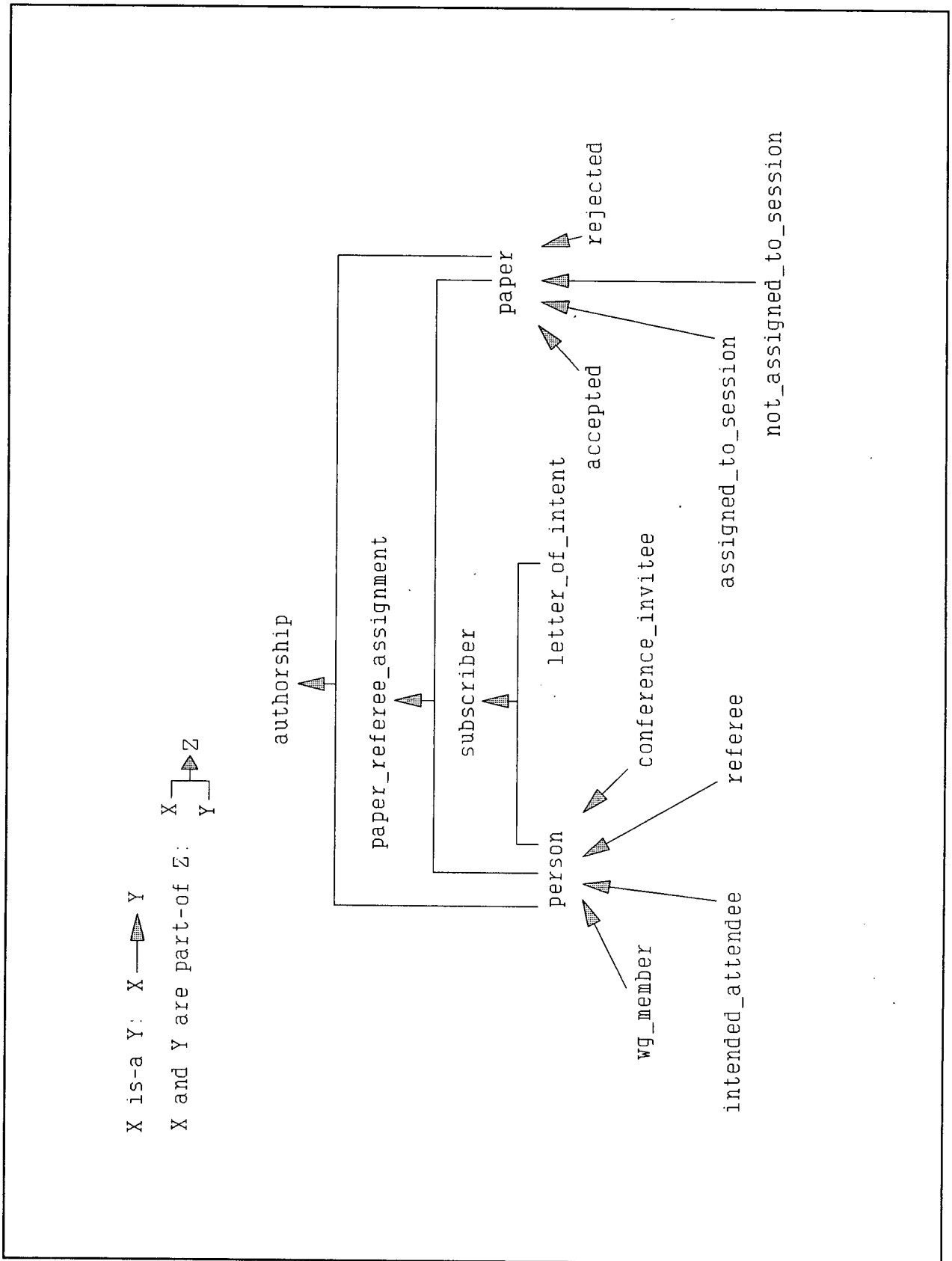


Figure 29: An extract from the ACM/PCM object scheme for the IFIP Working Conference problem.

conference) is more specific than "person". "Person" includes "referee" as a subcategory. "Pot_del" does not necessarily include referees. That is, the fact that someone is a referee is irrelevant to his or her participation in the conference itself. Where then, does the object "person" come from, and what are the consequences of including it in the system model?

Brodie and Silva have used world knowledge, not explicitly given in the description of the problem, to show a relationship between several different system objects. Nothing in the system description says that referees and Working Group members are somehow linked. No defined dynamics of the system (i.e. ACM/PCM actions, JSD actions or SELMA external and internal events) affect or examine group members and referees together[111]. Only the knowledge that Working Group members and referees are both human beings allows this link to be made. But is there anything wrong with making such a connection? If the connection could support possible extensions to the system (see footnote), why not include it from the start? Do JSD and SELMA fail to capitalize on an important and useful source of system information, namely general world knowledge?

Answering the last question first, let us determine what changes to the JSD model would be required in order to model an activity which linked referees and potential delegates. Suppose a free copy of the proceedings is given to all referees who are Working Group members. The "pot_del" entity could be modified as shown in Figure 30. A "referee part" concerned with the distribution of free copies of the proceedings has been added. An attribute indicating whether a delegate is a referee will also be required.

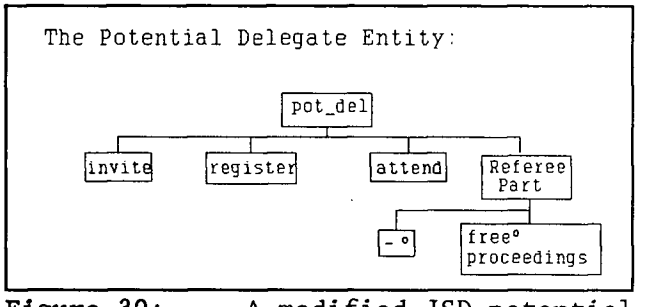


Figure 30: A modified JSD potential delegate entity for modelling the distribution of free copies of the proceedings.

The new activity (i.e. distribution of proceedings) does not require the creation of a new object. However, at the implementation level,

[111] It is possible to define a dynamic operation which would link Group members and referees. The system could be extended to include a rule stating that Group members who are referees are to be treated differently from other Group members. Perhaps they could be given a free copy of the proceedings. However, no rule of this form is given in the IFIP Working Conference problem description.

using an inheritance scheme whereby both "wg_member" and "referee" objects will have some identical attributes ensures that the objects can be successfully compared (assuming both Group member and referee objects were to be created). If the keys to the "wg_member" and "referee" objects were different, determining which Group members are also referees could be quite difficult. Such problems are implementation issues and, as such, should not be part of the system model. But why not? The usual argument against "mixing" levels of abstraction involves managing complexity[112]. Abstracting implementation issues out of the early stages of systems analysis and design will result in fewer concepts which the analyst must consider at any one time. Implementation issues could also obscure important relationships between the objects in a model. This happens in the ACM/PCM model.

Consider the object "paper_referee_assignment" as diagrammed in Figure 29. This object is used to show one kind of relationship between a paper and a person. However, the actual relationship is not between a paper and a person but between a paper and a referee. The inclusion of the "person" object between "referee" and "paper_referee_assignment" obscures this fact. As drawn, papers could be refereed by any "wg_member", "intended_invitee", "conference_attendee" or "referee". A similar observation can be made for the "authorship" object. This error is not difficult to correct. Figure 29 can be changed to appear as in Figure 31. Unfortunately, the error has been propagated throughout the rest of the ACM/PCM model. The action used to model the sending of a paper to a referee is shown below (Brodie and Silva, 1982, p. 77). The syntax "x:y" means that "x" is an instance of the object "y". Several of the objects mentioned in the action do not appear in Figure 29, but are shown in Figure 28. The IN and OUT sections specify the inputs and outputs to the action, respectively. The LOCAL section declares the objects which are local to the operation of the action. These objects are used in the execution of the DB-OPERATION (database operation) associated with the action. The PRE-CONDITIONs and POST-CONDITIONs are lists of predicates which must be satisfied before the action can begin and end, respectively.

```
ACTION  insert_paper_referee_assignment(pe_n, d_a, dlne, pa_n)
     IN  (pe_n:person_number, d_a:date_assignment, dlne:deadline,
```

---

[112] Removal of implementation concerns also allows a single conceptual model to be used for many alternative implementations. The analyst is not committed to a particular implementation right from the start.

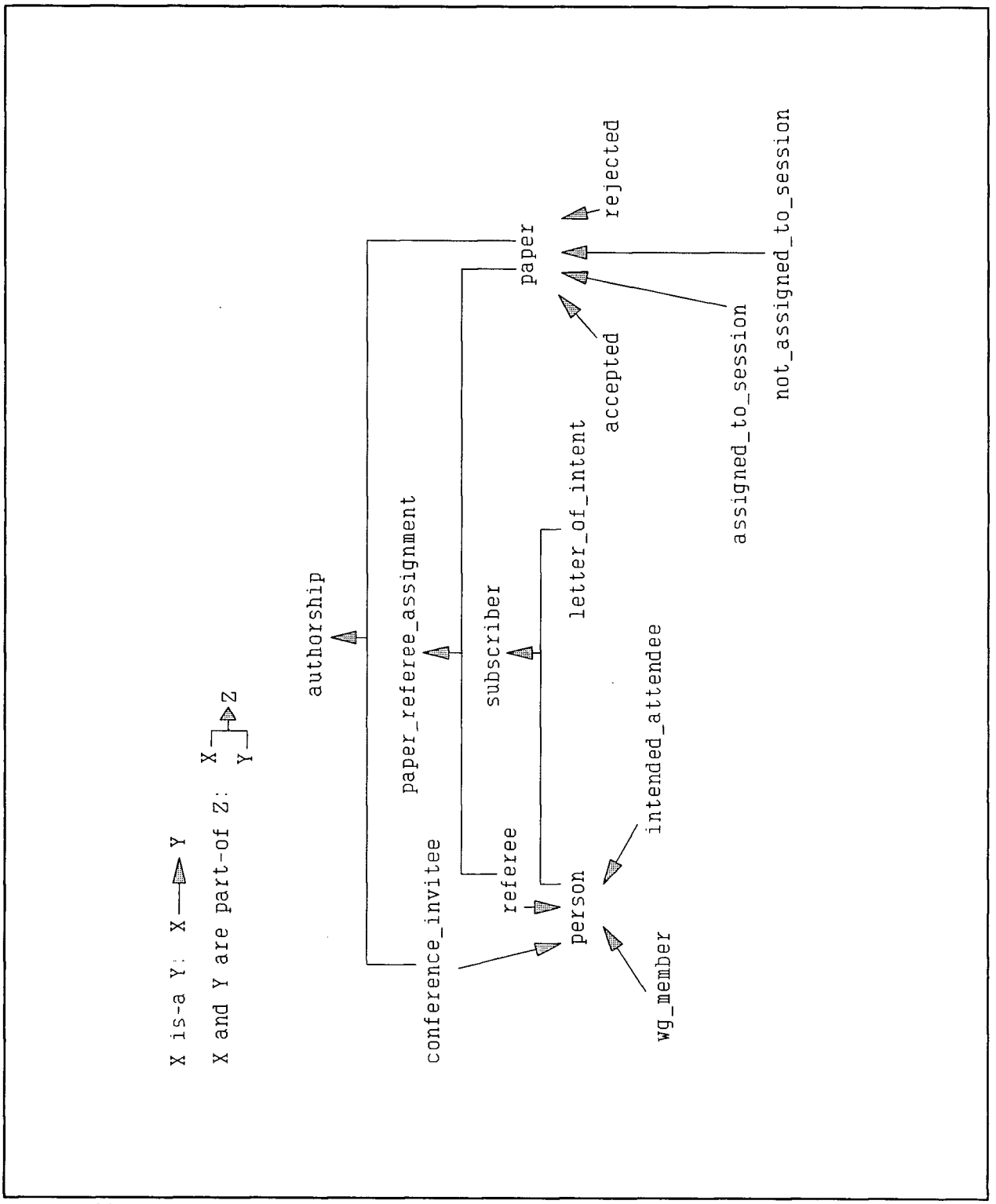


Figure 31: The corrected ACM/PCM object scheme for the IFIP Working Conference problem.

```
      pa_n:paper_number)
OUT   (pra:paper_referee_assignment)
LOCAL  (pe:person, pa:paper)
PRE-CONDITION:
      paper_exist(pa_n)?
      person_exist(pe_n)?
      the_person_is_not_refereeing_the_paper(pe_n, pa_n)?
POST-CONDITION:  true?
DB-OPERATION:  INSERT paper_referee_assignment(pe, d_a, dlne, pa)
```

The person_exist(X) predicate is defined to be true if the "person_number" assigned to X corresponds to an instance of the "person" object. This is not the correct pre-condition. The predicate should test whether an instance of the "referee" object exists. This error results from creating an action to complement the object scheme which shows "person" as part of a "paper_referee_assignment". This error would not have occurred if definition of the "person" object had been deferred to a later stage of design. The "person" object is not required to model the behaviour of the IFIP Working Conference system, and its inclusion has led to at least one error in the ACM/PCM system specification.

The ACM/PCM object scheme can be modified to more closely reflect the objects identified by JSD and SELMA as shown in Figure 32. "Person" can be replaced by the object "pot_del" (for potential delegate). The object "wg_member" has become a subcategory of "conference_invitee" to show that all "wg_members" are invited. "Referee" is now part-of the paper object, and is no longer associated with the same objects as "intended_attendee" and "conference_invitee". The object "external_invitee" has been added to represent those persons invited to the Conference who are not Working Group members. These people must have submitted a paper to the Programme Committee. This requirement is shown by including "paper" as part-of "external_invitee". These changes to "external_invitee" are probably not required as the result of an error made by Brodie and Silva. They likely result from different interpretations of the IFIP Working Conference Problem.

6.5. Conclusions

SELMA has been successfully applied to a "real" system: the IFIP Working Conference problem. The focus of states, events, and laws allowed the use of automated specifications analysis tools. These tools proved to be very useful

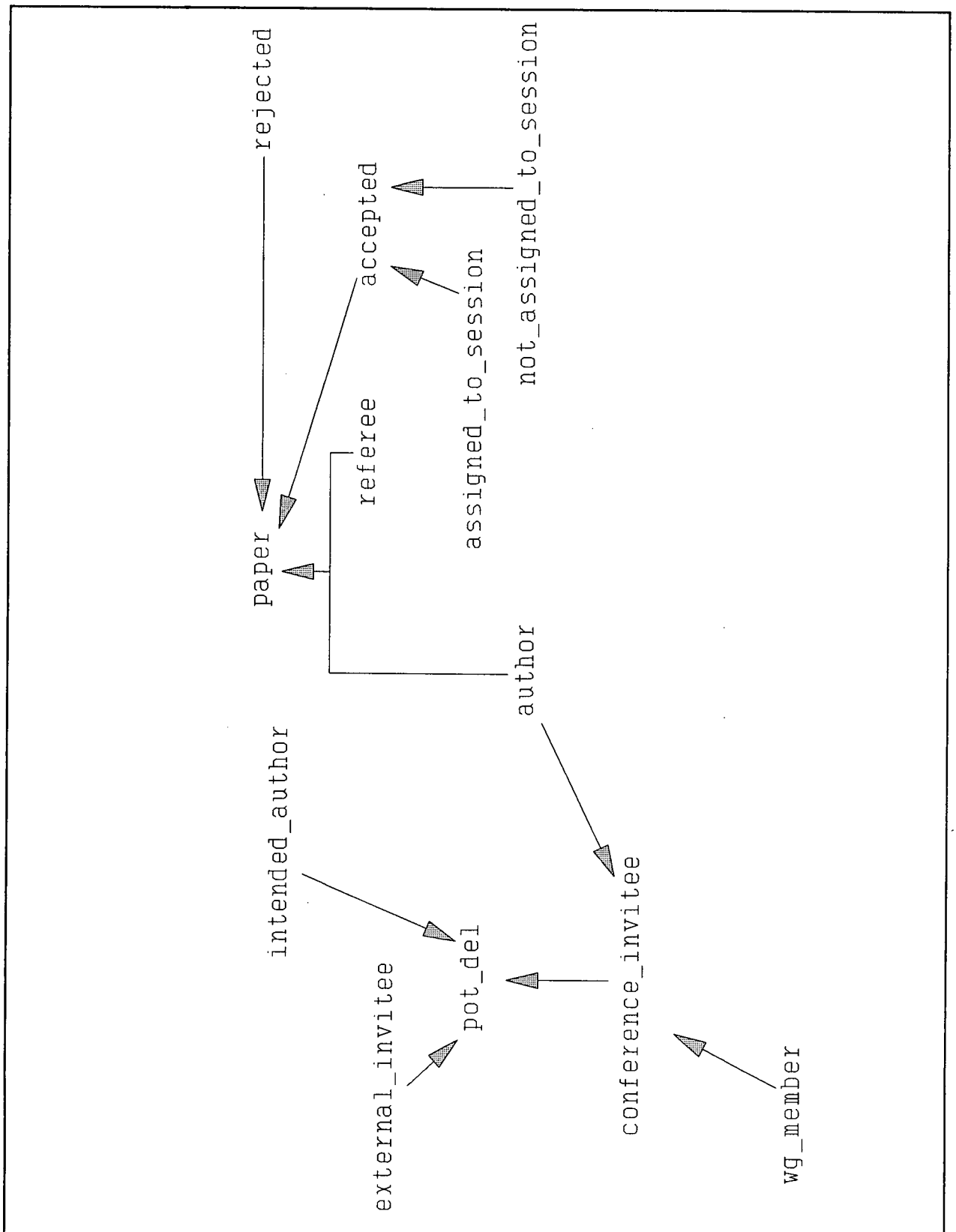


Figure 32: A modified ACM/PCM object scheme for the IFIP Working Conference problem which more closely resembles the structures found using JSD and SELMA.

in ensuring the system model was both complete and consistent.

The decompositions suggested by the specifications analysis tools exposed several problems with the system model as first proposed. The fact that some of the decompositions suggested by the tools were in conflict with intuitive expectations indicated that some information was not included in the model. As well, differences between the suggested decompositions and the decomposition implicit in the defined sublaws showed that some sublaws included redundant information.

One of the decompositions suggested by the tools was compared to the entities and objects identified for the IFIP Working Conference problem by JSD and ACM/PCM. The decompositions produced by all three techniques are illustrated in Figure 33. The entities of JSD are seen to be quite similar to one possible partition of the subsystems contained in the best decomposition. The fact that there are other partitions, means that specifications analysis tools are capable of finding alternatives to the JSD entity structure for the system. Not only can the tools suggest alternatives, but they can automatically determine the sequence of JSD actions, and the information required for successful completion of these actions, given a system model expressed using the SELMA formalism.

The decompositions suggested by the specifications analysis tools were quite different from the object classification hierarchy produced using ACM/PCM. The differences were attributed to the use of world knowledge by the analysts using ACM/PCM. That is, they used knowledge not contained in the IFIP Working Conference system description. When this world knowledge was not modelled, the object hierarchy closely resembled those of JSD and SELMA. Classification hierarchies involving "is-a" and "part-of" relationships are commonly used in the field of artificial intelligence (Borgida, 1981). There they support the modelling of how human beings think. Such hierarchies also support a useful implementation technique by allowing attributes and procedures at higher levels to be automatically "inherited" by lower-level objects. This helps to ensure consistency of the implementation. However, these hierarchies should not be used during the early stages of the analysis and design of non-reasoning systems. There they can overload an analyst with detail and obscure important relationships within a system.

SELMA can be used to identify suitable objects for real-world modelling. However, partitioning the subsystems suggested by the specifications analysis tools to create aggregate objects may not be necessary. The subsystems can be

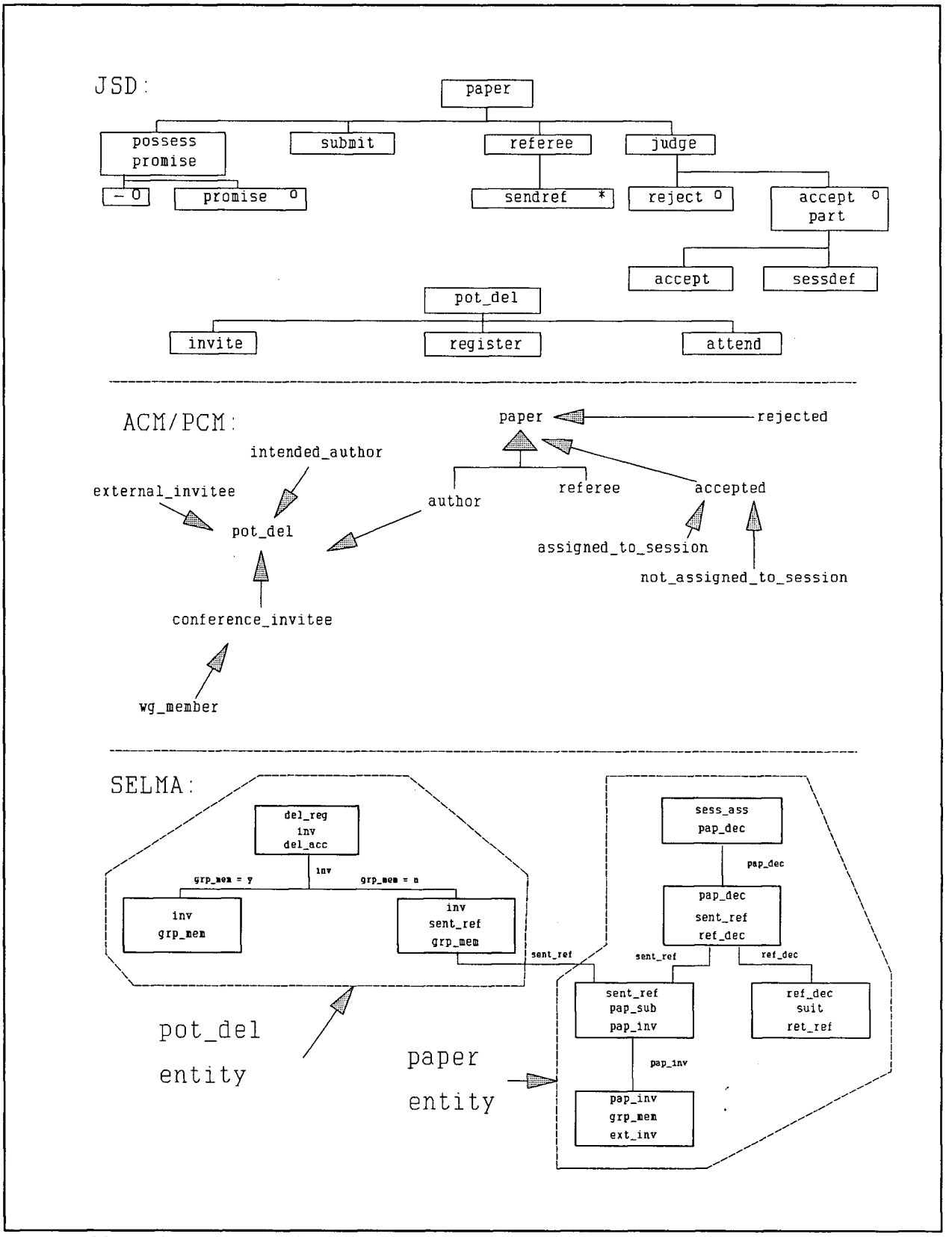


Figure 33: The principal entities and objects identified by JSD, ACM/PCM, and SELMA. The SELMA decomposition has been partitioned as earlier to simplify comparison with JSD and ACM/PCM.

used directly to form a basis for system design. Some critics may argue that SELMA requires too much effort "up front". That is, state variables and events must be identified, and complete and consistent sublaws must be defined. However, all systems analysis and design methodologies require that this work be done in one form or another. They only differ in when it should be done. In particular, completeness and consistency testing must typically wait until after implementation in most methodologies. Even then, such testing is usually done using some sort of trial-and-error test data generation technique. Correcting errors after system implementation is generally acknowledged to be quite expensive. SELMA puts such testing where it belongs: "up front".

