TOWARDS FORMALIZATION OF SPECIFICATION DESIGN

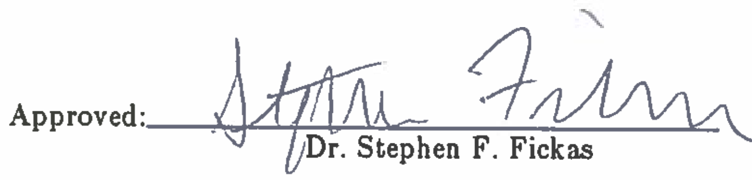
by

WILLIAM N. ROBINSON

A THESIS

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 1987



Approved: Dr. Stephen F. Fickas

An Abstract of the Thesis of

William N. Robinson for the degree of Master of Science

in the Department of Computer and Information Science

to be taken June 1987

Title: TOWARDS FORMALIZATION OF SPECIFICATION DESIGN

Approved:
Dr. Stephen F. Fickas

The primary objective of this research is to investigate the development of an intelligent environment for managing the complexity of a semantically rich specification model to assist in the design and evolution of specifications. In support of this objective, this thesis presents: (1) a model of specification development based on transformations of specifications and merging of divergent specifications, (2) an environment which assists in the application of a simplified set of these transformational operators.

An interesting aspect of the model is that some transformations are not correctness preserving. Such operators modify the meaning of a specification through (1) the modification of the goals and policies of the requirements which can then be compiled into the specification and (2) direct modification of the specification rationalized by goals and policies which have not been fully compiled into it.

VITA

NAME OF AUTHOR: William N. Robinson

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Oregon State University
Mt. Hood Community College

DEGREES AWARDED:

Master of Science, 1987, University of Oregon
Bachelor of Science, 1984, Oregon State University

AREAS OF SPECIAL INTEREST:

Artificial Intelligence
Software Engineering

PROFESSIONAL EXPERIENCE:

Research Assistant, Department of Computer and Information Sciences, University of Oregon, 1986-1987.

Teaching Assistant, Department of Computer and Information Sciences, University of Oregon, 1985-1986.

Research Assistant, Department of Computer and Information Sciences, University of Oregon, 1984-1985.

Programmer, Battelle Northwest Laboratories, Richland, Washington, Summer 1984.

NORCAS Fellow, Battelle Northwest Laboratories, Richland, Washington, Summer 1983.

PUBLICATIONS:

Fickas, S., Downing, K., Novick, D., Robinson, B., "The Specification, Design, and Implementation of Large Knowledge-Based Systems," in: Northcon, *Artificial Intelligence in the Northwest* (October 22-24 1985) p. 8/2.

ACKNOWLEDGMENTS

I happily take this opportunity to express my sincere thanks to all those that made this thesis possible and my experience at the University of Oregon enjoyable. My advisor, Stephen Fickas, constantly offered his gracious support and knowledge, and to him I owe my greatest debt. I also wish to thank Art Farley and Bill Bregar (external appraiser) for the time and energy they invested on my behalf. A special thanks to Martin Feather for his inspirational work. I would also like to express thanks to the graduate students who commented on parts of this work: Allen Brookes, Dan Lulich, David Meyer, and P Thyagarajan.

To my father and Alan

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION 1

1. Research Goals 1
2. Specification 2
3. Specification Design Model 4
4. Overview 6

II. FOUR MODELS OF DEVELOPMENT 7

1. Step-wise Refinement and Functional Decomposition 8
2. Correctness Preserving Transformations 10
3. Parallel Elaboration of Specifications 12
4. Summary of Issues 16
5. Why Parallel 18
6. Summary 19

III. KNOWLEDGE-BASED PES 20

1. Specification Model 20
2. Development Model 27
3. Power Tools 29
4. Summary 29

IV. CASE STUDY: A PATIENT MONITORING SYSTEM 30

1. Introduction 31
2. Oz 32
3. Oz Language 33
4. Specification Operator Architecture 36
5. KBPES Terminology 37
6. Simple Elaboration 39
7. Merging 62
8. Continuations 76
9. Summary 76

V. RELATED WORK 77

1. Theoretical Aspects (Languages) 78

2. Cognitive Aspects 79
3. Tools 79
4. Analysis 80
5. Models 81

VI. CONCLUSIONS 85

1. Methodology 85
2. Results 85
3. Future Directions 86

APPENDIX

A. OPERATOR SUMMARY 88

B. MERGE RULE SUMMARY 91

C. GIST SPECIFICATION 93

BIBLIOGRAPHY 99

TABLE OF FIGURES

Figure		Page
1.	The KBPES Model.	5
2.	Four Models of Development.	7
3.	Step-wise Functional Development	9
4.	Correctness Preserving Transformations.	11
5.	Parallel Elaboration of Specifications.	14
6.	A Set of Specification Constraints.	21
7.	Semantics of Specification Constraints.	22
8.	An Oz Specification.	25
9.	An Oz Specification Development Structure.	25
10.	The Oz Design Tool.	35
11.	Specification Modification as a Set of Changing Constraints.	37
12.	Initial Hierarchy of Specification Operators.	39
13.	Initial Oz Specification.	41
14.	Illustration of the 6 Basic Elaborations.	43
15.	Oz Modifications to Introduce Devices.	45
16.	Oz Modifications to Introduce Device Failure.	48
17.	Elaborating Patient Factors in Parallel with Devices.	51
18.	Oz Modifications to Introduce Composite of Factor Values.	51
19.	The Development Space After Starting the Clock Elaboration.	55
20.	Oz Modifications to Have the Monitor Read Periodically.	55
21.	Oz Modification to Elaborate Safe to Depend on which Patient It Is.	60
22.	Oz Modification to Store Patient Values.	62
23.	Illustration of the 6 Simple Elaborations Followed by 4 Merges.	64
24.	Summarization of the Interactions Between Specifications.	64
25.	Oz Merge of Factors and Devices.	69
26.	Oz Merge of Clock and Factors.	72
27.	Oz Merge of Clock and Devices_fail.	74
28.	Oz Merge of Storage and Devices_fail, Diagram Section.	74
29.	Oz Merge of Storage and Devices_fail, Constraint Section.	75
30.	Cyclic Model of Specification Design.	82
31.	Initial GIST Specification.	93
32.	Modifications to Introduce Devices.	94
33.	Modification to Introduce Device Failure.	94
34.	Modification of NOTICE_UNSAFE to Respond to Device Failure.	95
35.	Modifications to Adjust NOTIFY Invocations.	95
36.	Modifications to Introduce Composite of Factor Values.	96
37.	Modifications to Change the Monitor to Read Periodically.	97

38. Modifications to Elaborate Safe to Depend on Patient. 98
39. Modifications to Store Read Values. 98

CHAPTER I

INTRODUCTION

This thesis is based on the tenet that specification design is a process of compiling the goals and policies of problem requirements with the intention of (1) integrating goals and policies and (2) discovering unforeseen goal and policy interactions. Based on this requirements compilation assumption, I am seeking to develop an environment which supports this process. The next three sections present the overall research goals of this project and the results presented in this thesis. The fourth and final section of this chapter presents an overview of this thesis.

1. Research Goals

The primary objective of this research is to investigate the development of an intelligent environment for managing the complexity of a semantically rich specification model to assist in the design and evolution of specifications. To this end, the following supportive subgoals are being explored:

(1) Development of a specification representation that facilitates reasoning about specifications.

(2) Development of operators for goal integration and compilation into the above representation based on (a) transformations and (b) merges.

(3) Automation of the operators in 2 (a) and (b).

(4) Categorization of goal conflicts.

(5) Development of strategies for goal integration and compilation using knowledge of goal conflicts to guide the application of the operators of 3 (a) and (b).

These are the goals of this research and not the results reported here. Rather, here I present an initial formalization of the environment in which the above issues will be explored. This environment, called Oz, has the following (corresponding) characteristics which support this investigation:

(1) A very simple specification language which is useful in reasoning about control and data flow relations.