Chapter 7: Conclusions and Future Research

7.1. Introduction

This research has described the development of a set of concepts and tools to support the process of systems analysis and design. The specifications analysis tools are "reasoning" in that they incorporate system meta-knowledge in the form of various definitions and heuristics (e.g. consistency, completeness, decomposition heuristics). The tools are qualitatively different from most existing CASE[113] tools, such as Excelerator (Topper, 1987) and Plexsys (Konsynski and Nunamaker, 1982), in that they can aid in the detection of other than purely syntactic system modelling errors. That is, the tools can point to possible semantic errors such as dual roles for a single state variable, and missing information in a model. The tools are also "active" in that they suggest possible decompositions for a system. As was illustrated in Chapter 6, these decompositions can provide the basis of an entity or object structure which is firmly rooted in the explicitly-defined dynamics of the system.

In Chapter 2, the "fuzzy" state of many key concepts in information systems analysis and design was decried. While precise definitions for consistency, completeness and correctness, sequential decomposition, parallel decomposition, conditional decomposition, complexity, and many other terms were given in preceding chapters, some important concepts have yet to be reexamined in the light of the states, events, and laws formalism. These include the concepts of "coupling" and "cohesion" (as introduced in Chapter 1) and of "system", "system statics" and "system dynamics". This chapter will begin by suggesting definitions for these concepts. The conclusions reached at the end of each of the preceding chapters will be briefly recapped with the intent of reaching some sort of synthesis. This will be followed by a description of three areas suitable for further research and development - namely, further development of the specifications analysis tools, and further application of the tools and SELMA to real problems.

---

[113] Computer-Aided Systems Engineering

7.2. Definition of some key modelling concepts

SELMA allows precise definition of some of the key concepts in the field of information systems analysis and design. Many definitions were both intuitively and formally expressed in earlier chapters. A few others will be presented here.

Many important concepts are defined solely in implementation terms. As examples, Myers' taxonomy of coupling and cohesion is largely defined in terms of program code[114], and databases are often equated with system statics and programs with system dynamics. Such definitions are of little use during the early states of the systems analysis and design cycle. SELMA supports some improved definitions for modelling at the conceptual level.

7.2.1. Coupling

There appear to be two basic types of inter-subsystem coupling: BEHAVIOURAL COUPLING and INPUT COUPLING.

Definition: **Behavioural Coupling**

> Two subsystems are behaviourally coupled if the state of one subsystem is dependent on the state of the other.

Definition: **Input Coupling**

> Two subsystems exhibit input coupling if they share the same input state variable.

For example, consider the following decomposition of the modified payroll system.

3: {base,add_pay,total_pay}
2: {com,emp_t,over,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

---

[114] This is not a criticism of Myers' work. He was primarily interested in gauging the quality of computer programs, not conceptual models. Myers' taxonomy is described in Appendix B.

where hours = hours worked pay_r = pay rate
emp_p = employee position emp_t = employee type
sales = amount of sales base = base pay
total_pay = total pay add_pay = additional payments
over = overtime pay com = sales commissions

The subsystems {emp_p,sales,com} and {com,emp_t,over,add_pay} exhibit behavioural coupling in that the output state variable "com" of the lower subsystem is an input to the higher subsystem. The state of the higher subsystem will be dependent on the state of the lower. Behavioural coupling is a fundamental property of parallel/sequential decomposition. This sort of coupling provides the only means of communication between subsystems. Input coupling is more subtle than behavioural coupling. For example, in the decomposition of the modified payroll system show above, the subsystems {hours,pay_r,base} and {emp_p,hours,over} share the state variable "hours". The subsystems are input coupled.

At the conceptual level, only behavioural coupling is important. If new sublaws or events are added to the system, the behaviour of some subsystems may change. Any subsystem behaviourally coupled to the changed subsystem will have to be examined to see if changes are required[115]. Input coupling, on the other hand, is an implementation-level concern. For example, should the format of the common input state variable change, the programmer will have to examine both subsystems for possible corrections. However, if the subsystems are joined only through input coupling, the behaviour of one subsystem cannot affect the behaviour of the other. Therefore, this form of coupling cannot lead to changes in system behaviour which may require changes in a decomposition. In the terms of the system maintenance taxonomy introduced in Chapter 3: All minimal deterministic subsystems will remain minimal deterministic subsystems after an implementation change to a state variable.

Behavioural and input coupling can be mapped into Myers' implementation-oriented coupling taxonomy as follows:

---

[115] As noted in Chapter 4, the specifications analysis tools can help identify the subsystems affected by maintenance operations. Through the use of the system complexity measure, the tools can even help to quantify such effects.

SELMA	Myers
behavioural coupling	data coupling
	control coupling
	content coupling
	common coupling
input coupling	stamp coupling
	common coupling

Common coupling is included in both categories because a program's common area can be used to share input data and to exchange calculated values. By construction, the specifications analysis tools will never suggest decompositions which contain anything like control, content, or common coupling. In Myers' terms, the only form of coupling found in the decompositions suggested by the specifications analysis tools is data coupling. Data coupling is Myers' most desirable form of coupling.

7.2.3. Cohesion

If a subsystem cannot be further decomposed, it is cohesive. There are two degrees of cohesiveness in SELMA: Parallel/sequential cohesiveness and conditional cohesiveness.

Definition: **Parallel/Sequential Cohesion**

> If a subsystem cannot be further decomposed using parallel/sequential decomposition, it exhibits parallel/sequential cohesion.

Definition: **Conditional Cohesion**

> If a subsystem cannot be further decomposed using conditional decomposition, it exhibits conditional cohesion.

A subsystem exhibits parallel/sequential cohesiveness with respect to an entire intermediate state space. This means that if all defined subsystem state transitions are to be considered, no subset of the state variables describing

the subsystem will behave deterministically[116]. Conditional cohesion means that no such deterministic subsets of state variables can be found by partitioning the intermediate state space based on the values of some state variable[117].

Notice that the subsystems suggested by the specifications analysis tools are cohesive only with respect to a given set of state variables, events, and sublaws. Should the model be changed or the level of abstraction reduced[118], some subsystems may no longer be cohesive. For example consider the overtime pay subsystem of the modified payroll system {hours,emp_p,over}. Suppose the system is changed to require separate state variables for ordinary and statutory holiday overtime. It is conceivable that the tools might suggest new subsystems of the form:

{emp_p,ordinary_hours,ordinary_over}
{emp_p,holiday_hours,holiday_over}

That is, the system might be further decomposed so that separate subsystems perform the calculations for the different types of overtime pay.

7.2.3. System, Statics and Dynamics

The concept of "system" was never formally defined in Chapter 2. It was assumed that the systems analyst knew what a system was. This notion can be formally defined as follows:

Definition: **System**

Some part of the real world described by the set of state variables selected by a systems analyst.

---

[116] The subset of state variables would also have to satisfy the parallel/sequential decomposition heuristics described in Chapter 3.

[117] Parallel/sequential and conditional decomposition are thoroughly discussed in Chapters 3 and 5, respectively. Further discussion of their differences with respect to partitions of the intermediate state space is not appropriate here.

[118] That is, the number of state variables, external events, and/or sublaws is increased.

This definition is not meant to imply that systems consist only of state variables. Certainly, relationships between the descriptive state variables are an important part of a system. Still, this definition is not likely to satisfy a majority of ontologists. For example, Mattessich (1978, pp. 29-30) defines a system as a set satisfying the following conditions[119].

1. It contains two or more ELEMENTS with specific PROPERTIES.
2. It contains RELATIONS (connecting the elements of the system with each other) and qualities of those which in turn lend STRUCTURE, HOLISTIC properties, as well as possible REGULATORS to the system enabling also its transformation.
3. It is embedded in an ENVIRONMENT containing additional inter-related elements.
4. The boundaries between the system and its environment are determined by the system's elements and relations, and are sufficiently sharp and permanent to consider the system as an entity.
5. It contains at least one relation between an element of the system and an element of the environment (i.e. it is an open system).
6. It has evolved or been created to tend toward a GOAL.

Elements correspond to groups of SELMA subsystems, properties to state variables, and relations and regulators to sublaws. The system's environment is the set of all state variables not included in the system[120]. As described in Chapter 4, in SELMA, a system's goal is represented by a list of required state variables. Structure refers to a system decomposition, and holistic properties are functions of a decomposition[121]. SELMA supports the concept of a system goal to the extent that some state variables may be indispensable while others may have been included only to facilitate modelling of the system. The

---

[119] Mattessich holds that the first four conditions are "necessary" and the last two describe those systems in which we are mostly likely to be interested.

[120] This set is likely to be large or even infinite. In practice, the environment of a system can likely be restricted to the parts of the real world which can affect the behaviour of that part of the real world being modelled as the system. That is, that part of the real world responsible for the external events which may affect the system.

[121] In Chapter 3, holistic properties were equated with emergent state variables.

set of indispensable state variables defines the system goal in SELMA. Obviously SELMA is not incompatible with more "conventional" concepts of system. The only important difference is that, in SELMA, systems may be identified solely by sets of state variables. That is, subsystems (i.e. elements) are not seen as fundamental parts of a system. They result from adopting a particular system decomposition.

As exemplified by the ACM/PCM methodology, system statics are often taken to include classification hierarchies[122]. As a conceptual-level modelling technique, SELMA requires a much more restrictive definition of system statics.

Definition: **System Statics**

> The state variables (and their possible values) selected to describe a system.

In SELMA, these state variables are selected by considering the functionality of the system. World knowledge, not explicitly included in the system model, is not used. The exclusion of classification hierarchies means that SELMA cannot be used to represent some kinds of knowledge, nor to support implementation-level concerns such as attribute inheritance to ensure consistency. But then, SELMA was never intended to be a generalized knowledge representation language, nor an implementation tool.

In SELMA, the key to understanding or designing a system lies in its dynamics.

Definition: **System Dynamics**

> The manner in which a system changes its state.

State changes are governed by the system law. The system law defines which states are stable, which states are unstable, and what the system does when it finds itself in an unstable state. The state changes which will actually occur are determined by the external events which affect the system. In practice, an analyst uses sublaws and a subset of all possible external events to define a

---

[122] Is-a, part-of and member-of relationships form the classification hierarchies of ACM/PCM.

system model. Specification of system dynamics is usually the most difficult part of constructing a system model.

7.3. Conclusions

This section is included to provide a summary of the various observations and conclusions found elsewhere in this document. The reasoning leading to each conclusion will not be repeated here.

As noted earlier in this chapter, many key concepts for describing systems are vaguely defined. A formal model for describing systems (SELMA) was created by expanding the works of Bunge (1977, 1979) and Wand and Weber (1988, 1989). Its focus on states, events, and laws, rather than on procedures and data, allowed formal definition of many of these previously vague terms.

SELMA also facilitates automated tests of system specification completeness and consistency. These tests, as well as many others[123], have been incorporated in the Prolog-based specifications analysis tools. A formal definition of correctness was also suggested, but ensuring system correctness was seen to be generally impossible[124]. The "no state variable may change twice" rule, as introduced in Chapter 3, helps to ensure semantic integrity by forcing the analyst to define different state variables to represent different system properties.

The importance of the defined set of external events was noted in Chapter 3. The external events determine which of the defined sublaws will be activated during simulation of the system and creation of the system relation. That is, the behaviours exhibited by the system model are determined by the external events. Therefore, because a decomposition of a model is a function of its behaviour, changes in the set of external events may alter the decomposition. This observation has implications for systems design. That is, if the decomposition is to be used as the structure of an implementation, it may only be suitable for the explicitly-defined external events.

In Chapter 1, it was noted that there are only three basic types of decomposition: sequential, parallel, and conditional. However, despite this and

---

[123] The various tests performed by the specifications analysis tools are described in Appendix D.

[124] Correctness required full knowledge of the system law. Such knowledge is unattainable except for purely analytical systems.

the considerable attention focused on the problem at both the conceptual and implementation levels, there existed no automatable rules for actually decomposing a system. The system modelling aspects of SELMA were supplemented by a formal theory describing all three basic types of system decomposition. As mentioned several times in this research, this theory has a heavy focus on system dynamics. Decomposition is based on the behaviour of the system under defined external events and sublaws. "World knowledge" is not included. Decomposition was observed to be quite sensitive to addition and deletion of both external events and sublaws. The implications of this observation for system maintenance were explored in Chapter 4.

The theory of decomposition included a number of heuristics which allowed the development of an algorithm for parallel/sequential and conditional decomposition. This algorithm has been added to the specification analysis tools. The tools can not only test a system model, but can SUGGEST possible decompositions. Therefore, the tools may be described as ACTIVE rather than PASSIVE. Suggested decompositions are suitable for use as the subsystem structure of an implementation. Differences between suggested decompositions and the analyst's intuitive expectations can point to errors or missing information in the system model, which were not detected by the tests of completeness and consistency[125].

An intuitively-justifiable measure of system complexity (i.e. Hellerman's Computational Work) was adopted for use with SELMA. The face validity of the measure was established by tracing its step-by-step development from a simple estimate of complexity (Ashby's variety), and through the use of numerous examples. The measure allowed the specifications analysis tools to present alternative decompositions to the analyst in some sort of meaningful order. The validity of the measure was further enhanced when the simple example payroll systems were analyzed with the tools. The intuitive decompositions of these systems (i.e. those indicated by the defined sublaws) turned out to have the lowest complexities. The measure also allowed the tools to reject a large number of possible decompositions which were derivable from other decompositions through simple state variable substitutions. It was also incorporated in the conditional decomposition heuristic which prevented decomposition from increasing complexity.

---

[125] As shown during the analysis of the IFIP Working Conference problem, the mere fact that multiple decompositions are suggested by the tools may indicate deficiencies in the system model.

The quantitative evaluation of system maintenance strategies is also made possible by the complexity measure. When designing an information system, a designer must decide whether to include rules in the initial system to handle anticipated changes, or to change the system later. The measure allows the designer to choose the course of action with the lowest complexity.

While the measure of system complexity can be a powerful tool for systems analysis and design, it must be used with some degree of caution. The measure corresponds to a sort of "basic" system complexity. No allowances are made for language primitives or existing implemented program modules. For this reason, the tools do not simply inform the analyst of the lowest-complexity decomposition nor maintenance strategy. All possible decompositions can be presented[126]. They are merely ranked by the tools in order of increasing complexity. An analyst must be prepared to override this ranking. Similarly, the simple decision rule for system maintenance, described in Chapter 4, has not been implemented because of the possible inaccuracy of the complexity measure. It may be possible to improve the validity of the measure so that the decision rule can be automated. Such an improvement is one possible future research project.

SELMA was applied to the IFIP Working Conference problem. Only a small part of the effort required to construct the model of the Conference system was described in Chapter 6. Detailed discussion was relegated to a series of appendices. The specifications analysis tools were found to be useful in creating a complete and consistent system model. As well, differences between intuitive expectations and those decompositions suggested by the tools served to identify real-world information which was missing in the system model. Some relationships (i.e. sublaws) in the system model were found to involve more state variables than necessary[127]. The suggested decompositions mapped nicely into the sequential-entity structures produced by the Jackson System Development methodology, but did not compare well with those produced by some practitioners of ACM/PCM. However, correction of an error in the construction of the ACM/PCM model and elimination of implementation-level information, led to a structure similar to that found by SELMA and JSD.

---

[126] How many decompositions are actually presented to the analyst depends on the value of the "maximum percentage difference" parameter used by the tools to limit the search space. Use of this parameter was described in Chapter 4.

[127] That is, the system model as specified by the author was not minimal.

7.4. Future Research

Possible extensions to the research fall into three broad categories: enhancement of the tools, further testing, and extension of the theory of decomposition. Each category is discussed below.

7.4.1. Enhancement of the Specifications Analysis Tools

As currently implemented, the specifications analysis tools are not suitable for use by practising systems analysts. The tools were primarily created as a research tool to test various hypotheses related to the use of SELMA. There are a number of problems with the tools which can probably be reduced or eliminated.

a) The tools do not consider existing software modules and/or implementation language primitives.

The complexity measure currently used to rank the suggested decompositions does not allow for the fact that some subsystems may be very easy to construct. Code may already exist or the implementation language may provide an instruction which can be used to code a complex operation very simply. The tools can likely be enhanced to allow the analyst to include this information in the system model. Use of such information by the tools should increase the validity of alternative decomposition rankings and decisions related to system maintenance. In Chapter 4, validity concerns dictated that application of the simple rule, for deciding whether to include future modification in the initial implementation, be left to the system designer. It may be feasible to fully automate some of the maintenance-related decisions[128].

b) System decomposition using the specifications analysis tools is slow.

---

[128] Of course, other factors besides system complexity and design primitives are likely to influence these maintenance decisions. For example, if the subsystem's programmer will not be available in the future (he or she may have been hired on a temporary basis), it may be more efficient to include the modified subsystem's functions in the first implementation.

While the speed of completeness and consistency testing is acceptable, the speed of the decomposition process is not. The tools required about 3 hours to find the lowest-complexity decomposition of the IFIP Working Conference system[129]. The prototype tools were implemented using Turbo Prolog. Execution speed can be enhanced by coding selected time consuming steps in either Pascal or C. It may also be possible to identify additional heuristics which can be used to judge the quality of a decomposition, or to improve the search algorithm. A measure of computational work (Hellerman, 1972), based on Ashby's (1956) measure of entropy, is already used as a measure of subsystem complexity in an attempt to rank the decompositions suggested by the tools. Additional heuristics based perhaps on the simple coupling and cohesion measures of Myers (1975) might also be incorporated.

c) The specifications analysis tools have a crude user interface.

The interface could be improved to enhance the interaction between the analyst and the tools. In addition to the creation of a logically-organized, menu-driven and window-oriented interface, two specific concerns regarding input requirements and tool outputs must be addressed.

1. Preparation of the input to the computerized tools is tedious.

The tools currently require that a system model be expressed using a syntax similar to Prolog. This syntax is quite unnatural and requires a great deal of redundancy (e.g. repeated clause "names"). A compiler could be created to translate decision tables into the clauses required internally by the tools.

2. The readability of the output from the computerized tools is poor.

The computerized tools currently display possible decompositions in the textual format used throughout this research. This format is difficult

---

[129] The tools were running on an IBM AT clone operating at 7 times the speed of an IBM XT. Today's microcomputer technology is in a constant state of flux. The improving technology will impact the performance of the tools. For example, use of an IBM Model 70 (21.5 times the speed of an IBM XT) should improve the speed of the tools by a factor of three.

to interpret without considerable practice. The readability of the output could be greatly improved if decompositions were displayed in the more conventional graphical form where linkages between subsystems are clear and subordinate subsystems are shown below superordinate subsystems.

7.4.2. Additional Applications of SELMA

Application of SELMA and the specifications analysis tools has been described for several systems. However, most of these systems were quite simple and useful for illustration only. No analyst is likely to use the tools to create a model as simple as the modified payroll system. The only "real" test of SELMA was the IFIP Working Conference problem. More real applications are required to thoroughly test the modelling approach and the tools.

It would also be desirable to test the usefulness of the computerized tools by having real systems analysts use them to analyze real systems. Such testing is likely to reveal additional problems associated with model building and decomposition. One possible problem is discussed in the next section.

7.4.3. Extensions to the Theory of Decomposition

All the systems examined thus far could be analyzed using a single SELMA model. That is, all the systems could be described by fewer than 15 state variables. Large systems will have to be approached using a form of hierarchical analysis. The initial model of the system would be constructed with a sufficiently high level of abstraction that only a few state variables are needed. The subsystems identified for the initial model could then be further analyzed by constructing more detailed sub-models. This procedure could be repeated until the desired degree of detail is reached[130].

Hierarchical analysis is a form of decomposition, but it is not the same as the decomposition addressed by this research. It seems that there are at least two kinds of decomposition:

1) Entity Decomposition: identification of "good" system pieces at a single level of abstraction.

---

[130] This form of analysis is similar to hierarchical decomposition using data flow diagrams (DeMarco, 1979).

2) Hierarchical Analysis: successively increasing the degree of detail with which the system pieces are described.

The techniques presented in this research support entity decomposition. That is, a system model created by the analyst will be analyzed to determine possible functional partitions. These functional partitions consist of deterministic subsystems which may be used individually or combined together to form entities. They support hierarchical decomposition only to the extent that they help to identify the parts of the system to be examined in greater detail at the next level of abstraction. For example, in Chapter 6, the specifications analysis tools were shown to suggest a decomposition (or functional partition) which mapped easily into the entities identified using Jackson System Development (JSD). Presumably, if the JSD analysis had not been performed previously, the decomposition could have been used to identify the set of entities. These entities could be subjected to further, and more detailed, analysis and/or design. More detailed examination of the entities would constitute an application of hierarchical analysis. Hierarchical analysis could be carried out until the dynamics of the system being modelled were described with sufficient detail to satisfy the analyst or to allow easy implementation.

Unfortunately, the decomposition theory described in this research does not explicitly support sub-models at different levels of abstraction. For example, there is currently no way to ensure that sub-models are consistent with each other. The theory would have to be extended to ensure that a sub-model will provide the outputs required at the next higher level, and that it will not conflict with other sub-models[131].

[131] One possible form of conflict would occur when a lower-level model determines the value of a state variable found in a different branch of the hierarchical analysis tree.

References

Albrecht, Allan J., "Measuring Application Development Productivity", Proceedings of the IBM Applications Development Symposium, GUIDE/SHARE, October, 1979, pp. 83-92.

Alexander, Christopher, Notes on the Synthesis of Form, Harvard University Press, Cambridge, Massachusetts, 1967.

Ashby, W. Ross, An Introduction to Cybernetics, Chapman and Hall Ltd., 1956.

Bailey, J.W. and V.R. Basili, "A Meta-Model for Software Development Resource Expenditures", Proceedings of the Fifth International Conference on Software Engineering, IEEE/ACM/NBS, March, 1981, pp. 107-116.

Baker, F.T., "Chief Programmer Team Management of Production Programming", IBM Systems Journal, Vol. 11, No. 1, 1972, pp. 56-73.

Borgida, Alexander, "On the definition of specialization hierarchies for procedures", in Proceedings of the Seventh International Joint Conference on Artificial Intelligence, IJCAI-81, 24-28 August 1981, Vancouver, B.C., Canada.

Borland International, Inc., Turbo Prolog Owner's Handbook, 4584 Scotts Valley Drive, Scotts Valley, CA 95066, USA, copyright 1986.

Brodie, Michael L. and Erico Silva, "Active and Passive Component Modelling: ACM/PCM", in Information Systems Design Methodologies: A Comparative Review, edited by T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, North-Holland Publishing Company, copyright IFIP, 1982, pp. 93-142.

Brooks, Frederick P. Jr., "No Silver Bullet: Essence and Accidents of Software Engineering", Computer, Vol. 14, No. 12, April, 1987, pp. 10-19.

Browder, Sue, The New Age Baby Name Book, Workman Publishing, New York, 1987.

Bubenko, Janis A. Jr., "Information System Methodologies - A Research View", in Information Systems Design Methodologies: Improving the Practice, edited by T.W. Olle, H.G. Sol and A.A. Verrijn-Stuart, Elsevier Science Publishers B.V. (North-Holland), c. IFIP, 1986, pp. 289-317.

Bunge, M. Treatise on Basic Philosophy: Volume 3: Ontology I: The Furniture of the World, D. Reidel Publishing Co.,Boston, Mass., 1977.

Bunge, M. Treatise on Basic Philosophy: Volume 4: Ontology II: The World of Systems, D. Reidel Publishing Co.,Boston, Mass., 1979.

Charniak, Eugene and Drew V. McDermott, Introduction to Artificial Intelligence, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1985.

Chen, Peter, "The Entity-Relationship Model -- Toward a Unified View of Data", Acm Transactions on Database Systems, Vol. 1, No. 1, March, 1976, pp. 9-36.

Courtois, P. J., "On time and space decomposition of complex structures", Communications of the ACM, Vol. 2, No. 6, June, 1985, pp. 590-603.

De Kleer, Johan and John Seely Brown, "A Qualitative Physics Based on Confluences", in Qualitative Reasoning about Physical Systems, ed. Daniel G. Bobrow, MIT Press, Cambridge, Massachusetts, 1985.

DeMarco, Tom, Structured Analysis and System Specification, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979. Copyright 1978 YOURDON, INC.

Garey, Michael R. and David S. Johnson, Computers and Intractability: A Guide to the theory of NP-Completeness, W.H. Freeman and Company, San Francisco, 1979.

Gustafson, M.R., Karlsson, T., and J.A. Bubenko, "A declarative approach to conceptual information modelling", in Information Systems Design Methodologies: A Comparative Review, edited by T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, North-Holland Publishing Company, copyright IFIP, 1982, pp. 93-142.

Halstead, Maurice Howard, Elements of Software Science, Elsevier, New York, 1977.

Hamilton, M. and S. Zeldin, "Higher order software - A methodology for defining software", IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March, 1976, pp. 9-32.

Hellerman, Leo, "A Measure of Computational Work", IEEE Transactions on Computers, Vol. c-21, No. 5, May 1972, pp. 439-446.

Hutchens, David H. and Victor R. Basili, "System structure analysis clustering with data bindings", IEEE Transactions on Software Engineering, SE-11, August, 1985, pp. 749-757.

Jackson, Michael A., System Development, Prentice-Hall International, London, 1983.

Konsynski, Benn R. and Jay F. Nunamaker, "Plexsys: A System Development System", in Advanced System Development/Feasibility Techniques, edited by J. Daniel Couger, Mel A. Colter, and Robert W. Knapp, John Wiley and Sons, New York, 1982, pp. 399-423.

Krishnamurthy, V., Combinatorics: Theory and Applications, John Wiley and Sons, New York, 1986.

Mattessich, Richard, Instrumental Reasoning and Systems Methodology, D. Reidel Publishing Company, Boston, Mass., 1978.

Martin, James, Systems Design from Provably Correct Constructs, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

McNeile, Ashley T., "Jackson System Development (JSD)", in Information Systems Design Methodologies: Improving the Practice, edited by T.W. Olle, H.G. Sol and A.A. Verrijn-Stuart, Elsevier Science Publishers B.V. (North-Holland), c. IFIP, 1986.

Mili, ALi, Desharnais, Jules, and Jean Raymond Gagné, "Formal Models of Stepwise Refinement of Programs", ACM Computing Surveys, Vol. 18, No. 3, September, 1986.

Miller, G.A., "The magical number seven, plus or minus two: Some limitations on our capacity for information processing", Psychological Review, 63(1956), pp. 81-97.

Myers, Glenford J., Reliable Software Through Composite Design, Petrocelli/Charter, New York, 1975.

Myers, Glenford J., Composite/Structured Design, Von Nostrand Reinhold Company, New York, 1978.

Nierstrasz, O. M., "What is the "Object" in Object-oriented Programming?", in Objects and Things, ed. by D. C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, pp. 1-13, March, 1987.

Olivé, Antoni, "DADES: A Methodology for Specification and Design of Information Systems", in Information Systems Design Methodologies: A Comparative Review, edited by T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, North-Holland Publishing Company, copyright IFIP, 1982.

Olivé, Antoni, "Analysis of conceptual and logical models in information systems design methodologies", in Information Systems Design Methodologies: A Feature Analysis, edited by T.W. Olle, H.G. Sol, and C.J. Tully, Elsevier Science Publishers B.V. (North-Holland), copyright IFIP, 1983.

Olle, T. William, "Comparative Review of Information Systems Design Methodologies", in Information System Design Methodologies - A Comparative Review, edited by Olle, T. W., Sol, H. G., and A. A. Verrijn-Stuart, North Holland Publishing Company, 1982.

Orr, Kenneth T., Structured Systems Development, Yourdon Press, New York, 1977.

Page-Jones, M., The Practical Guide to Structured System Design, Yourdon Press, New York, 1980.

Panko, Raymond R., "Analyzing Needs in Individual Offices", ACM Transactions on Office Information Systems, Vol. 2, No. 3, July, 1984, pp. 226-234.

Parnas, D. L., "On the criteria to be used in decomposing systems into modules", Communications of the ACM, Vol. 15, No. 12, December, 1972, pp. 1053-1058.

Pick, James B., Computer Systems in Business, PWS Publishers, Boston, Massachusetts, USA, 1986.

Roman, Gruia-Catalin, "A Taxonomy of Current Issues in Requirements Engineering", Computer, April, 1985, pp. 14-22.

Rubin, Howard A., "Macro-Estimation of Software Development Parameters: The ESTIMACS System", SOFTFAIR - Software Development: Tools, Techniques, and Alternatives, IEEE, July, 1983, pp. 189-198.

Sandewall, E.J., "Heuristic Search: Concepts and Methods", in Artificial Intelligence and Heuristic Programming, edited by N.V. Findler and Bernard Meltzer, American Elsevier Publishing Company, Inc., New York, 1971, pp. 81-100.

Shannon, Claude E., "A Mathematical Theory of Communication", Bell System Technical Journal, 1948, pp. 378-423, 623-659.

Simon, Herbert A., The Sciences of the Artificial, Second Edition, MIT Press, Cambridge, Massachusetts, 1981.

Simon, H. A. and A. Ando, "Aggregation of variables in dynamic systems", Econometrica, Vol. 29, 1961, pp. 111-138.

Steward, Donald V., Software Engineering with Systems Analysis and Design, Brooks/Cole Publishing Company, Monterey, California, 1987.

Stevens, W.P., Myers, G.J. and L.L. Constantine, "Structured Design", IBM Systems Journal, No. 2, 1974, pp. 115-139.

Topper, Andrew, "Excelling with CASE", PC Tech Journal, Vol. 6, No. 8, 1988, pp. 71-79.

Turing, Allan, "On computable numbers, with an application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, Series 2, No. 42, pp. 230-265, and No. 43, pp. 544-546, 1937.

Von Bertalanffy, Ludwig, "The history and status of general systems theory", in Systems Analysis Techniques, ed. by J. Daniel Couger and Robert W. Knapp, John Wiley and Sons Inc., New York, 1974.

Wand, Yair and Ron Weber, "An Ontological Analysis of some Fundamental Information Systems Concepts", Proceedings of the Ninth Conference on Information Systems, Minneaplois, Minnesota, November 30 - December 3, 1988.

Wand, Yair and Ron Weber, "A model of control and audit procedure change in evolving data processing systems", The Accounting Review, (forthcoming January, 1989).

Warnier, Jean Dominique, Logical Construction of Programs, Van Nostrand Reinhold Company, New York, N.Y., 1974.

Weinberg, Gerald M., PL/1 Programming: A Manual of Style, McGraw-Hill, New York, 1970.

Yourdon, Edward and Larry L. Constantine, Structured Design: Fundamentals of Computer Program and Systems Design, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979.

Zissos, D., Problems and Solutions in Logic Design, Oxford University Press, 1979.

Appendix A: The Parable of Hora and Tempus

(from Simon, 1981, pp. 200-202)

Both Hora and Tempus constructed watches consisting of 1,000 parts. Tempus constructed his watches in such a way that if he was interrupted and had to put it down, it immediately fell to pieces and assembly had to begin again. Hora's watches performed precisely the same functions as Tempus', but he designed his to have stable subassemblies of 10 parts each. Ten of these subassemblies could be put together in another stable assembly, and ten of these final assemblies could be put together to form a completed watch. If Hora was interrupted, previously completed subassemblies would not be affected. Now assume that the probability of being interrupted while adding a part to a watch is 0.01. The simple calculation, described below, shows that it will take Tempus on average 4,000 times as long to complete a watch as Hora.

1. Hora must make 111 times as many complete assemblies as Tempus; but
2. Tempus will lose on the average 20 times as much work for each interrupted assembly as Hora (100 parts, on the average, as against 5); and
3. Tempus will complete an assembly only 44 times per million attempts ($(1-0.01)^{1000} = 44 \times 10^{-6}$), while Hora will complete nine out of ten ($(1-0.01)^{10} = 9 \times 10^{-1}$). Hence Tempus will have to make 20,000 as many attempts per completed assembly as Hora ($9 \times 10^{-1} / 44 \times 10^{-6} = 2 \times 10^{4}$). Multiplying these three ratios, we get $1/111 * 20 * 20{,}000 \approx 4{,}000$.

Appendix B: Myers' Taxonomy of Coupling and Cohesion

Coupling

The five forms of program module coupling are defined in order of decreasing desirability. The ranking is Myers' and was derived from experience.

a. Data Coupling - Data Coupling is the passing of data between two modules in the form of subroutine parameters. This is the least undesirable form of coupling.

b. Stamp Coupling - Stamp coupling occurs when entire records are passed between modules as parameters. This form of coupling is undesirable as record formats are subject to change.

c. Control Coupling - Control coupling occurs when one module uses knowledge of the internal operation of another module to control that modules execution. A common example of control coupling is the use of module to handle output of all error messages to the user. Other modules control the execution of the error message module by passing an integer parameter which selects the message to be displayed. The integer parameter used by the calling module represents knowledge of how the error message module operates. This form of coupling is best kept to a minimum.

d. Common Coupling - When modules are linked through common data structures, such a COBOL data division, the relationship is known as "common coupling". Common coupling should be avoided.

e. Content Coupling - Content, or pathological, coupling refers to the obviously poor practice of transferring control from one module to the interior of a loop or decision structure in another module. This form of coupling must never occur.

Cohesion

Myers also defines seven forms of cohesion ranging from strong to weak.

a. Functional Cohesion - In a functionally cohesive module each statement is directed at completing one function. This is the most desirable form of cohesion.

b. Sequential Cohesion - Sequential cohesion occurs if the output of each task serves as input for the next task.

c. Communicational Cohesion - If different tasks perform different functions on the same input or output parameters it is referred to as "communicational cohesion". Modules with this form of cohesion probably should be split.

d. Procedural Cohesion - Procedural cohesion occurs if one statement in a module follows another but no data is passed.

e. Temporal Cohesion - Temporal cohesion refers to the inclusion of tasks within a module because they occur at the same time. Initialization modules commonly exhibit temporal cohesion.

f. Logical Cohesion - If a module includes a series of actions that are related by a non-functional construct it is referred to as "logical cohesion". For example, if a module calculated a series of unrelated employee benefits it could be called logically cohesive.

g. Coincidental Cohesion - The weakest form of cohesion occurs when a large program is arbitrarily cut up into smaller pieces.

Appendix C: The Decomposition Rules of Mili, Desharnais and Gagné.

(from Mili, et al., 1986, pp. 244-253)

1. Sequence Statement Rule

Given the specification R on S, find relations R_1 and R_2 such that

a) $R = R_1 * R_2$, and

b) FOR ALL s, $s \bullet R_1 \subseteq \text{domain}(R_2)$

where $s \bullet R_1$ is the image of s by R_1, and $R_1 * R_2$ is the composition of R_1 and R_2 defined as follows:

$s \bullet R = \{s' \text{ such that } (s,s') \in R\}$

$R * R' = \{(s,s') \text{ such that}$
THERE EXISTS s″ where
$(s,s'') \in R \text{ and } (s'',s') \in R'\}$

The sequence statement rule can be used to decompose a large program specification into two sequentially activated specifications. It requires the programmer to find two relations where the input states of the first are the same as the input states of the original specification, and the output states of the second are the same as the output states of the original specification. The output states of the first relation must also be the input states of the second. For example, consider the following relation and set of program states.

$R = \{(4,0),(3,0),(2,1),(1,1),(0,0)\}$
$S = \{0,1,2,3,4\}$

The intent of this program specification is to classify the sum of all positive integers less than or equal to a given integer as either even or odd (0 for even; 1 for odd). Suppose programs could be found to match the following relations.

$R_1 = \{(4,10),(3,6),(2,3),(1,1),(0,0)\}$
$R_2 = \{(10,0),(6,0),(3,1),(1,1),(0,0)\}$

The first relation find the sum of all smaller positive integers. The second decides whether the result is even or odd. Each of these relations specifies a program which would be less complicated to code than the original. The first requirement is met since

$$R_1 * R_2 = \{(4,10),(3,6),(2,3),(1,1),(0,0)\} * \{(10,0),(6,0),(3,1),(1,1),(0,0)\}$$
$$= \{(4,0),(3,1),(2,0),(1,1),(0,0)\}$$
$$= R$$

The second requirement is also met as the output states of R_1 match the input states of R_2. Notice the state space of R_1 and R_2 does not have to be the same as that of R.

2. **The Alternation Statement Rule**

Given a program specification R on S, find relations R_1 and R_2 such that

a) $R = R_1 \cup R_2$, and
b) $\text{domain}(R_1) \cap \text{domain}(R_2) = \{\}$

The alternation statement rule is used to decompose a program specification into two conditionally executed specifications. The programmer is required to find some predicate t(s), where $s \in \text{domain}(R_1)$, which can be used to split the original relation into two non-intersecting parts. That is,

if $R = \{(s,s')\}$
then $R_1 = \{(s,s')$ such that $t(s)\}$ and
$R_2 = \{(s,s')$ such that not$(t(s))\}$.

If p_1 and p_2 are programs specified by R_1 and R_2, a new, and hopefully simpler, overall program of the following form may be written.

if t then p_1 else p_2

Consider the following example.

$S = \{-4,-1,0,1,2,4\}$
$R = \{(-4,2),(-1,1),(0,0),(1,1),(4,2)\}$

The intent of the relation is to specify a program which takes the absolute value of a number and then find its square root. Obviously, negative numbers are handled differently than positive numbers. The predicate s<0, where s is a program state, could be defined to split R into

$R_1 = \{(-4,2),(-1,1)\}$
$R_2 = \{(0,0),(1,1),(4,2)\}$

These relations meet both of the above requirements. The program matching R_2 will be simpler than the program matching the original relation R.

3. The Iteration Rule

Given a program specification R on S, such that domain(R) = S. If R satisfies the condition of applicability

$$I(\text{range}(R)) * R = I(\text{range}(R))$$

where $I(\text{range}(R)) = \{(s,s)$ such that $s \in \text{range}(R)\}$, then find a relation B such that

a) domain(B) = S - range(R), and
b) B^+ is well founded on S, and
c) $R = B^+ * I(\text{range}(R))$

Where B^+ is the transitive closure of B defined as

$\{(s,s')$ such that
FOR ALL $i \geq 1 : (s,s') \in B^i\}$

and

$$B^1 = B \text{ and for } i>1,\ B^i = B^{i-1} * B$$

and where B^+ is the reflexive transitive closure of B defined as

$$B^+ = B^+ \cup I(S)$$

where

$$I(S) = \{(s,s) \text{ such that } s \in S\}$$

This rule is somewhat difficult to understand. Consider a program where odd and even numbers are mapped into 1 and 0 respectively.

$S = \{4,3,2,1,0\}$
$R = \{(4,0),(3,1),(2,0),(1,1),(0,0)\}$

$range(R) = \{0,1\}$
$I(range(R)) = \{(0,0),(1,1)\}$
$I(range(R)) * R = \{(0,0),(1,1)\} * \{(4,0),(3,1),(2,0),(1,1),(0,0)\}$
$= \{(0,0),(1,1)\}$
$= I(range(R))$

Therefore, the condition of applicability for the iteration rule is satisfied. The trick is to find B. The first condition gives us the domain of B.

$domain(B) = S - range(R) = \{4,3,2\}$

The second condition ensures that when B is iteratively composed with itself, the result will eventually reach the empty set. Try

$B = \{(4,2),(3,1),(2,0)\}$.

This relation indicates that each input state is to be reduced by 2. Choice of B is largely arbitrary. Aside from the rule giving the domain of B, the only other restriction on B is that it "work". The last two conditions guarantee that B "works". Given B, B^+ can be found as shown:

$B^1 = B = \{(4,2),(3,1),(2,0)\}$

$B^2 = B^1 * B = \{(4,2),(3,1),(2,0)\} * \{(4,2),(3,1),(2,0)\}$

$= \{(4,0)\}$

$B^3 = B^2 * B^1 = \{(4,0)\} * \{(4,0)\}$

$= \{\}$

$B^+ = B^1 \cup B^2 \cup B^3 = \{(4,2),(4,0),(3,1),(2,0)\}$

And the reflexive transitive closure of B is

$B^+ = B^+ \cup I(S)$

$= B^+ \cup \{(4,4),(3,3),(2,2),(1,1),(0,0)\}$

$= \{(4,4),(4,2),(4,0),(3,3),(3,1),(2,2),(2,0),(1,1),(0,0)\}$

The last condition is also satisfied as

$B^+ * I(range(R)) = \{(4,4),(4,2),(4,0),(3,3),(3,1),(2,2),(2,0),(1,1),(0,0)\} * \{(0,0),(1,1)\}$

$= \{(4,0),(3,1),(2,0),(1,1),(0,0)\}$

$= R$

B^+ contains all the input/output states of the system generated when all possible input states are iteratively reduced by 2 until the input state is either 0 or 1. If a predicate t(s) is defined as

$t((s)) = true \quad$ if $not(s \in \{0,1\})$

$= false \quad$ if $s \in \{0,1\}$

and b is a program correct with respect to B, then a program, correct with respect to R, could be written

```
while t do b.
```

Appendix D: System Specification Testing

The model tests provided by the specifications analysis tools, and mentioned in Chapter 2, are described in detail in this appendix. Some failures halt the further processing of the model. Other, less serious, failures result only in the generation of a warning message.

1. Syntax Testing

Syntax testing occurs when the text file created by the user is read by the specification analysis tools. The tools ensure that parentheses are matched, there is proper use of capital and small letters, predicate names are spelled correctly, etc. In addition, a simple routine has been written to ensure the following:

a. State variable names used in the values(), static(), dynamic() and event() predicates are the same as are defined in the state_variable() predicate.

b. Values used in the static(), dynamic() and event() predicates are the same as are defined in the values() predicate.

2. Stability Condition Testing

In order to avoid having to generate all possible combinations of values for the state variables of a system (i.e. the possible state space), sublaws defined for use with the specification analysis tools must follow certain rules.

a. All state variables must be referred to in at least one stability condition rule.

b. If two state variables are referred to in one stability condition rule, and there exists other stability condition rule referring to those state variables, and the value of one state variable is the same in both rules, then the value of the other state variable must be the same in both rules.

The first rule ensures that the sublaws define the stable state space of the entire system. The specifications analysis tools scan the model to ensure that each state variable is mentioned in at least stability condition rule. If this test fails, no stable states can be generated. This results from the assumption that no stable state may include a state variable with an undefined value. If the tools are aware of a state variable not mentioned in any stability condition rule, that state variable cannot be assigned a value in any stable state. If this rule is violated, the specifications analysis tools will issue an error message.

The second rule is a direct consequence of the AND and OR combination rules for stability condition rules. Clauses with the same or different names are combined using inclusive OR or AND, respectively. For example, stability condition clauses of the form

```
static("Law 1",[v(a,0),v(b,0)]).
```

and

```
static("Law 2",[v(....),v(a,0),v(b,1),...]).
```

Assuming no other "Law 1" or "Law 2" clauses refer to both "a" and "b", the clauses are said to CONFLICT. The first rule asserts that whenever "a" has a value of "0", "b" will have a value of "0". The second rule asserts that there is a circumstance where this is not so. One of the rules is incorrect.

3. State Variable Variation

The model may be tested to ensure that every state variable reaches each of its defined values in either the stable state space of the system or in some unstable state in a response path. If an unused state variable value is defined there is evidence that either the sublaws are incomplete or the defined state variable values describe a model which is richer than is required. This test is carried out during the generation of the system's response paths. Consider the four-lights example defined in Chapter 2. Suppose the clause declaring the allowed values for state variable "d" included a value of "dummy", as shown below:

```
values(d,[0,1,dummy]).
```

After the specification analysis tools establish the system's response paths, a warning of the following form will be displayed.

```
** Warning **
** State variable 'd' is never assigned the following value(s):
   ["dummy"]
```

4. Local Completeness

Every unstable state, created by applying an external event to a stable state, must transform to a stable state by applying the defined sublaws. If the specifications analysis tools find no response path for an unstable state, an error message is sent to the user. For example, suppose the last corrective action rule in the four light example of Chapter 2 were entered as follows:

```
dynamic("D3",[v(a,1)],[v(d,0)]).
```

When the specification analysis tools attempt to establish the system's response paths, an error message of the following form will be displayed.

```
** ERROR **
** Initial state
      [0,0,1,0]
   does not transform to a final stable state under event 'E1'.
```

5. Local Consistency

The specifications analysis tools will search for all response paths leading to final stable states for every stable state of the system. All of these final stable states must be the same or an error message will be sent to the user. For example, consider adding the following clause to the last corrective action rule in the four light example of Chapter 2:

```
dynamic("D3",[v(a,"1")],[v(d,"0")]).
```

When the specification analysis tools attempt to establish the system's response paths, error messages of the following form will be displayed.

```
** ERROR **
** The system is not deterministic under event 'E2'.
** Initial state
      [1,1,0,1]
   transforms to final states:
      [0,0,1,0]
      [0,0,1,1]

** ERROR **
** The system is not deterministic under event 'E2'.
** Initial state
      [1,1,1,1]
   transforms to final states:
      [0,0,1,0]
      [0,0,1,1]
```

Appendix E: The Stable State Space and System Law

The specifications analysis tools examine the stability condition rules to determine a system's stable state space. A list of stable states is stored in the Prolog database. These states are only "locally stable" in that the "true" system law is not known. They are stable only with respect to the defined sublaws. Any state the system reaches, either as a result of the actions of an external event or a sublaw, which is not one of these states will be "locally unstable" or simply "unstable".

Once the stable state space of the system has been determined, the specifications analysis tools examine the external events and sublaws to find all the response paths. Each defined external event is applied to every stable state of the system creating a new set of states. Some of these states will be unstable (i.e. not stable). The corrective action rules of the various sublaws are used to transform each unstable state into a stable state. Tests for sublaw consistency and completeness, as described in Chapter 2 and in Appendix D, are performed while response paths leading to stable states are found. The search for all the paths (i.e. sequences of corrective action rule activations) leading to stable states is worse than NP-Complete. The problem of locating a single path leading to a stable state is equivalent to the NP-Complete travelling salesman problem (Garey and Johnson, p. 18-23). The problem at hand is worse because all paths must be found in order to verify the consistency of the sublaws. In the worst case, the number of intermediate states which must be examined during the search is O(N!) where N is the number of corrective action rules in the model (Figure 34 shows the search tree for three rules). Determination of system response paths (or sequences of rule activations) is simplified by the following assumption:

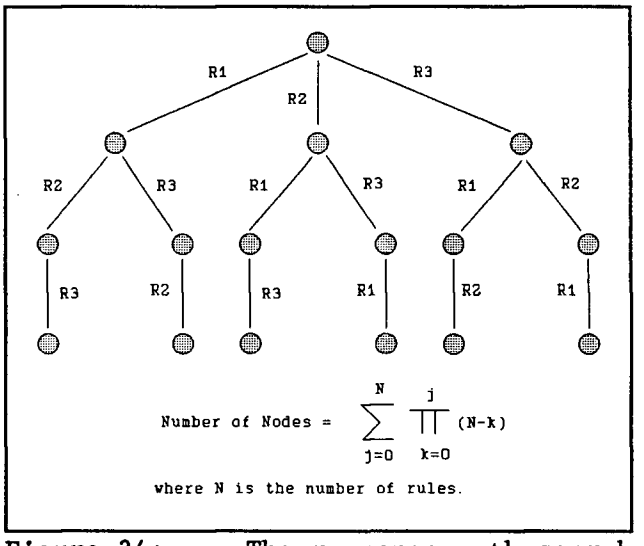


Figure 34: The response path search tree for a system described by three corrective action rules (R1, R2, and R3).

> No corrective action rule may be activated unless the value of one of the state variables in its conditions list has changed as the result of either an external or internal event.

This might be described as the "causality assumption". It ensures that rules are not activated spontaneously. The assumption is not critical for the simple examples described thus far, but is quite important for the IFIP Working Conference example described in Chapter 6. Without the assumption, determination of the system law for this larger example would take an unacceptably long time. In addition to this rule, sublaws are not allowed to be activated twice in a given response path. This effectively eliminates the possibility of loops and helps to ensure the decidability of the response path search problem[132]. These two restriction on the form of a response path greatly reduce the number of paths which must be examined. There are 25 corrective action rules in the IFIP Working Conference system model described in Chapter 6. This means that in the worst case 4.22×10^{25} states must be examined[133]. In fact only 1930 are tested for stability.

As demonstrated in Chapter 3, knowledge of the final stable state corresponding to each unstable state, is of primary importance when automating system decomposition. For this reason the specifications analysis tools store the first system relation[134] in the Prolog database.

---

[132] Turing (1936) showed that it is impossible to specify any algorithm which, given an arbitrary computer program and an arbitrary input to that program, can decide whether that program will eventually halt.

[133] In the worst case any sublaw may be activated from any state of the system. The number of states W which must be tested for stability in this case is

$$W = \sum_{j=0}^{N} \prod_{k=0}^{j} (N - k)$$

$$= \sum_{j=0}^{25} \prod_{k=0}^{j} (25 - k)$$

$$= 4.22 \times 10^{25}$$

where N is the number of corrective action rules.

[134] The first system relation consists of the set of initial unstable states, created by the application of the external events to the stable state space, and the corresponding final stable states. This relation is the

Appendix F: A Simple "Batch" Payroll System

```
/*****************************************************

  A Payroll System

*****************************************************/

clauses

/* Event Definitions */

event("End of Period",[v(end,"1")]).
event("Beginning of Period",[v(end,"0")]).

/* State Variable Definitions */

state_variable(end).        /* end of period flag */
state_variable(emp_t).      /* employee type */
state_variable(emp_p).      /* employee position */
state_variable(pay_r).      /* pay rate */
state_variable(hours).      /* hours worked */
state_variable(sales).      /* sales */
state_variable(base).       /* base pay */
state_variable(over).       /* overtime pay */
state_variable(com).        /* sales commissions */
state_variable(ben).        /* benefits */
state_variable(total_pay). /* total pay */

/* State Variable Values */

values(end,["0","1"]).      /* beginning of period, eop */
values(emp_t,[o,s]).        /* office and sales */
values(emp_p,[r,m]).        /* regular and management */
```

---

approximation of the system law used by the specifications analysis tools to perform decomposition.

```
values(hours,["0",reg,ot]). /* zero, normal or overtime */
values(pay_r,["0",nz]).     /* zero or positive */
values(sales,["0",nz]).
values(base,["0",nz]).
values(over,["0",nz]).
values(com,["0",nz]).
values(ben,["0",nz]).
values(total_pay,["0",nz]).

/* Stability Conditions */

/* Base salary, overtime, commissions and benefits are not
   calculated except at EOP. */
static("EOP requirements",[v(end,"0")]).
static("EOP requirements",[v(end,"1")]).

/* an employee may be in a regular or a management position */
static("management or regular",[v(emp_p,r)]).
static("management or regular",[v(emp_p,m)]).

/* an employee may have either an office or a sales job */
static("office or sales",[v(emp_t,o)]).
static("office or sales",[v(emp_t,s)]).

/* hours may be zero or not zero */
static("hours",[v(hours,ot)]).
static("hours",[v(hours,reg)]).
static("hours",[v(hours,"0")]).

/* pay rate may be zero or not zero */
static("pay rate",[v(pay_r,nz)]).
static("pay rate",[v(pay_r,"0")]).

/* sales may be zero or not zero */
static("sales",[v(sales,nz)]).
static("sales",[v(sales,"0")]).
```

```
/* non-management office staff is entitled to overtime pay if
   hours is not zero */
static("non-management office staff gets overtime",
    [v(end,"1"),v(emp_t,o),v(emp_p,r),v(hours,ot),v(over,nz)]).
static("non-management office staff gets overtime",
    [v(end,"1"),v(hours,reg),v(over,"0")]).
static("non-management office staff gets overtime",
    [v(end,"1"),v(hours,"0"),v(over,"0")]).
static("non-management office staff gets overtime",
    [v(end,"1"),v(emp_t,s),v(over,"0")]).
static("non-management office staff gets overtime",
    [v(end,"1"),v(emp_p,m),v(over,"0")]).
static("non-management office staff gets overtime",
    [v(end,"0"),v(over,"0")]).

/* non-management sales staff is entitled to commissions if
   sales is not zero */
static("non-management sales staff gets commissions",
    [v(end,"1"),v(emp_t,s),v(emp_p,r),v(sales,nz),v(com,nz)]).
static("non-management sales staff gets commissions",
    [v(end,"1"),v(sales,"0"),v(com,"0")]).
static("non-management sales staff gets commissions",
    [v(end,"1"),v(emp_t,o),v(com,"0")]).
static("non-management sales staff gets commissions",
    [v(end,"1"),v(emp_p,m),v(com,"0")]).
static("non-management sales staff gets commissions",
    [v(end,"0"),v(com,"0")]).

/* all employees are entitled to base pay if hours and pay rate
   are not zero */
static("everyone gets base pay",
    [v(end,"1"),v(hours,ot),v(pay_r,nz),v(base,nz)]).
static("everyone gets base pay",
    [v(end,"1"),v(hours,reg),v(pay_r,nz),v(base,nz)]).
static("everyone gets base pay",
```

```
    [v(end,"1"),v(hours,"0"),v(base,"0")]).
static("everyone gets base pay",
    [v(end,"1"),v(pay_r,"0"),v(base,"0")]).
static("everyone gets base pay",
    [v(end,"0"),v(base,"0")]).

/* benefits must be calculated at EOP */
static("benefits",[v(end,"1"),v(ben,nz)]).
static("benefits",[v(end,"0"),v(ben,"0")]).

/* total pay must be calculated at EOP */
static("total pay",[v(end,"1"),v(base,nz),v(total_pay,nz)]).
static("total pay",[v(end,"1"),v(over,nz),v(total_pay,nz)]).
static("total pay",[v(end,"1"),v(com,nz),v(total_pay,nz)]).
static("total pay",[v(end,"1"),v(base,"0"),v(over,"0"),v(com,"0"),
                    v(total_pay,"0")]).
static("total pay",[v(end,"0"),v(total_pay,"0")]).

/* Corrective Actions */

/* At start of period all calculated values must be reset to zero. */
dynamic("SOP",[v(end,"0")],
        [v(over,"0"),v(com,"0"),v(base,"0"),v(ben,"0"),
         v(total_pay,"0")]).

/* calculate base pay at EOP */
dynamic("calculate base pay",
         [v(end,"1"),v(hours,ot),v(pay_r,nz)],
         [v(base,nz)]).
dynamic("calculate base pay",
         [v(end,"1"),v(hours,reg),v(pay_r,nz)],
         [v(base,nz)]).
dynamic("calculate base pay",
         [v(end,"1"),v(hours,"0")],
         [v(base,"0")]).
dynamic("calculate base pay",
```

```
        [v(end,"1"),v(pay_r,"0")],
        [v(base,"0")]).

/* calculate overtime at EOP */
dynamic("calculate overtime",
        [v(end,"1"),v(emp_t,o),v(emp_p,r),v(hours,ot)],
        [v(over,nz)]).
dynamic("calculate overtime",
        [v(end,"1"),v(hours,reg)],
        [v(over,nz)]).
dynamic("calculate overtime",
        [v(end,"1"),v(hours,"0")],
        [v(over,"0")]).

/* calculate commissions at EOP */
dynamic("calculate commissions",
        [v(end,"1"),v(emp_t,s),v(emp_p,r),v(sales,nz)],
        [v(com,nz)]).
dynamic("calculate commissions",
        [v(end,"1"),v(sales,"0")],
        [v(com,"0")]).

/* calculate benefits at the EOP */
dynamic("calculate benefits",
        [v(end,"1")],
        [v(ben,nz)]).

/* calculate total pay */
dynamic("calculate total pay",
        [v(base,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(over,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(com,nz)],
```

```
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(base,"0"),v(over,"0"),v(com,"0")],
        [v(total_pay,"0")]).
```

Appendix G: A Simple "Interactive" Payroll System

```
/*****************************************************

  A Payroll System

  This version defines several events and does not
  user the state variable "end".

*****************************************************/

clauses

/* Event Definitions */

event("work lots",[v(hours,ot)]).
event("work some",[v(hours,reg)]).
event("no work",[v(hours,"0")]).
event("sales",[v(sales,nz)]).
event("no sales",[v(sales,"0")]).

/* State Variable Definitions */

state_variable(emp_t).      /* employee type */
state_variable(emp_p).      /* employee position */
state_variable(pay_r).      /* pay rate */
state_variable(hours).      /* hours worked */
state_variable(sales).      /* sales */
state_variable(base).       /* base pay */
state_variable(over).       /* overtime pay */
state_variable(com).        /* sales commissions */
state_variable(ben).        /* benefits */
state_variable(total_pay). /* total pay */

/* State Variable Value Definitions */
```

```
values(emp_t,[o,s]).              /* office and sales */
values(emp_p,[r,m]).              /* regular and management */
values(hours,["0",reg,ot]).       /* zero, normal or overtime */
values(pay_r,["0",nz]).           /* zero or positive */
values(sales,["0",nz]).
values(base,["0",nz]).
values(over,["0",nz]).
values(com,["0",nz]).
values(ben,[nz]).
values(total_pay,["0",nz]).

/* Stability Conditions */

/* an employee may be in a regular or a management position */
static("management or regular",[v(emp_p,r)]).
static("management or regular",[v(emp_p,m)]).

/* an employee may have either an office or a sales job */
static("office or sales",[v(emp_t,o)]).
static("office or sales",[v(emp_t,s)]).

/* hours may be zero or not zero */
static("hours",[v(hours,ot)]).
static("hours",[v(hours,reg)]).
static("hours",[v(hours,"0")]).

/* pay rate may be zero or not zero */
static("pay rate",[v(pay_r,nz)]).
static("pay rate",[v(pay_r,"0")]).

/* sales may be zero or not zero */
static("sales",[v(sales,nz)]).
static("sales",[v(sales,"0")]).

/* benefits must be calculated */
static("benefits",[v(ben,nz)]).
```

```
/* non-management office staff is entitled to overtime
   pay if hours is not zero */
static("non-management office staff gets overtime",
    [v(emp_t,o),v(emp_p,r),v(hours,ot),v(over,nz)]).
static("non-management office staff gets overtime",
    [v(hours,reg),v(over,"0")]).
static("non-management office staff gets overtime",
    [v(hours,"0"),v(over,"0")]).
static("non-management office staff gets overtime",
    [v(emp_t,s),v(over,"0")]).
static("non-management office staff gets overtime",
    [v(emp_p,m),v(over,"0")]).

/* non-management sales staff is entitled to commissions if
   sales is not zero */
static("non-management sales staff gets commissions",
    [v(emp_t,s),v(emp_p,r),v(sales,nz),v(com,nz)]).
static("non-management sales staff gets commissions",
    [v(sales,"0"),v(com,"0")]).
static("non-management sales staff gets commissions",
    [v(emp_t,o),v(com,"0")]).
static("non-management sales staff gets commissions",
    [v(emp_p,m),v(com,"0")]).

/* all employees are entitled to base pay if hours and
   pay rate are not zero */
static("everyone gets base pay",
    [v(hours,ot),v(pay_r,nz),v(base,nz)]).
static("everyone gets base pay",
    [v(hours,reg),v(pay_r,nz),v(base,nz)]).
static("everyone gets base pay",
    [v(hours,"0"),v(base,"0")]).
static("everyone gets base pay",
    [v(pay_r,"0"),v(base,"0")]).
```

```
/* total pay must be calculated at EOP */
static("total pay",[v(base,nz),v(total_pay,nz)]).
static("total pay",[v(over,nz),v(total_pay,nz)]).
static("total pay",[v(com,nz),v(total_pay,nz)]).
static("total pay",[v(base,"0"),v(over,"0"),v(com,"0"),
                    v(total_pay,"0")]).

/* Corrective Actions */

/* calculate base pay at EOP */
dynamic("calculate base pay",
         [v(hours,ot),v(pay_r,nz)],
         [v(base,nz)]).
dynamic("calculate base pay",
         [v(hours,reg),v(pay_r,nz)],
         [v(base,nz)]).
dynamic("calculate base pay",
         [v(hours,"0")],
         [v(base,"0")]).
dynamic("calculate base pay",
         [v(pay_r,"0")],
         [v(base,"0")]).

/* calculate overtime at EOP */
dynamic("calculate overtime",
         [v(emp_t,o),v(emp_p,r),v(hours,ot)],
         [v(over,nz)]).
dynamic("calculate overtime",
         [v(hours,reg)],
         [v(over,"0")]).
dynamic("calculate overtime",
         [v(hours,"0")],
         [v(over,"0")]).

/* calculate commissions at EOP */
dynamic("calculate commissions",
```

```
        [v(emp_t,s),v(emp_p,r),v(sales,nz)],
        [v(com,nz)]).
dynamic("calculate commissions",
        [v(sales,"0")],
        [v(com,"0")]).

/* calculate total pay */
dynamic("calculate total pay",
        [v(base,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(over,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(com,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(base,"0"),v(over,"0"),v(com,"0")],
        [v(total_pay,"0")]).
```

Appendix H: Decomposition of the Four-Lights System

The decomposition algorithm is applied to the four-lights system described in Chapters 2 and 3. Each step is labelled using the following convention.

x(Ay|Lz)

where

x = Step number starting with 1 and increasing by 1 until the algorithm finishes.

y = Algorithm step number.

z = The current level of recursion with respect to the Decompose() procedure.

START

Given: The first intermediate state space and corresponding final stable states.

This information may be obtained by applying the events "set a=1" and "set a=0" to each of the four stable states. This yields the first intermediate state space or R_1. The final stable states corresponding to each of the states in the first intermediate state space are obtained by examining the response paths of the system (These paths follow directly from the sublaws and are listed in Chapter 2.).

First Intermediate State Space					Corresponding final stable state			
a	**b**	**c**	**d**	--->	**a**	**b**	**c**	**d**
0	0	1	0		0	0	1	0
0	0	1	1		0	0	1	1
0	1	0	1		0	0	1	1
0	1	1	1		0	0	1	1
1	0	1	0		1	1	1	1
1	0	1	1		1	1	1	1
1	1	0	1		1	1	0	1
1	1	1	1		1	1	1	1

1(A1|L1) Find the output state variables with respect to the current intermediate state space.

The only state variables which change their values between the first intermediate state space and the corresponding final stable states are {b,c,d}.

2(A2|L1) The set of output state variables is not empty.

3(A3|L1) Find the good subsystems.

The smallest good subsystems, with respect to the first intermediate state space, which are described by at least one output state variable are {a,$\underline{b}$}, {a,c,$\underline{c}$}, and {a,d,$\underline{d}$}.

4(A4|L1) Find subsets of the good subsystems for intermediate state space update.

The subsets of this set of good subsystems are
{{a,$\underline{b}$}} {{a,c,$\underline{c}$}} {{a,d,$\underline{d}$}}
{{a,$\underline{b}$},{a,c,$\underline{c}$}} {{a,$\underline{b}$},{a,d,$\underline{d}$}} {{a,c,$\underline{c}$},{a,d,$\underline{d}$}}
{{a,$\underline{b}$},{a,c,$\underline{c}$},{a,d,$\underline{d}$}}

5(A5|L1) Update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The first intermediate state space will be eventually updated using all the sets found in step 5. The first set selected is {{a,$\underline{b}$}}. The second intermediate state space created by this update is as shown below.

First ISS		Second ISS		Corresponding final stable states
a b c d	-->	a b c d	-->	a b c d
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 1 0 1		0 0 0 1		0 0 1 1
0 1 1 1		0 0 1 1		0 0 1 1
1 0 1 0		1 1 1 0		1 1 1 1
1 0 1 1		1 1 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1
1 1 1 1		1 1 1 1		1 1 1 1

6(A1|L2) Find the output state variables with respect to the current intermediate state space.

The only state variables with values which change between the second intermediate state space and the corresponding final stable states are {c,d}.

7(A2|L2) The set of output state variables is not empty.

8(A3|L2) Find the good subsystems with respect to the second intermediate state space.

The only good subsystems described by at least one output state variable and one output state variable from the subsystems used in the last update, are {b,c,$\underline{c}$} and {b,d,$\underline{d}$}. Notice that while {a,c,$\underline{c}$} and {a,d,$\underline{d}$} are still good subsystems, they are not described by an output state variable from the subsystems used to create the second intermediate state space (i.e. they are not described by state variable b).

9(A4|L2) Find subsets of the good subsystems for intermediate state space update.

The subsets of this set of good subsystems are
{{b,c,$\underline{c}$},{b,d,$\underline{d}$}} {{b,c,$\underline{c}$}} {{b,d,$\underline{d}$}}

10(A5|L2) Update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The second intermediate state space will be eventually updated using all the sets found in step 9. The first set selected is {{b,c,$\underline{c}$},{b,d,$\underline{d}$}}. The third intermediate state space created by this update is as shown below.

Second ISS		Third ISS		Corresponding final stable states
a b c d	-->	**a b c d**	-->	**a b c d**
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 0 0 1		0 0 1 1		0 0 1 1
0 0 1 1		0 0 1 1		0 0 1 1

1 1 1 0	1 1 1 1	1 1 1 1
1 1 1 1	1 1 1 1	1 1 1 1
1 1 0 1	1 1 0 1	1 1 0 1
1 1 1 1	1 1 1 1	1 1 1 1

11(A1|L3) Find the outputs with respect to the current intermediate state space.

There are no output state variables.

12(A2|L3) Since there are no output state variables, output a good decomposition.

The sets of subsystems used to transform the first intermediate state space into a stable states defines a decomposition. The second intermediate state space was formed using {a,b}. The third was formed using {b,c,c} and {b,d,d}. Therefore, the first discovered decomposition is therefore

2: {b,c,c} {b,d,d}

1: {a,b}

After the Decompose() procedure returns from level 3, execution continues with the next suitable set of subsystems for update at level 2. Notice that the current intermediate state space is again the second intermediate state space found in step 5.

13(A5|L2) Update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The next set of good subsystems is described by {{b,c,c}}. The third intermediate state space created by this update is as shown below.

Second ISS		Third ISS		Corresponding final stable states
a b c d	-->	**a b c d**	-->	**a b c d**
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 0 0 1		0 0 1 1		0 0 1 1
0 0 1 1		0 0 1 1		0 0 1 1

1 1 1 0	1 1 1 0	1 1 1 1
1 1 1 1	1 1 1 1	1 1 1 1
1 1 0 1	1 1 0 1	1 1 0 1
1 1 1 1	1 1 1 1	1 1 1 1

14(A1|L3) Find the output state variables with respect to the current intermediate state space.

The only state variable with values which change between the second intermediate state space and the corresponding final stable states is {d}.

15(A2|L3) The set of output state variables is not empty.

16(A3|L3) Find the good subsystems with respect to the current intermediate state space.

There are no good subsystems which meet the criteria for selection by the GoodSubsystems() function. Notice that although {b,d,$\underline{d}$} describes a good subsystem, it is not include an output state variable from the subsystems used to update the second intermediate state space.

17(A4|L3) Since the are no suitable good subsystems, there can be no suitable sets for updating.

Execution of the algorithm will continue with the iteration over the sets of subsystems suitable for updating found at level 2.

18(A5|L2) Update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The next set of subsystems suitable for updating is described by {b,d,$\underline{d}$}. The third intermediate state space created by this update is as shown below.

Second ISS		Third ISS		Corresponding final stable states
a b c d	-->	**a b c d**	-->	**a b c d**
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 0 0 1		0 0 0 1		0 0 1 1

0 0 1 1	0 0 1 1	0 0 1 1
1 1 1 0	1 1 1 1	1 1 1 1
1 1 1 1	1 1 1 1	1 1 1 1
1 1 0 1	1 1 0 1	1 1 0 1
1 1 1 1	1 1 1 1	1 1 1 1

19(A1|L3) Find the output state variables with respect to the current intermediate state space.

The only state variable with values which change between the second intermediate state space and the corresponding final stable states is {d}.

20(A2|L3) The set of output state variables is not empty.

21(A3|L3) Find the good subsystems with respect to the current intermediate state space.

There are no good subsystems which meet the criteria for selection by the GoodSubsystems() function. Notice that although {b,c,c} describes a good subsystem, it is not include an output state variable from the subsystems used to update the second intermediate state space.

22(A4|L3) Since the are no suitable good subsystems, there can be no suitable sets for updating.

Since there are no more sets for updating at level 2, execution of the algorithm will continue with the iteration over the sets of subsystems suitable for updating found at level 1.

23(A5|L1) Update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The next set of good subsystems is described by {{a,c,c}}. The second intermediate state space created by this update is as follows:

First ISS		Second ISS		Corresponding final stable states
a b c d	-->	a b c d	-->	a b c d
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 1 0 1		0 1 1 1		0 0 1 1
0 1 1 1		0 1 1 1		0 0 1 1
1 0 1 0		1 0 1 0		1 1 1 1
1 0 1 1		1 0 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1

24(A1|L2) Find the output state variables with respect to the current intermediate state space.

The only state variables with values which change between the second intermediate state space and the corresponding final stable states are {b,d,$\underline{d}$}.

25(A2|L2) The set of output state variables is not empty.

26(A3|L2) Find the good subsystems with respect to the second intermediate state space.

There are no good subsystems which meet the criteria for selection by the GoodSubsystems() function.

27(A4|L3) Since the are no suitable good subsystems, there can be no suitable sets for updating.

Execution of the algorithm will continue with the iteration over the sets of subsystems suitable for updating found at level 1.

28(A5|L1) Update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The next set of good subsystems is described by {{a,d,$\underline{d}$}}. The second intermediate state space created by this update is as shown below.

First ISS		Second ISS		Corresponding final stable states
a b c d	-->	**a b c d**	-->	**a b c d**
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 1 0 1		0 1 0 1		0 0 1 1
0 1 1 1		0 1 1 1		0 0 1 1
1 0 1 0		1 0 1 1		1 1 1 1
1 0 1 1		1 0 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1
1 1 1 1		1 1 1 1		1 1 1 1

29(A1|L2) Find the output state variables with respect to the current intermediate state space.

The only state variables with values which change between the second intermediate state space and the corresponding final stable states are {b,c,c}.

30(A2|L2) The set of output state variables is not empty.

31(A3|L2) Find the good subsystems with respect to the second intermediate state space.

There are no good subsystems which meet the criteria for selection by the GoodSubsystems() function.

32(A4|L2) Since the are no suitable good subsystems, there can be no suitable sets for updating.

Execution of the algorithm will continue with the iteration over the sets of subsystems suitable for updating found at level 1.

33(A5|L1) There are update possibilities so update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The next set of good subsystems is described by {{a,b},{a,c,c}}. The second intermediate state space created by this update is as shown below.

First ISS		Second ISS		Corresponding final stable states
a b c d	-->	a b c d	-->	a b c d
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 1 0 1		0 0 1 1		0 0 1 1
0 1 1 1		0 0 1 1		0 0 1 1
1 0 1 0		1 1 1 0		1 1 1 1
1 0 1 1		1 1 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1
1 1 1 1		1 1 1 1		1 1 1 1

34(A1|L2) Find the output state variables with respect to the current intermediate state space.

The only state variable with a value which changes between the second intermediate state space and the corresponding final stable states is {d}.

35(A2|L2) The set of output state variables is not empty.

36(A3|L2) Find the good subsystems with respect to the second intermediate state space.

The only good subsystem described by at least one output state variable, and one output state variable from the subsystems used in the last update, is {b,d,$\underline{d}$}.

37(A4|L2) Find subsets of the good subsystems for intermediate state space update.

There is one subset of this set of good subsystems: {{b,d,$\underline{d}$}}.

38(A5|L2) There are update possibilities so update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The third intermediate state space created by updating with {{b,d,$\underline{d}$}} is as shown below.

Second ISS		Third ISS		Corresponding final stable states
a b c d	-->	**a b c d**	-->	**a b c d**
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 0 1 1		0 0 1 1		0 0 1 1
0 0 1 1		0 0 1 1		0 0 1 1
1 1 1 0		1 1 1 1		1 1 1 1
1 1 1 1		1 1 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1
1 1 1 1		1 1 1 1		1 1 1 1

39(A1|L3) Find the outputs with respect to the current intermediate state space.

There are no output state variables.

40(A2|L3) Since there are no output state variables, output a good decomposition.

The sets of subsystems used to transform the first intermediate state space into a stable states defines a decomposition. The second discovered decomposition is therefore

2: {b,d,$\underline{d}$}

1: {a,$\underline{b}$} {a,c,$\underline{c}$}

Since there are no more sets for updating at level 2, execution of the algorithm will continue with the iteration over the sets of subsystems suitable for updating found at level 1.

41(A5|L1) There are update possibilities so update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The next set of good subsystems is described by {{a,$\underline{b}$},{a,d,$\underline{d}$}}. The second intermediate state space created by this update is as shown below.

First ISS		Second ISS		Corresponding final stable states
a b c d	-->	a b c d	-->	a b c d
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 1 0 1		0 0 0 1		0 0 1 1
0 1 1 1		0 0 1 1		0 0 1 1
1 0 1 0		1 1 1 1		1 1 1 1
1 0 1 1		1 1 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1
1 1 1 1		1 1 1 1		1 1 1 1

42(A1|L2) Find the output state variables with respect to the current intermediate state space.

The only state variable with a value which changes between the second intermediate state space and the corresponding final stable states is {c}.

43(A2|L2) The set of output state variables is not empty.

44(A3|L2) Find the good subsystems with respect to the second intermediate state space.

The only good subsystem described by at least one output state variable, and one output state variable from the subsystems used in the last update, is {b,c,$\underline{c}$}.

45(A4|L2) Find subsets of the good subsystems for intermediate state space update.

There is one subset of this set of good subsystems: {{b,c,$\underline{c}$}}.

46(A5|L2) There are update possibilities so update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The third intermediate state space created by updating with {{b,c,$\underline{c}$}} is as shown below.

Second ISS		Third ISS		Corresponding final stable states
a b c d	-->	a b c d	-->	a b c d
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 0 0 1		0 0 1 1		0 0 1 1
0 0 1 1		0 0 1 1		0 0 1 1
1 1 1 1		1 1 1 1		1 1 1 1
1 1 1 1		1 1 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1
1 1 1 1		1 1 1 1		1 1 1 1

47(A1|L3) Find the outputs with respect to the current intermediate state space.

There are no output state variables.

48(A2|L3) Since there are no output state variables, output a good decomposition.

The sets of subsystems used to transform the first intermediate state space into a stable states defines a decomposition. The third discovered decomposition is therefore

2: {b,c,c}

1: {a,b} {a,d,d}

Since there are no more sets for updating at level 2, execution of the algorithm will continue with the iteration over the sets of subsystems suitable for updating found at level 1.

49(A5|L1) Update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The next set of good subsystems is described by {{a,c,c},{a,d,d}}. The second intermediate state space created by this update is as shown below.

First ISS		Second ISS		Corresponding final stable states
a b c d	-->	a b c d	-->	a b c d
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 1 0 1		0 1 1 1		0 0 1 1
0 1 1 1		0 1 1 1		0 0 1 1
1 0 1 0		1 0 1 1		1 1 1 1
1 0 1 1		1 0 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1
1 1 1 1		1 1 1 1		1 1 1 1

50(A1|L2) Find the output state variables with respect to the current intermediate state space.

The only state variable with a value which changes between the second intermediate state space and the corresponding final stable states is {b}.

51(A2|L2) The set of output state variables is not empty.

52(A3|L2) Find the good subsystems with respect to the second intermediate state space.

There are no good subsystems which meet the criteria for selection by the GoodSubsystems() function.

53(A4|L2) Since the are no suitable good subsystems, there can be no suitable sets for updating.

Execution of the algorithm will continue with the iteration over the sets of subsystems suitable for updating found at level 1.

54(A5|L1) There are update possibilities so update the current intermediate state space using one subset of the set of good subsystems, and call the Decompose() procedure.

The next set of good subsystems is described by {{a,b},{a,c,c},{a,d,d}}. The second intermediate state space created by this update is as shown below.

First ISS		Second ISS		Corresponding final stable states
a b c d	-->	a b c d	-->	a b c d
0 0 1 0		0 0 1 0		0 0 1 0
0 0 1 1		0 0 1 1		0 0 1 1
0 1 0 1		0 0 1 1		0 0 1 1
0 1 1 1		0 0 1 1		0 0 1 1
1 0 1 0		1 1 1 1		1 1 1 1
1 0 1 1		1 1 1 1		1 1 1 1
1 1 0 1		1 1 0 1		1 1 0 1
1 1 1 1		1 1 1 1		1 1 1 1

55(A1|L2) Find the outputs with respect to the current intermediate state space.

There are no output state variables.

56(A2|L2) Since there are no output state variables, output a good decomposition.

The sets of subsystems used to transform the first intermediate state space into a stable states defines a decomposition. The third discovered decomposition is therefore

1: {a,b} {a,c,c} {a,d,d}

There are no more sets of subsystems suitable for updating at any level of iteration, therefore the algorithm is finished.

Appendix I: Possible Decompositions for the "Batch" Payroll System

This appendix lists all of the possible decompositions for the "batch" payroll system. The state variable names have been abbreviated to conserve space.

Abbreviations

end = end	hours = hours worked
pay_r = pay rate	emp_p = employee position
emp_t = employee type	sales = sales
base = base pay	com = commissions
over = over time pay	total_pay = total pay
ben = benefits	

Decompositions

Decomposition #1
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}
{end,emp_t,emp_p,hours,over}
{end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #2
2: {pay_r,hours,ben,base}
1: {end,ben} {end,emp_t,emp_p,sales,com}
{end,emp_t,emp_p,hours,over}
{end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #3
2: {pay_r,hours,total_pay,base}
1: {end,ben} {end,emp_t,emp_p,sales,com}
{end,emp_t,emp_p,hours,over}
{end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #4
2: {emp_t,emp_p,sales,base,com}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,hours,over}
{end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #5
2: {emp_t,emp_p,sales,total_pay,com}
1: {end,pay_r,hours,base} {end,ben}
{end,emp_t,emp_p,hours,over}
{end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #6
2: {pay_r,hours,ben,base} {emp_t,emp_p,sales,ben,com}
1: {end,ben} {end,emp_t,emp_p,hours,over}

```
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #7
2:    (pay_r,hours,ben,base)  (emp_t,emp_p,sales,total_pay,com)
1:    (end,ben)  (end,emp_t,emp_p,hours,over)
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #8
2:    (pay_r,hours,total_pay,base)  (emp_t,emp_p,sales,ben,com)
1:    (end,ben)  (end,emp_t,emp_p,hours,over)
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #9
2:    (pay_r,hours,total_pay,base)  (emp_t,emp_p,sales,total_pay,com)
1:    (end,ben)  (end,emp_t,emp_p,hours,over)
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #10
2:    (emp_t,emp_p,hours,ben,over)
1:    (end,pay_r,hours,base)  (end,ben)  (end,emp_t,emp_p,sales,com)
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #11
2:    (emp_t,emp_p,hours,total_pay,over)
1:    (end,pay_r,hours,base)  (end,ben)  (end,emp_t,emp_p,sales,com)
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #12
2:    (pay_r,hours,ben,base)  (emp_t,emp_p,hours,ben,over)
1:    (end,ben)  (end,emp_t,emp_p,sales,com)
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #13
2:    (pay_r,hours,ben,base)  (emp_t,emp_p,hours,total_pay,over)
1:    (end,ben)  (end,emp_t,emp_p,sales,com)
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #14
2:    (pay_r,hours,total_pay,base)  (emp_t,emp_p,hours,ben,over)
1:    (end,ben)  (end,emp_t,emp_p,sales,com)
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #15
2:    (pay_r,hours,total_pay,base)  (emp_t,emp_p,hours,total_pay,over)
1:    (end,ben)  (end,emp_t,emp_p,sales,com)
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #16
2:    (emp_t,emp_p,sales,ben,com)  (emp_t,emp_p,hours,ben,over)
1:    (end,pay_r,hours,base)  (end,ben)
      (end,emp_t,emp_p,pay_r,hours,sales,total_pay)

Decomposition #17
2:    (emp_t,emp_p,sales,ben,com)  (emp_t,emp_p,hours,total_pay,over)
1:    (end,pay_r,hours,base)  (end,ben)
```

```
      {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #18
2:    {emp_t,emp_p,sales,total_pay,com}  {emp_t,emp_p,hours,ben,over}
1:    {end,pay_r,hours,base}  {end,ben}
      {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #19
2:     {emp_t,emp_p,sales,total_pay,com}
      {emp_t,emp_p,hours,total_pay,over}
1:    {end,pay_r,hours,base}  {end,ben}
      {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #20
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
      {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #21
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
      {emp_t,emp_p,hours,total_pay,over}
1:    {end,ben}  {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #22
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,sales,total_pay,com}
      {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #23
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,sales,total_pay,com}
      {emp_t,emp_p,hours,total_pay,over}
1:    {end,ben}  {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #24
2:    {pay_r,hours,total_pay,base}  {emp_t,emp_p,sales,ben,com}
      {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #25
2:    {pay_r,hours,total_pay,base}  {emp_t,emp_p,sales,ben,com}
      {emp_t,emp_p,hours,total_pay,over}
1:    {end,ben}  {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #26
2:    {pay_r,hours,total_pay,base}  {emp_t,emp_p,sales,total_pay,com}
      {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #27
2:    {pay_r,hours,total_pay,base}  {emp_t,emp_p,sales,total_pay,com}
      {emp_t,emp_p,hours,total_pay,over}
1:    {end,ben}  {end,emp_t,emp_p,pay_r,hours,sales,total_pay}

Decomposition #28
2:    {end,emp_t,emp_p,pay_r,hours,com,total_pay}
```

```
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #29
2:    {end,emp_t,emp_p,hours,sales,base,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #30
2:    {end,emp_t,emp_p,hours,base,com,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #31
2:    {end,emp_t,emp_p,sales,base,over,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #32
2:    {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #33
2:    {emp_t,emp_p,pay_r,hours,com,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #34
2:    {emp_t,emp_p,hours,sales,base,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #35
2:    {emp_t,emp_p,hours,base,com,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #36
2:    {emp_t,emp_p,sales,base,over,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #37
2:    {end,pay_r,hours,over,com,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #38
2:    {pay_r,hours,over,com,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #39
2:    {base,over,com,total_pay}
```

```
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #40
2:    {pay_r,hours,ben,base}
      {end,emp_t,emp_p,pay_r,hours,com,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #41
3:    {pay_r,hours,total_pay,base}
2:    {end,emp_t,emp_p,pay_r,hours,com,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #42
3:    {end,emp_t,emp_p,hours,sales,base,total_pay}
2:    {pay_r,hours,ben,base}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #43
3:    {end,emp_t,emp_p,hours,base,com,total_pay}
2:    {pay_r,hours,ben,base}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #44
3:    {end,emp_t,emp_p,sales,base,over,total_pay}
2:    {pay_r,hours,ben,base}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #45
3:    {emp_t,emp_p,hours,sales,base,ben,total_pay}
2:    {pay_r,hours,ben,base}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #46
3:    {emp_t,emp_p,hours,base,com,ben,total_pay}
2:    {pay_r,hours,ben,base}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #47
3:    {emp_t,emp_p,sales,base,over,ben,total_pay}
2:    {pay_r,hours,ben,base}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #48
3:    {base,over,com,total_pay}
2:    {pay_r,hours,ben,base}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
```

```
      {end,emp_t,emp_p,hours,over}

Decomposition #49
2:     {pay_r,hours,ben,base}
      {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #50
3:    {pay_r,hours,total_pay,base}
2:    {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #51
2:    {pay_r,hours,ben,base}
      {emp_t,emp_p,pay_r,hours,com,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #52
3:    {pay_r,hours,total_pay,base}
2:    {emp_t,emp_p,pay_r,hours,com,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #53
2:    {pay_r,hours,ben,base}  {end,pay_r,hours,over,com,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #54
3:    {pay_r,hours,total_pay,base}
2:    {end,pay_r,hours,over,com,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #55
2:    {pay_r,hours,ben,base}  {pay_r,hours,over,com,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #56
3:    {pay_r,hours,total_pay,base}
2:    {pay_r,hours,over,com,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}
      {end,emp_t,emp_p,hours,over}

Decomposition #57
2:    {emp_t,emp_p,sales,ben,com}
      {end,emp_t,emp_p,hours,sales,base,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #58
3:    {emp_t,emp_p,sales,total_pay,com}
```

```
2:    {end,emp_t,emp_p,hours,sales,base,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #59
3:    {end,emp_t,emp_p,pay_r,hours,com,total_pay}
2:    {emp_t,emp_p,sales,ben,com}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #60
3:    {end,emp_t,emp_p,hours,base,com,total_pay}
2:    {emp_t,emp_p,sales,ben,com}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #61
3:    {emp_t,emp_p,pay_r,hours,com,ben,total_pay}
2:    {emp_t,emp_p,sales,ben,com}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #62
3:    {emp_t,emp_p,hours,base,com,ben,total_pay}
2:    {emp_t,emp_p,sales,ben,com}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #63
3:    {end,pay_r,hours,over,com,total_pay}
2:    {emp_t,emp_p,sales,ben,com}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #64
3:    {pay_r,hours,over,com,ben,total_pay}
2:    {emp_t,emp_p,sales,ben,com}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #65
3:    {base,over,com,total_pay}
2:    {emp_t,emp_p,sales,ben,com}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #66
2:    {emp_t,emp_p,sales,ben,com}
      {end,emp_t,emp_p,sales,base,over,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #67
3:    {emp_t,emp_p,sales,total_pay,com}
2:    {end,emp_t,emp_p,sales,base,over,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #68
2:    {emp_t,emp_p,sales,ben,com}
      {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #69
3:    {emp_t,emp_p,sales,total_pay,com}
```

```
2:    {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #70
2:    {emp_t,emp_p,sales,ben,com}
      {emp_t,emp_p,hours,sales,base,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #71
3:    {emp_t,emp_p,sales,total_pay,com}
2:    {emp_t,emp_p,hours,sales,base,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #72
2:    {emp_t,emp_p,sales,ben,com}
      {emp_t,emp_p,sales,base,over,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #73
3:    {emp_t,emp_p,sales,total_pay,com}
2:    {emp_t,emp_p,sales,base,over,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #74
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
      {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #75
3:    {pay_r,hours,total_pay,base}
2:    {emp_t,emp_p,sales,ben,com}
      {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #76
3:    {emp_t,emp_p,sales,total_pay,com}
2:    {pay_r,hours,ben,base}
      {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #77
3:    {pay_r,hours,total_pay,base}  {emp_t,emp_p,sales,total_pay,com}
2:    {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #78
3:    {end,emp_t,emp_p,pay_r,hours,com,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
1:    {end,ben}  {end,emp_t,emp_p,hours,over}

Decomposition #79
3:    {end,emp_t,emp_p,hours,sales,base,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
1:    {end,ben}  {end,emp_t,emp_p,hours,over}
```

Decomposition #80
3: {end,emp_t,emp_p,hours,base,com,total_pay}
2: {pay_r,hours,ben,base} {emp_t,emp_p,sales,ben,com}
1: {end,ben} {end,emp_t,emp_p,hours,over}

Decomposition #81
3: {end,emp_t,emp_p,sales,base,over,total_pay}
2: {pay_r,hours,ben,base} {emp_t,emp_p,sales,ben,com}
1: {end,ben} {end,emp_t,emp_p,hours,over}

Decomposition #82
3: {emp_t,emp_p,pay_r,hours,com,ben,total_pay}
2: {pay_r,hours,ben,base} {emp_t,emp_p,sales,ben,com}
1: {end,ben} {end,emp_t,emp_p,hours,over}

Decomposition #83
3: {emp_t,emp_p,hours,sales,base,ben,total_pay}
2: {pay_r,hours,ben,base} {emp_t,emp_p,sales,ben,com}
1: {end,ben} {end,emp_t,emp_p,hours,over}

Decomposition #84
3: {emp_t,emp_p,hours,base,com,ben,total_pay}
2: {pay_r,hours,ben,base} {emp_t,emp_p,sales,ben,com}
1: {end,ben} {end,emp_t,emp_p,hours,over}

Decomposition #85
3: {emp_t,emp_p,sales,base,over,ben,total_pay}
2: {pay_r,hours,ben,base} {emp_t,emp_p,sales,ben,com}
1: {end,ben} {end,emp_t,emp_p,hours,over}

Decomposition #86
3: {end,pay_r,hours,over,com,total_pay}
2: {pay_r,hours,ben,base} {emp_t,emp_p,sales,ben,com}
1: {end,ben} {end,emp_t,emp_p,hours,over}

Decomposition #87
3: {pay_r,hours,over,com,ben,total_pay}
2: {pay_r,hours,ben,base} {emp_t,emp_p,sales,ben,com}
1: {end,ben} {end,emp_t,emp_p,hours,over}

Decomposition #88
3: {base,over,com,total_pay}
2: {pay_r,hours,ben,base} {emp_t,emp_p,sales,ben,com}
1: {end,ben} {end,emp_t,emp_p,hours,over}

Decomposition #89
2: {emp_t,emp_p,hours,ben,over}
{end,emp_t,emp_p,pay_r,hours,com,total_pay}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #90
3: {emp_t,emp_p,hours,total_pay,over}
2: {end,emp_t,emp_p,pay_r,hours,com,total_pay}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #91
3: {end,emp_t,emp_p,sales,base,over,total_pay}
2: {emp_t,emp_p,hours,ben,over}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #92
3: {emp_t,emp_p,sales,base,over,ben,total_pay}
2: {emp_t,emp_p,hours,ben,over}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #93
3: {end,pay_r,hours,over,com,total_pay}
2: {emp_t,emp_p,hours,ben,over}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #94
3: {pay_r,hours,over,com,ben,total_pay}
2: {emp_t,emp_p,hours,ben,over}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #95
3: {base,over,com,total_pay}
2: {emp_t,emp_p,hours,ben,over}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #96
2: {emp_t,emp_p,hours,ben,over}
{end,emp_t,emp_p,hours,sales,base,total_pay}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #97
3: {emp_t,emp_p,hours,total_pay,over}
2: {end,emp_t,emp_p,hours,sales,base,total_pay}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #98
2: {emp_t,emp_p,hours,ben,over}
{end,emp_t,emp_p,hours,base,com,total_pay}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #99
3: {emp_t,emp_p,hours,total_pay,over}
2: {end,emp_t,emp_p,hours,base,com,total_pay}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #100
2: {emp_t,emp_p,hours,ben,over}
{emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

Decomposition #101
3: {emp_t,emp_p,hours,total_pay,over}
2: {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1: {end,pay_r,hours,base} {end,ben} {end,emp_t,emp_p,sales,com}

```
Decomposition #102
2:    {emp_t,emp_p,hours,ben,over}
      {emp_t,emp_p,pay_r,hours,com,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #103
3:    {emp_t,emp_p,hours,total_pay,over}
2:    {emp_t,emp_p,pay_r,hours,com,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #104
2:    {emp_t,emp_p,hours,ben,over}
      {emp_t,emp_p,hours,sales,base,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #105
3:    {emp_t,emp_p,hours,total_pay,over}
2:    {emp_t,emp_p,hours,sales,base,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #106
2:    {emp_t,emp_p,hours,ben,over}
      {emp_t,emp_p,hours,base,com,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #107
3:    {emp_t,emp_p,hours,total_pay,over}
2:    {emp_t,emp_p,hours,base,com,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #108
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
      {end,emp_t,emp_p,pay_r,hours,com,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #109
3:    {pay_r,hours,total_pay,base}
2:    {emp_t,emp_p,hours,ben,over}
      {end,emp_t,emp_p,pay_r,hours,com,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #110
3:    {emp_t,emp_p,hours,total_pay,over}
2:    {pay_r,hours,ben,base}
      {end,emp_t,emp_p,pay_r,hours,com,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #111
3:    {pay_r,hours,total_pay,base}  {emp_t,emp_p,hours,total_pay,over}
2:    {end,emp_t,emp_p,pay_r,hours,com,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #112
3:    {end,emp_t,emp_p,hours,sales,base,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
```

```
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #113
3:    {end,emp_t,emp_p,hours,base,com,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #114
3:    {end,emp_t,emp_p,sales,base,over,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #115
3:    {emp_t,emp_p,hours,sales,base,ben,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #116
3:    {emp_t,emp_p,hours,base,com,ben,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #117
3:    {emp_t,emp_p,sales,base,over,ben,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #118
3:    {end,pay_r,hours,over,com,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #119
3:    {pay_r,hours,over,com,ben,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #120
3:    {base,over,com,total_pay}
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #121
2:    {pay_r,hours,ben,base}  {emp_t,emp_p,hours,ben,over}
      {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #122
3:    {pay_r,hours,total_pay,base}
2:    {emp_t,emp_p,hours,ben,over}
      {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,ben}  {end,emp_t,emp_p,sales,com}

Decomposition #123
3:    {emp_t,emp_p,hours,total_pay,over}
```

```
2:    (pay_r,hours,ben,base)
      (emp_t,emp_p,pay_r,hours,sales,ben,total_pay)
1:    (end,ben)  (end,emp_t,emp_p,sales,com)

Decomposition #124
3:    (pay_r,hours,total_pay,base)  (emp_t,emp_p,hours,total_pay,over)
2:    (emp_t,emp_p,pay_r,hours,sales,ben,total_pay)
1:    (end,ben)  (end,emp_t,emp_p,sales,com)

Decomposition #125
2:    (pay_r,hours,ben,base)  (emp_t,emp_p,hours,ben,over)
      (emp_t,emp_p,pay_r,hours,com,ben,total_pay)
1:    (end,ben)  (end,emp_t,emp_p,sales,com)

Decomposition #126
3:    (pay_r,hours,total_pay,base)
2:    (emp_t,emp_p,hours,ben,over)
      (emp_t,emp_p,pay_r,hours,com,ben,total_pay)
1:    (end,ben)  (end,emp_t,emp_p,sales,com)

Decomposition #127
3:    (emp_t,emp_p,hours,total_pay,over)
2:    (pay_r,hours,ben,base)
      (emp_t,emp_p,pay_r,hours,com,ben,total_pay)
1:    (end,ben)  (end,emp_t,emp_p,sales,com)

Decomposition #128
3:    (pay_r,hours,total_pay,base)  (emp_t,emp_p,hours,total_pay,over)
2:    (emp_t,emp_p,pay_r,hours,com,ben,total_pay)
1:    (end,ben)  (end,emp_t,emp_p,sales,com)

Decomposition #129
2:    (emp_t,emp_p,sales,ben,com)  (emp_t,emp_p,hours,ben,over)
      (end,emp_t,emp_p,hours,sales,base,total_pay)
1:    (end,pay_r,hours,base)  (end,ben)

Decomposition #130
3:    (emp_t,emp_p,sales,total_pay,com)
2:    (emp_t,emp_p,hours,ben,over)
      (end,emp_t,emp_p,hours,sales,base,total_pay)
1:    (end,pay_r,hours,base)  (end,ben)

Decomposition #131
3:    (emp_t,emp_p,hours,total_pay,over)
2:    (emp_t,emp_p,sales,ben,com)
      (end,emp_t,emp_p,hours,sales,base,total_pay)
1:    (end,pay_r,hours,base)  (end,ben)

Decomposition #132
3:    (emp_t,emp_p,sales,total_pay,com)
      (emp_t,emp_p,hours,total_pay,over)
2:    (end,emp_t,emp_p,hours,sales,base,total_pay)
1:    (end,pay_r,hours,base)  (end,ben)

Decomposition #133
```

```
3:    {end,emp_t,emp_p,pay_r,hours,com,total_pay}
2:    {emp_t,emp_p,sales,ben,com}  {emp_t,emp_p,hours,ben,over}
1:    {end,pay_r,hours,base}  {end,ben}

Decomposition #134
3:    {end,emp_t,emp_p,hours,base,com,total_pay}
2:    {emp_t,emp_p,sales,ben,com}  {emp_t,emp_p,hours,ben,over}
1:    {end,pay_r,hours,base}  {end,ben}

Decomposition #135
3:    {end,emp_t,emp_p,sales,base,over,total_pay}
2:    {emp_t,emp_p,sales,ben,com}  {emp_t,emp_p,hours,ben,over}
1:    {end,pay_r,hours,base}  {end,ben}

Decomposition #136
3:    {emp_t,emp_p,pay_r,hours,com,ben,total_pay}
2:    {emp_t,emp_p,sales,ben,com}  {emp_t,emp_p,hours,ben,over}
1:    {end,pay_r,hours,base}  {end,ben}

Decomposition #137
3:    {emp_t,emp_p,hours,base,com,ben,total_pay}
2:    {emp_t,emp_p,sales,ben,com}  {emp_t,emp_p,hours,ben,over}
1:    {end,pay_r,hours,base}  {end,ben}

Decomposition #138
3:    {emp_t,emp_p,sales,base,over,ben,total_pay}
2:    {emp_t,emp_p,sales,ben,com}  {emp_t,emp_p,hours,ben,over}
1:    {end,pay_r,hours,base}  {end,ben}

Decomposition #139
3:    {end,pay_r,hours,over,com,total_pay}
2:    {emp_t,emp_p,sales,ben,com}  {emp_t,emp_p,hours,ben,over}
1:    {end,pay_r,hours,base}  {end,ben}

Decomposition #140
3:    {pay_r,hours,over,com,ben,total_pay}
2:    {emp_t,emp_p,sales,ben,com}  {emp_t,emp_p,hours,ben,over}
1:    {end,pay_r,hours,base}  {end,ben}

Decomposition #141
3:    {base,over,com,total_pay}
2:    {emp_t,emp_p,sales,ben,com}  {emp_t,emp_p,hours,ben,over}
1:    {end,pay_r,hours,base}  {end,ben}

Decomposition #142
2:    {emp_t,emp_p,sales,ben,com}  {emp_t,emp_p,hours,ben,over}
      {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}

Decomposition #143
3:    {emp_t,emp_p,sales,total_pay,com}
2:    {emp_t,emp_p,hours,ben,over}
      {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:    {end,pay_r,hours,base}  {end,ben}
```

```
Decomposition #144
3:    (emp_t,emp_p,hours,total_pay,over)
2:    (emp_t,emp_p,sales,ben,com)
      (emp_t,emp_p,pay_r,hours,sales,ben,total_pay)
1:    (end,pay_r,hours,base)  (end,ben)

Decomposition #145
3:    (emp_t,emp_p,sales,total_pay,com)
      (emp_t,emp_p,hours,total_pay,over)
2:    (emp_t,emp_p,pay_r,hours,sales,ben,total_pay)
1:    (end,pay_r,hours,base)  (end,ben)

Decomposition #146
2:    (emp_t,emp_p,sales,ben,com)  (emp_t,emp_p,hours,ben,over)
      (emp_t,emp_p,hours,sales,base,ben,total_pay)
1:    (end,pay_r,hours,base)  (end,ben)

Decomposition #147
3:    (emp_t,emp_p,sales,total_pay,com)
2:    (emp_t,emp_p,hours,ben,over)
      (emp_t,emp_p,hours,sales,base,ben,total_pay)
1:    (end,pay_r,hours,base)  (end,ben)

Decomposition #148
3:    (emp_t,emp_p,hours,total_pay,over)
2:    (emp_t,emp_p,sales,ben,com)
      (emp_t,emp_p,hours,sales,base,ben,total_pay)
1:    (end,pay_r,hours,base)  (end,ben)

Decomposition #149
3:    (emp_t,emp_p,sales,total_pay,com)
      (emp_t,emp_p,hours,total_pay,over)
2:    (emp_t,emp_p,hours,sales,base,ben,total_pay)
1:    (end,pay_r,hours,base)  (end,ben)

Decomposition #150
2:    (pay_r,hours,ben,base)  (emp_t,emp_p,sales,ben,com)
      (emp_t,emp_p,hours,ben,over)
      (emp_t,emp_p,pay_r,hours,sales,ben,total_pay)
1:    (end,ben)

Decomposition #151
3:    (pay_r,hours,total_pay,base)
2:    (emp_t,emp_p,sales,ben,com)  (emp_t,emp_p,hours,ben,over)
      (emp_t,emp_p,pay_r,hours,sales,ben,total_pay)
1:    (end,ben)

Decomposition #152
3:    (emp_t,emp_p,sales,total_pay,com)
2:    (pay_r,hours,ben,base)  (emp_t,emp_p,hours,ben,over)
      (emp_t,emp_p,pay_r,hours,sales,ben,total_pay)
1:    (end,ben)

Decomposition #153
3:    (pay_r,hours,total_pay,base)  (emp_t,emp_p,sales,total_pay,com)
```

```
2:     {emp_t,emp_p,hours,ben,over}
       {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:     {end,ben}

Decomposition #154
3:     {emp_t,emp_p,hours,total_pay,over}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:     {end,ben}

Decomposition #155
3:     {pay_r,hours,total_pay,base}  {emp_t,emp_p,hours,total_pay,over}
2:     {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:     {end,ben}

Decomposition #156
3:     {emp_t,emp_p,sales,total_pay,com}
       {emp_t,emp_p,hours,total_pay,over}
2:     {pay_r,hours,ben,base}
       {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:     {end,ben}

Decomposition #157
3:     {pay_r,hours,total_pay,base}  {emp_t,emp_p,sales,total_pay,com}
       {emp_t,emp_p,hours,total_pay,over}
2:     {emp_t,emp_p,pay_r,hours,sales,ben,total_pay}
1:     {end,ben}

Decomposition #158
3:     {end,emp_t,emp_p,pay_r,hours,com,total_pay}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}

Decomposition #159
3:     {end,emp_t,emp_p,hours,sales,base,total_pay}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}

Decomposition #160
3:     {end,emp_t,emp_p,hours,base,com,total_pay}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}

Decomposition #161
3:     {end,emp_t,emp_p,sales,base,over,total_pay}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}

Decomposition #162
3:     {emp_t,emp_p,pay_r,hours,com,ben,total_pay}
```

```
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}

Decomposition #163
3:     {emp_t,emp_p,hours,sales,base,ben,total_pay}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}

Decomposition #164
3:     {emp_t,emp_p,hours,base,com,ben,total_pay}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}

Decomposition #165
3:     {emp_t,emp_p,sales,base,over,ben,total_pay}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}

Decomposition #166
3:     {end,pay_r,hours,over,com,total_pay}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}

Decomposition #167
3:     {pay_r,hours,over,com,ben,total_pay}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}

Decomposition #168
3:     {base,over,com,total_pay}
2:     {pay_r,hours,ben,base}  {emp_t,emp_p,sales,ben,com}
       {emp_t,emp_p,hours,ben,over}
1:     {end,ben}
```

Appendix J: The Modified Payroll System

The modified payroll system as described in Chapter 3 is listed below. Note the inclusion of an "additional_payments" state variable.

The Model

```
/*****************************************************

  A Payroll System

  Basic system is from Wand's October 14, 1987
  example to the Wand and Weber "Control and
  Audit" paper.

  This system reflects the changes to the system
  described on page 4.

*****************************************************/

clauses

/* Event Definitions */

event("work lots",[v(hours,ot)]).
event("work some",[v(hours,reg)]).
event("no work",[v(hours,"0")]).
event("sales",[v(sales,nz)]).
event("no sales",[v(sales,"0")]).

/* State Variable Definitions */

state_variable(emp_t).      /* employee type */
state_variable(emp_p).      /* employee position */
state_variable(pay_r).      /* pay rate */
state_variable(hours).      /* hours worked */
```

```
state_variable(sales).      /* sales */
state_variable(base).       /* base pay */
state_variable(over).       /* overtime pay */
state_variable(com).        /* sales commissions */
state_variable(add_pay).    /* additional payments */
state_variable(total_pay). /* total pay */

/* State Variable Value Definitions */

values(emp_t,[o,s]).              /* office and sales */
values(emp_p,[r,m]).              /* regular and management */
values(hours,["0",reg,ot]).       /* zero, normal or overtime */
values(pay_r,["0",nz]).           /* zero or positive */
values(sales,["0",nz]).
values(base,["0",nz]).
values(over,["0",nz]).
values(com,["0",nz]).
values(add_pay,["0",nz]).
values(total_pay,["0",nz]).

/* Stability Conditions */

/* an employee may be in a regular or a management position */
static("management or regular",[v(emp_p,r)]).
static("management or regular",[v(emp_p,m)]).

/* an employee may have either an office or a sales job */
static("office or sales",[v(emp_t,o)]).
static("office or sales",[v(emp_t,s)]).

/* hours may be zero or non zero */
static("hours",[v(hours,ot)]).
static("hours",[v(hours,reg)]).
static("hours",[v(hours,"0")]).

/* pay rate may be zero or not zero */
```

```
static("pay rate",[v(pay_r,nz)]).
static("pay rate",[v(pay_r,"0")]).

/* sales may be zero or not zero */
static("sales",[v(sales,nz)]).
static("sales",[v(sales,"0")]).

/* non-management staff is entitled to overtime pay if hours is
   not zero */
static("non-management staff gets overtime",
    [v(emp_p,r),v(hours,ot),v(over,nz)]).
static("non-management staff gets overtime",
    [v(hours,reg),v(over,"0")]).
static("non-management staff gets overtime",
    [v(hours,"0"),v(over,"0")]).
static("non-management staff gets overtime",
    [v(emp_p,m),v(over,"0")]).

/* non-management staff is entitled to commissions if sales is
   not zero */
static("non-management staff gets commissions",
    [v(emp_p,r),v(sales,nz),v(com,nz)]).
static("non-management staff gets commissions",
    [v(sales,"0"),v(com,"0")]).
static("non-management staff gets commissions",
    [v(emp_p,m),v(com,"0")]).

/* office employees cannot earn more commissions than overtime
   and vice versa for sales employees */
static("office commissions and sales overtime are limited",
    [v(com,nz),v(over,nz),v(add_pay,nz)]).
static("office commissions and sales overtime are limited",
    [v(com,"0"),v(over,"0"),v(add_pay,"0")]).
static("office commissions and sales overtime are limited",
    [v(emp_t,o),v(over,nz),v(add_pay,nz)]).
static("office commissions and sales overtime are limited",
```

```
    [v(emp_t,o),v(over,"0"),v(add_pay,"0")]).
static("office commissions and sales overtime are limited",
    [v(emp_t,s),v(com,nz),v(add_pay,nz)]).
static("office commissions and sales overtime are limited",
    [v(emp_t,s),v(com,"0"),v(add_pay,"0")]).

/* all employees are entitled to base pay if hours and pay rate
   are not zero */
static("everyone gets base pay",
    [v(hours,ot),v(pay_r,nz),v(base,nz)]).
static("everyone gets base pay",
    [v(hours,reg),v(pay_r,nz),v(base,nz)]).
static("everyone gets base pay",
    [v(hours,"0"),v(base,"0")]).
static("everyone gets base pay",
    [v(pay_r,"0"),v(base,"0")]).

/* total pay must be calculated at EOP */
static("total pay",
    [v(base,nz),v(add_pay,nz),v(total_pay,nz)]).
static("total pay",
    [v(base,nz),v(add_pay,"0"),v(total_pay,nz)]).
static("total pay",
    [v(base,"0"),v(add_pay,nz),v(total_pay,nz)]).
static("total pay",
    [v(base,"0"),v(add_pay,"0"),v(total_pay,"0")]).

/* Corrective Actions */

/* calculate base pay at EOP */
dynamic("calculate base pay",
        [v(hours,ot),v(pay_r,nz)],
        [v(base,nz)]).
dynamic("calculate base pay",
        [v(hours,reg),v(pay_r,nz)],
        [v(base,nz)]).
```

```
dynamic("calculate base pay",
        [v(hours,"0")],
        [v(base,"0")]).
dynamic("calculate base pay",
        [v(pay_r,"0")],
        [v(base,"0")]).

/* calculate overtime for non-management staff */
dynamic("calculate overtime",
    [v(emp_p,r),v(hours,ot)],
    [v(over,nz)]).
dynamic("calculate overtime",
    [v(hours,reg)],
    [v(over,"0")]).
dynamic("calculate overtime",
    [v(hours,"0")],
    [v(over,"0")]).
dynamic("calculate overtime",
    [v(emp_p,m)],
    [v(over,"0")]).

/* calculate commissions for non-management staff */
dynamic("calculate commissions",
    [v(emp_p,r),v(sales,nz)],
    [v(com,nz)]).
dynamic("calculate commissions",
    [v(emp_p,m)],
    [v(com,"0")]).
dynamic("calculate commissions",
    [v(sales,"0")],
    [v(com,"0")]).

/* calculate additional payments */
dynamic("calculate additional payments",
    [v(com,nz),v(over,nz)],
    [v(add_pay,nz)]).
```

```
dynamic("calculate additional payments",
    [v(com,"0"),v(over,"0")],
    [v(add_pay,"0")]).
dynamic("calculate additional payments",
    [v(emp_t,o),v(over,nz)],
    [v(add_pay,nz)]).
dynamic("calculate additional payments",
    [v(emp_t,o),v(over,"0")],
    [v(add_pay,"0")]).
dynamic("calculate additional payments",
    [v(emp_t,s),v(com,nz)],
    [v(add_pay,nz)]).
dynamic("calculate additional payments",
    [v(emp_t,s),v(com,"0")],
    [v(add_pay,"0")]).

/* calculate total pay */
dynamic("calculate total pay",
    [v(base,nz)],
    [v(total_pay,nz)]).
dynamic("calculate total pay",
    [v(add_pay,nz)],
    [v(total_pay,nz)]).
dynamic("calculate total pay",
    [v(base,"0"),v(add_pay,"0")],
    [v(total_pay,"0")]).
```

Appendix K: Decompositions of the Modified Payroll System

This appendix lists all of the possible decompositions of the modified payroll system. The state variable names have been abbreviated to conserve space.

Abbreviations

hours = hours worked
emp_p = employee position
sales = employee sales
com = commissions
add_pay = additional pay
pay_r = pay rate
emp_t = employee type
base = base pay
over = over time pay
total_pay = total pay

Decompositions

Decomposition #1
2: {com,emp_t,over,add_pay} {base,com,emp_t,over,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #2
2: {emp_p,emp_t,over,sales,add_pay} {base,com,emp_t,over,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #3
2: {com,emp_p,emp_t,hours,add_pay} {base,com,emp_t,over,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #4
2: {com,emp_t,over,add_pay} {base,emp_p,emp_t,over,sales,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #5
2: {emp_p,emp_t,over,sales,add_pay}
{base,emp_p,emp_t,over,sales,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #6
3: {base,com,emp_t,hours,total_pay,add_pay}
2: {base,com,emp_t,over,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #7
2: {com,emp_t,over,add_pay} {com,emp_t,hours,over,pay_r,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #8

```
3:    {com,emp_t,hours,pay_r,total_pay,add_pay}
2:    {base,com,emp_t,over,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #9
2:    {com,emp_p,emp_t,hours,add_pay}  {base,emp_p,emp_t,over,sales,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #10
2:    {com,emp_t,over,add_pay}  {base,com,emp_p,emp_t,hours,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #11
2:    {emp_p,emp_t,over,sales,add_pay}  {com,emp_t,hours,over,pay_r,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #12
2:    {emp_p,emp_t,over,sales,add_pay}  {base,com,emp_p,emp_t,hours,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #13
3:    {base,com,emp_t,hours,total_pay,add_pay}
2:    {base,emp_p,emp_t,over,sales,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #14
2:    {com,emp_p,emp_t,hours,add_pay}  {com,emp_t,hours,over,pay_r,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #15
2:    {com,emp_t,over,add_pay}  {com,emp_p,emp_t,hours,pay_r,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #16
2:    {com,emp_p,emp_t,hours,add_pay}  {base,com,emp_p,emp_t,hours,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #17
3:    {com,emp_t,hours,pay_r,total_pay,add_pay}
2:    {base,emp_p,emp_t,over,sales,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #18
2:    {emp_p,emp_t,over,sales,add_pay}
      {com,emp_p,emp_t,hours,pay_r,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #19
3:    {base,com,emp_t,hours,total_pay,add_pay}
2:    {com,emp_t,hours,over,pay_r,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}

Decomposition #20
2:    {com,emp_t,over,add_pay}  {base,emp_p,emp_t,hours,sales,total_pay}
1:    {hours,pay_r,base}  {emp_p,sales,com}  {emp_p,hours,over}
```

Decomposition #21
3: {base,com,emp_t,hours,total_pay,add_pay}
2: {base,com,emp_p,emp_t,hours,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #22
2: {emp_p,emp_t,over,sales,add_pay}
{base,emp_p,emp_t,hours,sales,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #23
3: {com,emp_t,hours,pay_r,total_pay,add_pay}
2: {com,emp_t,hours,over,pay_r,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #24
2: {com,emp_p,emp_t,hours,add_pay} {com,emp_p,emp_t,hours,pay_r,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #25
3: {com,emp_t,hours,pay_r,total_pay,add_pay}
2: {base,com,emp_p,emp_t,hours,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #26
2: {com,emp_p,emp_t,hours,add_pay}
{base,emp_p,emp_t,hours,sales,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #27
3: {add_pay,base,total_pay}
2: {com,emp_t,over,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #28
3: {add_pay,hours,pay_r,total_pay}
2: {com,emp_t,over,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #29
3: {base,com,emp_t,hours,total_pay,add_pay}
2: {com,emp_p,emp_t,hours,pay_r,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #30
3: {add_pay,base,total_pay}
2: {emp_p,emp_t,over,sales,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #31
3: {add_pay,hours,pay_r,total_pay}
2: {emp_p,emp_t,over,sales,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #32
3: {com,emp_t,hours,pay_r,total_pay,add_pay}
2: {com,emp_p,emp_t,hours,pay_r,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #33
3: {base,com,emp_t,hours,total_pay,add_pay}
2: {base,emp_p,emp_t,hours,sales,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #34
3: {add_pay,base,total_pay}
2: {com,emp_p,emp_t,hours,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #35
3: {add_pay,hours,pay_r,total_pay}
2: {com,emp_p,emp_t,hours,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #36
3: {com,emp_t,hours,pay_r,total_pay,add_pay}
2: {base,emp_p,emp_t,hours,sales,total_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}

Decomposition #37
2: {add_pay,base,total_pay}
1: {emp_p,emp_t,hours,sales,add_pay} {hours,pay_r,base} {emp_p,sales,com}
{emp_p,hours,over}

Decomposition #38
2: {add_pay,hours,pay_r,total_pay}
1: {emp_p,emp_t,hours,sales,add_pay} {hours,pay_r,base} {emp_p,sales,com}
{emp_p,hours,over}

Decomposition #39
2: {base,com,emp_t,over,total_pay}
1: {emp_p,emp_t,hours,sales,add_pay} {hours,pay_r,base} {emp_p,sales,com}
{emp_p,hours,over}

Decomposition #40
2: {base,emp_p,emp_t,over,sales,total_pay}
1: {emp_p,emp_t,hours,sales,add_pay} {hours,pay_r,base} {emp_p,sales,com}
{emp_p,hours,over}

Decomposition #41
2: {com,emp_t,hours,over,pay_r,total_pay}
1: {emp_p,emp_t,hours,sales,add_pay} {hours,pay_r,base} {emp_p,sales,com}
{emp_p,hours,over}

Decomposition #42
2: {base,com,emp_p,emp_t,hours,total_pay}
1: {emp_p,emp_t,hours,sales,add_pay} {hours,pay_r,base} {emp_p,sales,com}
{emp_p,hours,over}

Decomposition #43
2: {com,emp_p,emp_t,hours,pay_r,total_pay}
1: {emp_p,emp_t,hours,sales,add_pay} {hours,pay_r,base} {emp_p,sales,com}
{emp_p,hours,over}

Decomposition #44
2: {base,emp_p,emp_t,hours,sales,total_pay}
1: {emp_p,emp_t,hours,sales,add_pay} {hours,pay_r,base} {emp_p,sales,com}
{emp_p,hours,over}

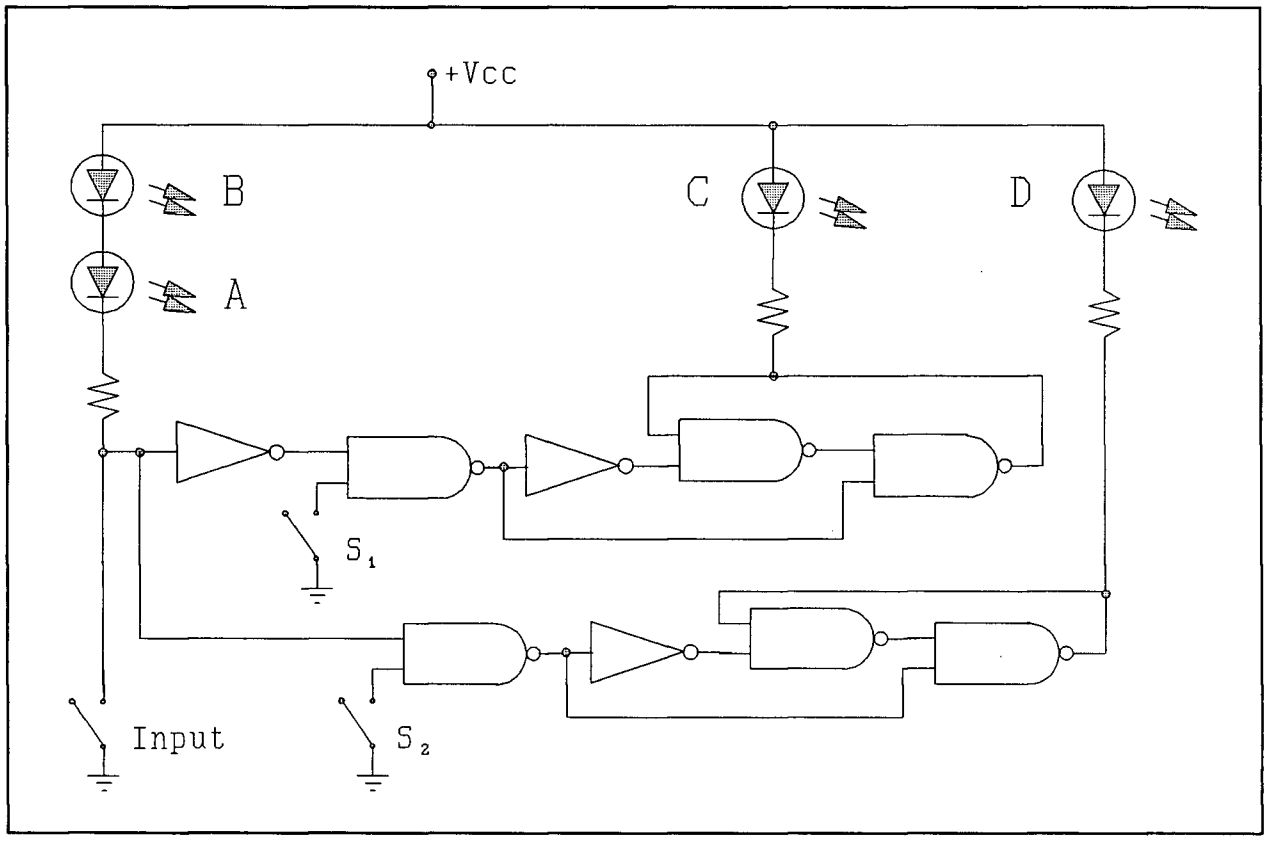
Decomposition #45
2: {com,emp_t,over,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}
{emp_p,emp_t,hours,pay_r,sales,total_pay}

Decomposition #46
2: {emp_p,emp_t,over,sales,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}
{emp_p,emp_t,hours,pay_r,sales,total_pay}

Decomposition #47
2: {com,emp_p,emp_t,hours,add_pay}
1: {hours,pay_r,base} {emp_p,sales,com} {emp_p,hours,over}
{emp_p,emp_t,hours,pay_r,sales,total_pay}

Decomposition #48
1: {emp_p,emp_t,hours,sales,add_pay} {hours,pay_r,base} {emp_p,sales,com}
{emp_p,hours,over} {emp_p,emp_t,hours,pay_r,sales,total_pay}

Appendix L: A Schematic Diagram of the Four-Lights System



The four lights A, B, C, and D have been implemented as light emitting diodes. When the switch labelled "Input" is closed, diodes A and B will both go on. The arrangement of inverters and NAND gates forms a sort of flip-flop arrangement. This is ensures the final states of diodes C and D are dependent on their initial states as well as on the state of diodes A and B. Switches S_1 and S_2 are used to set the initial stated of diodes C and D, respectively. (This circuit was developed by using the stable state and response path information described in Chapter 2 as inputs to the sequential circuit design techniques of Zissos (1979).)

Appendix M: Modified Variety and Two Independent Subsystems

Theorem:

The modified variety V_{AB} of a system S formed by two independent subsystems A and B, is equal to the product of the varieties of the subsystems V_A and V_B.

Proof:

Let I_A = the number of input states to subsystem A.

I_B = the number of input states to subsystem B.

I_{Ai} = the number of input states to subsystem A which lead to output state i.

I_{Bj} = the number of input states to subsystem B which lead to output state j.

n = the number of output states of subsystem A.

m = the number of output states of subsystem B.

By definition, if A and B are independent, the occurrence of a particular input state to A is not influenced by the occurrence of any input state to B. Therefore,

$I_A * I_B$ = the number of input states to the merged system S.

$I_{Ai} * I_{Bj}$ = the number of input states to S where A exhibits output state i and B exhibits output state j.

and the modified variety of S is given by

$$V_{AB} = \prod_{i=1}^{n} \prod_{j=1}^{m} \frac{I_A * I_B}{I_{Ai} * I_{Bj}} \wedge \frac{I_{Ai} * I_{Bj}}{I_A * I_B}$$

$$= \prod_{i=1}^{n} \prod_{j=1}^{m} \frac{I_A}{I_{Ai}} \wedge \frac{I_{Ai} * I_{Bj}}{I_A * I_B} * \frac{I_B}{I_{Bj}} \wedge \frac{I_{Ai} * I_{Bj}}{I_A * I_B}$$

$$= \prod_{i=1}^{n} \prod_{j=1}^{m} \frac{I_A}{I_{Ai}} \wedge \frac{I_{Ai} * I_{Bj}}{I_A * I_B} * \prod_{i=1}^{n} \prod_{j=1}^{m} \frac{I_B}{I_{Bj}} \wedge \frac{I_{Ai} * I_{Bj}}{I_A * I_B}$$

$$= \prod_{i=1}^{n} \prod_{j=1}^{m} \frac{I_A}{I_{Ai}} \wedge \frac{I_{Ai}}{I_A} \wedge \frac{I_{Bj}}{I_B} * \prod_{i=1}^{n} \prod_{j=1}^{m} \frac{I_B}{I_{Bj}} \wedge \frac{I_{Ai}}{I_A} \wedge \frac{I_{Bj}}{I_B}$$

and since

$$\prod_{i=1}^{n} X \wedge \frac{I_{Ai}}{I_A} = X \quad \text{and} \quad \prod_{j=1}^{m} X \wedge \frac{I_{Bj}}{I_B} = X \quad \text{for all } X,$$

$$V_{AB} = \prod_{i=1}^{n} \frac{I_A}{I_{Ai}} \wedge \frac{I_{Ai}}{I_A} * \prod_{j=1}^{m} \frac{I_B}{I_{Bj}} \wedge \frac{I_{Bj}}{I_B}$$

$$= V_A * V_B$$

Appendix N: The Combined Payroll System Model

This appendix contains a listing of a system model which will exhibit the behaviours of both the initial and modified payroll systems. Note the use of the state variable "sys" to distinguish between the two version of the system where necessary.

```
/*****************************************************

  A Payroll System

  Basic system is from Wand's October 14, 1987
  example to the Wand and Weber "Control and
  Audit" paper.

  This version is a combined system of both the
  basic and modified systems.

*****************************************************/

/* Event Definitions */

event("work lots",[v(hours,ot)]).
event("work some",[v(hours,reg)]).
event("no work",[v(hours,"0")]).
event("sales",[v(sales,nz)]).
event("no sales",[v(sales,"0")]).

/* State Variable Definitions */

state_variable(emp_t).      /* employee type */
state_variable(emp_p).      /* employee position */
state_variable(pay_r).      /* pay rate */
state_variable(hours).      /* hours worked */
state_variable(sales).      /* sales */
state_variable(base).       /* base pay */
```

```
state_variable(over).       /* overtime pay */
state_variable(com).        /* sales commissions */
state_variable(ben).        /* benefits */
state_variable(total_pay). /* total pay */
state_variable(add_pay).    /* additional payments */
state_variable(sys).        /* system identifier */

/* State Variable Value Definitions */

values(emp_t,[o,s]).              /* office and sales */
values(emp_p,[r,m]).              /* regular and management */
values(hours,["0",reg,ot]).       /* zero, normal or overtime */
values(pay_r,["0",nz]).           /* zero or positive */
values(sales,["0",nz]).
values(base,["0",nz]).
values(over,["0",nz]).
values(com,["0",nz]).
values(ben,[nz]).
values(total_pay,["0",nz]).
values(add_pay,["0",nz,nc]).      /* zero, positive, not calculated */
values(sys,[a,b]).                /* a=basic, b=modified system */

/* Stability Conditions */

/* an employee may be in a regular or a management position */
static("management or regular",[v(emp_p,r)]).
static("management or regular",[v(emp_p,m)]).

/* an employee may have either an office or a sales job */
static("office or sales",[v(emp_t,o)]).
static("office or sales",[v(emp_t,s)]).

/* hours may be zero or not zero */
static("hours",[v(hours,ot)]).
static("hours",[v(hours,reg)]).
static("hours",[v(hours,"0")]).
```

```
/* pay rate may be zero or not zero */
static("pay rate",[v(pay_r,nz)]).
static("pay rate",[v(pay_r,"0")]).

/* sales may be zero or not zero */
static("sales",[v(sales,nz)]).
static("sales",[v(sales,"0")]).

/* benefits must be calculated */
static("benefits",[v(ben,nz)]).

/* non-management staff might be entitled to overtime
   pay if hours is not zero */
static("non-management staff gets overtime",
    [v(sys,a),v(emp_t,o),v(emp_p,r),v(hours,ot),v(over,nz)]).
static("non-management staff gets overtime",
    [v(sys,a),v(emp_t,s),v(over,"0")]).
static("non-management staff gets overtime",
    [v(sys,b),v(emp_p,r),v(hours,ot),v(over,nz)]).
static("non-management staff gets overtime",
    [v(hours,reg),v(over,"0")]).
static("non-management staff gets overtime",
    [v(hours,"0"),v(over,"0")]).
static("non-management staff gets overtime",
    [v(emp_p,m),v(over,"0")]).

/* non-management staff might be entitled to commissions if
   sales is not zero */
static("non-management staff gets commissions",
    [v(sys,a),v(emp_t,s),v(emp_p,r),v(sales,nz),v(com,nz)]).
static("non-management staff gets commissions",
    [v(sys,a),v(emp_t,o),v(com,"0")]).
static("non-management staff gets commissions",
    [v(sys,b),v(emp_p,r),v(sales,nz),v(com,nz)]).
static("non-management staff gets commissions",
```

```
    [v(sales,"0"),v(com,"0")]).
static("non-management staff gets commissions",
    [v(emp_p,m),v(com,"0")]).

/* all employees are entitled to base pay if hours and
   pay rate are not zero */
static("everyone gets base pay",
    [v(hours,ot),v(pay_r,nz),v(base,nz)]).
static("everyone gets base pay",
    [v(hours,reg),v(pay_r,nz),v(base,nz)]).
static("everyone gets base pay",
    [v(hours,"0"),v(base,"0")]).
static("everyone gets base pay",
    [v(pay_r,"0"),v(base,"0")]).

/* office employees cannot earn more commissions than overtime
   and vice versa for sales employees */
static("office commissions and sales overtime are limited",
    [v(sys,a),v(add_pay,nc)]).
static("office commissions and sales overtime are limited",
    [v(sys,b),v(com,nz),v(over,nz),v(add_pay,nz)]).
static("office commissions and sales overtime are limited",
    [v(sys,b),v(com,"0"),v(over,"0"),v(add_pay,"0")]).
static("office commissions and sales overtime are limited",
    [v(sys,b),v(emp_t,o),v(over,nz),v(add_pay,nz)]).
static("office commissions and sales overtime are limited",
    [v(sys,b),v(emp_t,o),v(over,"0"),v(add_pay,"0")]).
static("office commissions and sales overtime are limited",
    [v(sys,b),v(emp_t,s),v(com,nz),v(add_pay,nz)]).
static("office commissions and sales overtime are limited",
    [v(sys,b),v(emp_t,s),v(com,"0"),v(add_pay,"0")]).

/* total pay must be calculated at EOP */
static("total pay",[v(base,nz),v(total_pay,nz)]).
static("total pay",[v(sys,a),v(over,nz),v(total_pay,nz)]).
static("total pay",[v(sys,a),v(com,nz),v(total_pay,nz)]).
```

```
static("total pay",[v(sys,a),v(base,"0"),v(over,"0"),v(com,"0"),
                    v(total_pay,"0")]).
static("total pay",[v(sys,b),v(add_pay,nz),v(total_pay,nz)]).
static("total pay",[v(sys,b),v(base,"0"),v(add_pay,"0"),
                    v(total_pay,"0")]).

/* Corrective Actions */

/* calculate base pay at EOP */
dynamic("calculate base pay",
         [v(hours,ot),v(pay_r,nz)],
         [v(base,nz)]).
dynamic("calculate base pay",
         [v(hours,reg),v(pay_r,nz)],
         [v(base,nz)]).
dynamic("calculate base pay",
         [v(hours,"0")],
         [v(base,"0")]).
dynamic("calculate base pay",
         [v(pay_r,"0")],
         [v(base,"0")]).

/* calculate overtime at EOP */
dynamic("calculate overtime",
         [v(sys,a),v(emp_t,o),v(emp_p,r),v(hours,ot)],
         [v(over,nz)]).
dynamic("calculate overtime",
         [v(sys,b),v(emp_p,r),v(hours,ot)],
         [v(over,nz)]).
dynamic("calculate overtime",
         [v(hours,reg)],
         [v(over,"0")]).
dynamic("calculate overtime",
         [v(hours,"0")],
         [v(over,"0")]).
```

```
/* calculate commissions at EOP */
dynamic("calculate commissions",
         [v(sys,a),v(emp_t,s),v(emp_p,r),v(sales,nz)],
         [v(com,nz)]).
dynamic("calculate commissions",
         [v(sys,b),v(emp_p,r),v(sales,nz)],
         [v(com,nz)]).
dynamic("calculate commissions",
         [v(sales,"0")],
         [v(com,"0")]).

/* calculate additional payments */
dynamic("calculate additional payments",
         [v(sys,b),v(com,nz),v(over,nz)],
         [v(add_pay,nz)]).
dynamic("calculate additional payments",
         [v(sys,b),v(com,"0"),v(over,"0")],
         [v(add_pay,"0")]).
dynamic("calculate additional payments",
         [v(sys,b),v(emp_t,o),v(over,nz)],
         [v(add_pay,nz)]).
dynamic("calculate additional payments",
         [v(sys,b),v(emp_t,o),v(over,"0")],
         [v(add_pay,"0")]).
dynamic("calculate additional payments",
         [v(sys,b),v(emp_t,s),v(com,nz)],
         [v(add_pay,nz)]).
dynamic("calculate additional payments",
         [v(sys,b),v(emp_t,s),v(com,"0")],
         [v(add_pay,"0")]).

/* calculate total pay */
dynamic("calculate total pay",
         [v(base,nz)],
         [v(total_pay,nz)]).
dynamic("calculate total pay",
```

```
        [v(sys,a),v(over,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(sys,a),v(com,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(sys,a),v(base,"0"),v(over,"0"),v(com,"0")],
        [v(total_pay,"0")]).
dynamic("calculate total pay",
        [v(sys,b),v(add_pay,nz)],
        [v(total_pay,nz)]).
dynamic("calculate total pay",
        [v(sys,b),v(base,"0"),v(add_pay,"0")],
        [v(total_pay,"0")]).
```

Appendix O: Calculation of Total Pay in the "Combined" Payroll System

This appendix has been included to explain the rather surprising result that the amount of total pay in the combined payroll system can be found without any knowledge of the current version of the system. The explanation will proceed by recalling the manner in which total pay is calculated in both the original and modified versions of the payroll system. Then the combined system will be examined. It will be shown that, given the correct inputs, calculation of total pay may be performed in exactly the same way in both system.

Total pay is a function of base pay (hours multiplied by pay rate), overtime pay, and commission pay. Calculation of overtime and commission pay depends on the employee's position ("emp_p") and type ("emp_t"), and is performed differently in the original and modified payroll systems. Calculation of overtime and commission pay in the original payroll system was governed by the following rules.

- the office staff ("emp_t" = "o")is entitled to overtime pay but not to commissions.
- the sales staff ("emp_t" = "s") is entitled to commissions but not to overtime pay.
- managers ("emp_p" = "m") are not entitled to overtime pay or commissions.

In the original payroll system, the employee's position and type information is used when calculating the final values of overtime and commission pay. Total pay could then be calculated knowing only the final values of base, overtime, and commission pay. These relationships are reflected in this decomposition of the original payroll system.

2: {base,com,over,total_pay}
1: {hours,pay_r,base} {emp_p,emp_t,hours,over} {emp_p,emp_t,sales,com}

where "hours" is the number of hours worked, and "sales" is the amount of sales.

In the modified payroll calculation of overtime and commission pay was somewhat different.

1. Both office staff and sales employees are entitled to both overtime pay and sales commissions.
2. An office employee cannot receive more in commissions than in overtime.
3. A sales employee cannot receive more in overtime than in commissions.

The state variable "add_pay" (for additional payments) was introduced to represent the total amount of overtime and commission pay earned by an employee after these restrictions are applied. Notice that calculation of overtime and commission pay no longer requires knowledge of employee type. The following decomposition reflects these new relationships.

3: {base,add_pay,total_pay}
2: {com,emp_t,over,add_pay}
1: {hours,pay_r,base} {emp_p,hours,over} {emp_p,sales,com}

Now examine the decomposition of the combined payroll system suggested by the specifications analysis tools. The state variable "sys" is used to distinguish between the two versions of the payrolls system (i.e. original and modified).

2: {com,emp_t,over,add_pay} {base,com,emp_t,over,total_pay}
1: {hours,pay_r,base} {emp_p,emp_t,sales,sys,com}
{emp_p,emp_t,hours,sys,over}

As expected, calculation of commission and overtime pay requires knowledge of the system version. However, notice that if the additional payments state variable in the modified payroll system total pay subsystem is replaced by its inputs, the resulting subsystem is the same as shown here. How can such a subsystem correctly calculate total pay for the original subsystem if the system version (i.e. original or modified) is not known? In other words, how does the subsystem know when to use the employee type information to limit the amount of overtime and commission pay, and when not to?

In fact, the limits on overtime and commission pay can always be applied. The limitations can only affect the amount of total pay if overtime exceeds commission pay and the employee type is sales, or commission exceeds overtime pay and the employee type is office. In the original payroll system, these two

situations cannot occur. In the original system, sales staff earn no overtime and office staff earn no commissions. Therefore, the same calculation for total pay as a function of employee type and base, commission, and overtime pay can be used for both the original and modified payroll systems.

Appendix P: The IFIP Working Conference Case

(from Olle, 1982, pp. 8-9)

1. Background

An IFIP Working Conference is an international conference intended to bring together experts from all IFIP countries to discuss some technical topic of specific interest to one or more IFIP Working Groups. The usual procedure, and that to be considered for the present purposes, is an invited conference which is not open to everyone. For such conferences it is something of a problem to ensure that members of the involved Working Group(s) and Technical Committee(s) are invited even if they do not come. Furthermore, it is important to ensure that sufficient people attend the conference so that the financial break-even point is reached without exceeding the maximum dictated by the facilities available.

IFIP Policy on Working Groups suggest the appointment of a Program Committee to deal with the technical content of the conference and an Organizing Committee to handle financial matters, local arrangements, and invitations and/or publicity. These committees clearly need to work together closely and have a need for common information and to keep their recorded information consistent and up to date.

2. Information system to be designed

The information system which is to be designed is that necessary to support the activities of both a Program Committee and an Organizing Committee involved in arranging an IFIP Working Conference. The involvement of the two committees is seen as analogous to two organizational entities within a corporate structure using some common information.

The following activities of the committees should be supported.

Program Committee:

1. Preparing a list to whom the call for papers is to be sent.
2. Registering the letters of intent received in response to the call.
3. Registering the contributed papers on receipt.
4. Distributing the papers among those undertaking the refereeing.
5. Collecting the referees' reports and selecting the papers for inclusion in the program.
6. Grouping selected papers into sessions for presentation and selecting chairman for each session.

Organizing Committee:

1. Preparing a list of people to invite to the conference.
2. Issuing priority invitations to National Representatives, Working Group members and members of associated working groups.
3. Ensuring all authors of each selected paper receive an invitation.
4. Ensuring authors of rejected papers receive an invitation.
5. Avoiding sending duplicate invitations to any individual.
6. Registering acceptance of invitations.
7. Generating a final list of attendees.

3. Boundaries of system

It should be noted that budgeting and financial aspects of the Organizing Committee's work, meeting plans of both committees, hotel accommodations for attendees and the matter of preparing camera-ready copy of the proceedings have been omitted from this exercise, although a submission may include some or all of these extra aspects if the authors feel so motivated.

Appendix Q: State Variable Identification for the IFIP Working Conference Problem

This appendix described the identification of most of the state variables required to model the IFIP Working Conference system. Each of the Working Group Conference activities (as presented in Appendix P) must be examined. The state variables required to model each of these activities will provide a starting point for modelling the underlying real-world system.

Activity: Preparing a list to whom the call for papers is to be sent.

This activity suggests that one property of the real world, with which the information system will be concerned, should indicate whether a particular person is to be invited to submit a paper to the Conference. This property, or state variable, will be called "pap_inv" (for "invited paper"). Invitations to submit papers are always sent to National Representatives, Working Group members, and members of associated working groups. A state variable indicating whether a person is in any of these categories will be called "grp_mem" (for "group member"). Individuals in each category are treated the same with respect to all of the activities which the information system is to support. Therefore, to avoid unnecessary complexity only one state variable is used. Individuals not in any of the above categories could also be invited to submit a paper. The state variable "ext_inv" (for "external invitation") will be used to indicate whether this is the case. Each of these state variables will have two values "y" and "n" (for "yes" and "no") to indicate whether a person has been invited to submit a paper, is a group member, or will be invited to submit a paper regardless of group membership. Notice that state variables describing the list itself are not properly a part of the system being modelled[135]. The list is an artifact of the implemented information system and need not be included in model of the real world.

Activity: Registering the letters of intent received in response to the call.

---

[135] For example, state variables describing the might include list currency or length. If some activities of the Committees required these properties, they would have to be included in the model.

A state variable to show that a letter of intent has been received is required. This state variable will be called "pap_prom" (for "paper promised") with values "y" and "n".

Activity: Registering the contributed papers on receipt.

A state variable indicating whether a paper has been received is required. This state variable will be called "pap_sub" (for "submitted paper") with values "y" and "n".

Activity: Distributing the papers among those undertaking the refereeing.

A state variable indicating the whether a paper has been sent to the referees is required. This state variable will be called "sent_ref" (for "sent to referees") with values "y" and "n".

Activity: Collecting the referees' reports and selecting the papers for inclusion in the program.

A state variable is required to indicate whether a paper has been returned by the referees. This state variable will be called "ret_ref" (for "returned by referees") with values "y" and "n". Another state variable required to represent the referees' decision as to the suitability of the paper for the Conference. This state variable will be called "ref_dec" (for "referees' decision") with values "acc" and "rej" (for "accept" and "reject"). Hopefully, this decision will be based solely on some property of the paper itself, and not on any other external considerations. A "paper suitability" state variable will be created to represent this property. This state variable will be called "suit" (for "suitable") with values "y" and "n". A suitable paper will be approved by the referees. A unsuitable paper will be rejected.

Activity: Grouping selected papers into sessions for presentation and selecting chairman for each session.

This activity requires two state variables. One to indicate whether a paper has been included in a session, and one indicating whether a session has been

assigned a chairman. These state variables will be called "sess_ass" and "chair" (for "session assignment" and "chairman", respectively). Each of these state variables will have values "y" and "n". Notice that the problem of assigning a particular paper to a particular session based on the contents of the paper is not explicitly addressed here. A state variable with values indicating the topic of the session, and another with values indicating the topic of the paper could be introduced. However, the problem description does not provide session topics, and as shall be seen, such detail is not necessary to form an initial model of the system.

Activity: Preparing a list of people to invite to the Conference.
Activity: Issuing priority invitations to National Representatives, Working Group members, and members of associated working groups.
Activity: Ensuring all authors of each selected paper receive an invitation.
Activity: Ensuring authors of rejected papers receive an invitation.
Activity: Avoiding sending duplicate invitations to any individual.

All of the above activities deal with the issuing of invitations. A state variables is required to indicate whether an invitation should be sent to an individual. This state variable will be called "inv" (for "invite") with values "y" and "n". National Representatives, Working Group Members, and members of associate working groups may be identified by the already defined state variable "grp_mem". Authors of papers may be identified by the already defined state variable "pap_sub".

Activity: Registering acceptance of invitations.

A state variable is required to indicate whether an individual has accepted an invitation. This state variable will be called "del_acc" (for "delegate accepts") with values "y" and "n".

Activity: Generating a final list of attendees.

A final list of attendees can be generated by examining the "del_acc" state variable.

Examination of the required system functions is now complete. The preliminary list of state variables for the IFIP Working Group Conference system is as follows:

State Variable	Values	Description
pap_inv	y,n	has the person been invited to submit a paper
grp_mem	y,n	is the person a Group member, National Representative, or member of an associated group
ext_inv	y,n	has the person been invited to submit a paper regardless of Group membership
pap_prom	y,n	has a paper been promised
pap_sub	y,n	has a paper been submitted
sent_ref	y,n	has the paper been sent to the referees
ret_ref	y,n	has the paper been returned by the referees
ref_dec	acc,rej	the referees either accept or reject a paper
suit	y,n	is the paper suitable for inclusion in the Conference
sess_ass	y,n	has the paper been assigned to a session
chair	y,n	has a session been assigned a chairman
inv	y,n	should the person be invited to attend the Conference
del_acc	y,n	has the person has accepted an invitation to attend the Conference

This list of state variable and values is not complete. Some state variables and values may be added to facilitate the definition of external events and sublaws. Additionally, some state variables will be dropped for reasons of computational efficiency.

Appendix R: External Event Identification for the IFIP Working Conference Problem

In SELMA, external events affect a system by altering the values of state variables. The values of other state variables may be changed by the system itself in response to an external event. Such secondary changes are called internal events. During this step, the analyst is primarily concerned with external events. Internal events are considered when system sublaws are defined. Each of the above state variables must be examined to decide whether its value is set by an external event. Discussions of only a few representative state variables are presented here.

pap_inv

The decision to invite a paper is dependent on the values of other state variables. The decision may not be made directly by forces outside of the system. Therefore, there is no external event affecting this state variable.

grp_mem

A person can become a Group member, or he or she can leave the Group. The decision affecting group membership is not made within the Conference system[136]. Therefore, an external event affecting the value of this state variable must be defined.

ref_dec

As explained earlier, the referees' decision as to the suitability of a paper is not an external event. The decision is being modelled as if it were based solely of the properties of the paper itself (represented here by the state variable "suit"). The effects of varying referee biases are not modelled here. It is likely that the decision of the referees is not, by itself, a sufficient

---

[136] Group membership may depend on other factors besides a willingness to join. For example, there may be membership requirements which must be met, or some form of payment may be required. Such considerations are beyond the scope of the Working Conference system.

reason for the Programme Committee to accept a paper. Obviously, if referees return a paper that was not sent to them by the Committee, the paper should not be accepted. There could be other factors, such as Committee member biases or paper length, but this sort of information is not detailed in the problem description. Another state variable is required to indicate acceptance of a paper by the Programme Committee. This state variable will be called "pap_dec" (for "final paper decision") and will have values "acc" and "rej" similar to the state variable "ref_dec". The value of "pap_dec" will be determined solely by the values of other state variables within the system. Therefore, no external event will be defined to directly affect "pap_dec".

del_acc

Whether a person accepts an invitation to attend the Conference is beyond the influence of the system. Therefore, acceptance of an invitation must be modelled using external events. However, this implies that registration for the Conference is entirely decided by factors external to the system. This is not the case. Mere acceptance of an invitation is not a sufficient condition for registration at the Conference. The delegate must also have been invited. Another state variable is required to indicate whether the delegate has actually been registered. This state variable will be called "del_reg" (for "registered delegate" and will have the values "y" and "n". The value of "del_reg" will not directly affected by external events, but is determined solely by the values of "inv" and "del_acc". Also note, the activity "generating a final list of attendees" will require the examination of the state variable "del_reg", instead of "del_acc" as suggested in Appendix Q.

Similar considerations can classify all the state variables into two groups: those directly affected by external events, and those not directly affected.

Directly Affected	Not Directly Affected
grp_mem	pap_inv
ext_inv	sent_ref
pap_prom	ref_dec
pap_sub	pap_dec

ret_ref

suit

chair

del_acc

sess_ass

inv

del_reg

Appendix S: Sublaw Identification for the IFIP Working Conference System Problem

This appendix describes the development of sublaws for a SELMA model of the IFIP Working Conference system. Some deliberate errors have been made. The discovery and correction of these errors is described in Chapter 6. The corrected system, expressed using the syntax required by the specifications analysis tools, is included as Appendix T.

Any external event may occur when the system is in any stable state. Therefore, state variables which are directly affected by external events may have any value in any situation.

```
Sublaw:      "A person may be a group member"
Stability Conditions:
grp_mem
   y
   n

Sublaw:      "A paper may be promised"
Stability Conditions:
pap_prom
   y
   n

Sublaw:      "Nonmembers may be invited to submit a paper"
Stability Conditions:
ext_inv
   y
   n

Sublaw:      "A paper may be submitted"
Stability Conditions:
pap_sub
   y
   n

Sublaw:      "A paper may be returned by the referees"
Stability Conditions:
ret_ref
   y
   n

Sublaw:      "A person may accept an invitation"
Stability Conditions:
del_acc
   y
   n
```

Sublaw: "A chairman may be assigned to a session"
Stability Conditions:

chair
y
n

The state variables "pap_sub" and "suit" are related. If a paper is not submitted to the Programme Committee, its suitability for inclusion in the Conference is irrelevant. The following sublaw expresses this relationship.

Sublaw: "Papers may be suitable of unsuitable"
Stability Conditions:

pap_sub	**suit**
y	y
y	n
n	n/a

The value "n/a" (for "not applicable") has been added to show that papers that have not yet been submitted are neither suitable or unsuitable.

The following sublaws are defined for state variables not directly affected by external events.

pap_inv

A person will be invited to submit a paper to the Conference if either of two conditions are met.

1. He or she is a Working Group member.
2. The Programme Committee decides to ask the person to submit a paper regardless of group membership status.

Sublaw: "Group members are invited to submit a paper"
Stability Conditions:

grp_mem	**ext_inv**	**pap_inv**
y	-	y
-	y	y
n	n	n

Corrective Actions:

Conditions			Actions
grp_mem	**ext_inv**	-->	**pap_inv**
y	-		y
-	y		y
n	n		n

Where "-" indicates "any value" or "don't care". Notice, that "grp_mem" and "ext_inv" may both have the value "y" at the same time. This corresponds to a situation where the Programme Committee decides to request a paper from a member of the Working Group. It was decided that this would be an acceptable (or "stable") situation, as the Programme Committee may wish to add extra incentive for some Working Group members to submit papers. If this situation were not acceptable, a new state variable would have to be introduced. For example, "ext_val" might indicate the validity of a request for an external paper. Sublaws would be added and modified as follows:

```
Sublaw:     "Only nonmembers may be invited to submit external papers"
Stability Conditions:
ext_inv     grp_mem     ext_val
   y           n           y
   n           -           n
Corrective Actions:
Conditions                      Actions
ext_inv     grp_mem     -->     ext_val
   y           n                   y
   n           -                   n

Sublaw:     "Group members are invited to submit a paper"
Stability Conditions:
grp_mem     ext_val     pap_inv
   y           -           y
   -           y           y
   n           n           n
Corrective Actions:
Conditions                      Actions
grp_mem     ext_val     -->     pap_inv
   y           -                   y
   -           y                   y
   n           n                   n
```

sent_ref

Papers are sent to referees only if the paper was both invited and submitted.

```
Sublaw:     "Referees only get invited papers"
Stability Conditions:
pap_sub     pap_inv     sent_ref
   y           y           y
   n           -           n
   -           n           n
```

Corrective Actions:

Conditions			Actions
pap_sub	**pap_inv**	-->	**sent_ref**
y	y		y
n	-		n
-	n		n

ref_dec

The referees' decision to accept or reject a paper is based solely on the suitability of the paper for the Working Conference. The system is only made aware of their decision when the paper is returned.

Sublaw: "Referees either accept or reject"

Stability Conditions

ret_ref	**suit**	**ref_dec**
y	y	acc
y	n	rej
n	-	n/a

Corrective Actions:

Conditions			Actions
ret_ref	**suit**	-->	**ref_dec**
y	y		acc
y	n		rej
n	-		n/a

A value of "n/a" (for "not applicable") has been added to show that when a paper has not been returned by the referees, the value of the state variable representing their decision is meaningless.

pap_dec

It is assumed that the decision to include a paper in the Conference is based solely on the validity of the referees' decision. The referees' decision will not be valid if the paper they judged was not sent to them by the Programme Committee or no paper was submitted. This last requirement may seem trivial,

but it is a necessary "error detection" condition[137]. External events may happen when the system is in any stable state. This means that the external event corresponding to the referees returning a paper could happen when the system is in a state where the paper had not been sent to the referees.

Sublaw: "Papers are accepted or rejected"
Stability Conditions:

ref_dec	sent_ref	pap_sub	pap_dec
acc	y	-	acc
rej	y	-	rej
n/a	-	-	rej
-	-	n	n/a

Corrective Actions:

Conditions				Actions
ref_dec	sent_ref	pap_sub	-->	pap_dec
acc	y	-		acc
rej	y	-		rej
n/a	-	-		rej
-	-	n		n/a

As for the state variable "ref_dec", a value of "n/a" (for "not applicable") has been added to "pap_dec" to show that, under some circumstances, the value of the state variable representing paper acceptance is meaningless.

sess_ass

Papers are assigned to session if they are accepted for inclusion in the Conference by the Programme Committee. Naturally papers are not assigned to a session if they are not submitted.

Sublaw: "Accepted papers are assigned to a session"
Stability Conditions:

pap_dec	pap_sub	sess_ass
acc	-	y
rej	-	n
-	n	n

Corrective Actions:

Conditions			Actions
pap_dec	pap_sub	-->	sess_ass
acc	-		y
rej	-		n
-	n		n

[137] This sort of error detection is often referred to as "input validation".

inv

A person is invited if one of the following conditions is met.

1. He or she is a Working Group member.
2. He or she has submitted a paper that has been accepted, rejected, or not yet returned by the referees[138].

Furthermore, no person should be invited twice and no invitation should be cancelled once issued. This last requirement implies that the stability conditions relevant to the state variable "inv" are not very restrictive. A person will not be invited if his or her paper is not considered by the Programme Committee (i.e. "sent_ref" is "n") and he or she is not a Group member. However, an invitation may be (or may have been) sent in any other situation.

Sublaw: "Authors of processed papers and group members are invited"
Stability Conditions:

sent_ref	grp_mem	inv
-	-	y
n	n	n

Corrective Actions:

Conditions					Actions
pap_dec	sent_ref	grp_mem	inv	-->	inv
-	-	-	y		y
acc	-	-	n		y
rej	-	-	n		y
-	y	-	n		y
-	-	y	n		y

del_reg

In order for a person to register for the Conference, he or she must have been invited and the invitation must be accepted.

[138] Notice that mere submission of a paper does not guarantee a person an invitation to attend the Conference. This is an invited paper conference. No paper will be sent to the referees by the Programme Committee unless it was previously invited.

Sublaw: "Delegates must be invited to register"
Stability Conditions:

inv	del_acc	del_reg
y	y	y
n	-	n
-	n	n

Corrective Actions:

Conditions			Actions
inv	del_acc	-->	del_reg
y	y		y
n	-		n
-	n		n

Appendix T: The IFIP Working Conference System Model

This appendix contains a listing of the IFIP Working Conference model expressed using the syntax required by the specifications analysis tools.

```
/************************************

   The IFIP Conference Problem

************************************/

clauses

event("become group member",[v(grp_mem,y)]).
event("leave group",[v(grp_mem,n)]).
event("submit suitable paper",[v(pap_sub,y),v(suit,y)]).
event("submit unsuitable paper",[v(pap_sub,y),v(suit,n)]).
event("no paper",[v(pap_sub,n)]).
event("referees return paper",[v(ret_ref,y)]).
event("referees do not return paper",[v(ret_ref,n)]).
event("invite external paper",[v(ext_inv,y)]).
event("do not invite external paper",[v(ext_inv,n)]).
event("delegate accepts invitation",[v(del_acc,y)]).
event("delegate does not accept invitation",[v(del_acc,n)]).

state_variable(pap_inv).
state_variable(ext_inv).
state_variable(pap_sub).
state_variable(sent_ref).
state_variable(ret_ref).
state_variable(ref_dec).
state_variable(pap_dec).
state_variable(sess_ass).
state_variable(del_acc).
state_variable(del_reg).
state_variable(inv).
```

```
state_variable(grp_mem).
state_variable(suit).

values(pap_inv,[y,n]).
values(ext_inv,[y,n]).
values(pap_sub,[y,n]).
values(suit,[y,n]).
values(sent_ref,[y,n]).
values(ret_ref,[y,n]).
values(ref_dec,[acc,rej,"n/a"]).
values(pap_dec,[acc,rej,"n/a"]).
values(sess_ass,[y,n]).
values(del_acc,[y,n]).
values(del_acc,[y,n]).
values(del_reg,[y,n]).
values(inv,[y,n]).
values(grp_mem,[y,n]).

static("a person may be a group member",[v(grp_mem,y)]).
static("a person may be a group member",[v(grp_mem,n)]).

static("nonmember's may be invited to submit a paper",[v(ext_inv,y)]).
static("nonmember's may be invited to submit a paper",[v(ext_inv,n)]).

static("group members are invited to submit a paper",
       [v(grp_mem,y),v(pap_inv,y)]).
static("group members are invited to submit a paper",
       [v(ext_inv,y),v(pap_inv,y)]). /* but so might others */
static("group members are invited to submit a paper",
       [v(grp_mem,n),v(ext_inv,n),v(pap_inv,n)]).

static("a paper may be submitted",[v(pap_sub,y)]).
static("a paper may be submitted",[v(pap_sub,n)]).

static("paper suitability may be anything",[v(suit,y)]).
static("paper suitability may be anything",[v(suit,n)]).
```

```
static("submitted papers may be suitable",[v(pap_sub,y),v(suit,y)]).
static("submitted papers may be suitable",[v(pap_sub,y),v(suit,n)]).
static("submitted papers may be suitable",[v(pap_sub,n)]).

static("considered papers must be
invited",[v(pap_sub,y),v(pap_inv,y),v(sent_ref,y)]).
static("considered papers must be invited",[v(pap_sub,n),v(sent_ref,n)]).
static("considered papers must be invited",[v(pap_inv,n),v(sent_ref,n)]).

static("a paper may be returned by the referees",[v(ret_ref,y)]).
static("a paper may be returned by the referees",[v(ret_ref,n)]).

static("referees either accept or reject",
       [v(ref_dec,acc),v(ret_ref,y),v(suit,y)]).
static("referees either accept or reject",
       [v(ref_dec,rej),v(ret_ref,y),v(suit,n)]).
static("referees either accept or reject",
       [v(ref_dec,"n/a"),v(ret_ref,n)]).

static("papers are accepted or rejected",
       [v(pap_dec,acc),v(sent_ref,y),v(ref_dec,acc)]).
static("papers are accepted or rejected",
       [v(pap_dec,rej),v(sent_ref,y),v(ref_dec,rej)]).
static("papers are accepted or rejected",
       [v(pap_dec,"n/a"),v(sent_ref,n)]).
static("papers are accepted or rejected",
       [v(pap_dec,"n/a"),v(ref_dec,"n/a")]).

static("accepted papers are assigned to a session",
       [v(sess_ass,y),v(pap_dec,acc)]).
static("accepted papers are assigned to a session",
       [v(sess_ass,n),v(pap_dec,rej)]).
static("accepted papers are assigned to a session",
       [v(sess_ass,n),v(pap_dec,"n/a")]).
static("accepted papers are assigned to a session",
```

```
        [v(sess_ass,n),v(pap_sub,n)]).

static("authors of processed papers and group members are invited",
        [v(inv,y)]).
static("authors of processed papers and group members are invited",
        [v(inv,n),v(sent_ref,n),v(grp_mem,n)]).

static("delegate may submit registration form",[v(del_acc,y)]).
static("delegate may submit registration form",[v(del_acc,n)]).

static("delegate must be invited to register",
        [v(del_reg,y),v(del_acc,y),v(inv,y)]).
static("delegate must be invited to register",
        [v(del_reg,n),v(del_acc,n)]).
static("delegate must be invited to register",
        [v(del_reg,n),v(inv,n)]).

dynamic("invite group members to submit a paper",
         [v(grp_mem,y)],[v(pap_inv,y)]).
dynamic("a nonmember is invited to submit a paper",
         [v(ext_inv,y)],[v(pap_inv,y)]).
dynamic("a nonmember is not invited to submit a paper",
         [v(grp_mem,n),v(ext_inv,n)],[v(pap_inv,n)]).

dynamic("invited papers are considered",
         [v(pap_sub,y),v(pap_inv,y)],[v(sent_ref,y)]).
dynamic("no paper -> not considered",
         [v(pap_sub,n)],[v(sent_ref,n)]).
dynamic("uninvited papers are not considered",
         [v(pap_inv,n)],[v(sent_ref,n)]).

dynamic("referees like acceptable papers",
         [v(suit,y),v(ret_ref,y)],[v(ref_dec,acc)]).
dynamic("referees don't like unacceptable papers",
         [v(suit,n),v(ret_ref,y)],[v(ref_dec,rej)]).
dynamic("no decision on unreturned papers",
```

```
        [v(ret_ref,n)],[v(ref_dec,"n/a")]).

dynamic("accept referee/acc decision if valid",
        [v(ref_dec,acc),v(sent_ref,y)],[v(pap_dec,acc)]).
dynamic("accept referee/rej decision if valid",
        [v(ref_dec,rej),v(sent_ref,y)],[v(pap_dec,rej)]).
dynamic("no decision on unreturned papers",
        [v(ref_dec,"n/a")],[v(pap_dec,"n/a")]).
dynamic("no decision on unconsidered papers",
        [v(sent_ref,n)],[v(pap_dec,"n/a")]).

dynamic("assign paper to a session",
        [v(pap_dec,acc)],[v(sess_ass,y)]).
dynamic("do not assign rejected paper to a session",
        [v(pap_dec,rej)],[v(sess_ass,n)]).
dynamic("do not assign unconsidered papers to a session",
        [v(pap_dec,"n/a")],[v(sess_ass,n)]).
dynamic("do not assign papers which were not submitted to a session",
        [v(pap_sub,n)],[v(sess_ass,n)]).

dynamic("invite accepted author",
        [v(pap_dec,acc),v(inv,n)],[v(inv,y)]).
dynamic("invite rejected author",
        [v(pap_dec,rej),v(inv,n)],[v(inv,y)]).
dynamic("invite unreturned author",
        [v(sent_ref,y),v(inv,n)],[v(inv,y)]).
dynamic("invite group member",
        [v(grp_mem,y),v(inv,n)],[v(inv,y)]).
dynamic("do not rescind an invitation",
        [v(inv,y)],[v(inv,y)]).

dynamic("those invited are registered",
        [v(inv,y),v(del_acc,y)],[v(del_reg,y)]).
dynamic("those not invited are not registered",
        [v(inv,n)],[v(del_reg,n)]).
dynamic("no form/no registration",
        [v(del_acc,n)],[v(del_reg,n)]).
```

Appendix U: Functional Forms of Some IFIP Working Conference Subsystems

The functional forms of the subsystems responsible for the calculation of state variable "pap_dec" and "sess_ass", in all of the suggested decompositions for the IFIP Working Conference system, are shown below.

{ref_dec,sent_ref,sess_ass}
Complexity = 3.90

ref_dec	sent_ref	--->	sess_ass
acc	y		y
n/a	y		n
rej	y		n
acc	n		n
n/a	n		n
rej	n		n

{ret_ref,sent_ref,sess_ass,pap_dec}
Complexity = 6.85

ret_ref	sent_ref	sess_ass	--->	pap_dec
y	y	y		acc
y	y	n		rej
n	y	n		n/a
y	n	n		n/a
n	n	n		n/a

{ref_dec,sent_ref,pap_dec}
Complexity = 7.51

ref_dec	sent_ref	--->	pap_dec
acc	y		acc
rej	y		rej
n/a	y		n/a
acc	n		n/a
n/a	n		n/a
rej	n		n/a

{pap_dec,sess_ass}

Complexity = 2.75

pap_dec --->	sess_ass
acc	y
n/a	n
rej	n

James Daniel Paulson

PUBLICATIONS:

Gattiker, U.E. and Dan Paulson, "The quest for effective teaching methods: Achieving computer literacy for end-users", INFOR, Vol. 25, No. 3, August 1987, pp. 256-272.

Gattiker, U.E. and Dan Paulson, "Effectiveness of teaching methods: Computer literacy and end-users", in Human Factors in Organizational Design and Management - II, Proceedings of the Second Symposium held in Vancouver, B.C., Canada, 19-21 August 1986, edited by O. Brown, Jr. and H.W. Hendrick, North-Holland, New York, 1986.

A. Hirose, D. Paulson, H.M. Skarsgard, S. Wolfe, "Experimental study of a toroidal plasma under the conditions for adiabatic Budker-Buneman instability", Physical Review Letters, 51(13), 26 September 1983.

A. Hirose, D. Paulson and H.M. Skarsgard, "Evolution of Electron Velocity Distribution Function Perpendicular to the Magnetic Field in a Turbulent Heating Experiment", APS Bulletin, 24(8), October 1979.

Paulson, Dan, "Automation of System Decomposition", Proceedings of the ICIS Doctoral Consortium, Faculty and Student Papers, New York University Graduate School of Business Administration, 1987.