(2) A model of specification development based on transformations of specifications and merging of divergent specifications.

(3) Interactive application of a simplified set of the above operators.

The next section defines a *specification* as a compilation of the goals and policies of a *requirements* document. This is the view also taken by Fickas[19]. Formalization of the derivation of specifications from requirements is useful in that specifications can be constructed faster, more accurately, and more cost effectively. Moreover, the results of this research will function as a front-end to research on automatic implementation of specifications. Such technology is an area of active and productive research, and part of a larger endeavor to significantly alter and automate the software life cycle[4, 6, 27, 44]. This thesis represents an investigation of the design process of specification construction based on the tenet that this process is one of compilation of requirements to specification.

2. Specification

The requirements document expresses the needs of the user in terms of the problem domain (i.e., goals, policies, and a process model) and must be expressed in a language that can also express design decisions. This is because of the variety of constraints a requirement can express; essentially any property of the system that is necessary to gain its acceptability, including organization or resource utilization[22, 56]. However, it should not include expressions which do not describe the problem, yet constrain the software implementations. Hence, the requirements

document should not degrade into what is traditionally referred to as a *design specification*[41].

When I refer to a specification, it is not as the traditional use of the term, but rather a compilation of requirements into a formal document. Traditionally, a design specification describes the internal *functional* decomposition of the software into modules and the *performance* of each of those modules expressed in terms of their utilization of resources. Unless this information is contained in the requirements, and it should not unless it is an explicitly necessity of the user,[1] it will not exist in the specification.

What the specification does contain is the integration and compilation of goals and policies expressed in the requirements. *Compilation of goals and policies* is the process by which their explicit, high level, inefficient representation is converted into an implicit, lower level, and efficient representation. Goals and policies describe *what* the system is to achieve, but not *how* it is to achieve it.[2] Furthermore, their interactions are implicit. The process of moving *what* descriptions to *how* descriptions moves from higher level abstractions to lower level abstractions and from explicit achievements to implicit achievements. The lower level representation is more efficient in the (1) specification and discovery of interactions and (2) derivation of an implementation.[3]

Detection of conflicting and interacting goals and policies is an open problem. In the form of goals and policies the requirements of the system is explicit but goal

---

[1]The user may in fact explicitly request the specified software interface with other software, requiring functional decomposition and performance constraints in the requirements. But, despite the request, the analyst may perceive the user's request an unnecessary and not include it.

[2]A goal is a simple declarative statement of a desired state of the system. Policies describe desired *activities* (i.e., transition goals) which take place over time. Hence, these statements are comparatively more procedural.

[3]Systems such as Mostow's FOO [38] and transformational implementation [33] exemplify goal compilation. See also the workshop on knowledge compilation[13].

interactions are implicit. Compiling these goals and policies makes explicit their interactions and implicit the rationale behind them. Recording the compilation process is a way to link these polar representations.

3. Specification Design Model

The view of specification design taken here is that specification design knowledge can be partitioned into three major areas: design knowledge, development knowledge, and specification knowledge. This perspective is begin explored and formalized as the KBPES model of specification design.

Figure 1 illustrates how KBPES is divided into two major interacting components, the case database and the knowledge-base. The case database (workspace) contains the various aspects of the particular specification being constructed; it changes with each new system to be specified. The knowledge-base is less dynamic and is divided into three interacting partitions: the design model, the development model, and the specification model. Each of these are further divided into general and domain specific components.

The design model contains general and domain specific elaboration and merge strategies. Elaboration is the method by which goals are compiled into a specification. As will be described, elaboration can result in the separation of the specification into two or more documents. Merge is then used to make the specification a single consistent document again. This part of the model is more speculative; hence, it will not be discussed in detail.

The development model keeps track of the development structure of the specification. It supports isolation of specification components, their elaboration, and the merging of divergent specifications. The development model is based on a new (and speculative) paradigm for development, PES, which is described in chapter II.

The specification model contains general knowledge of syntactic "well-formedness" of specifications. Much of the well-formedness knowledge is based on a

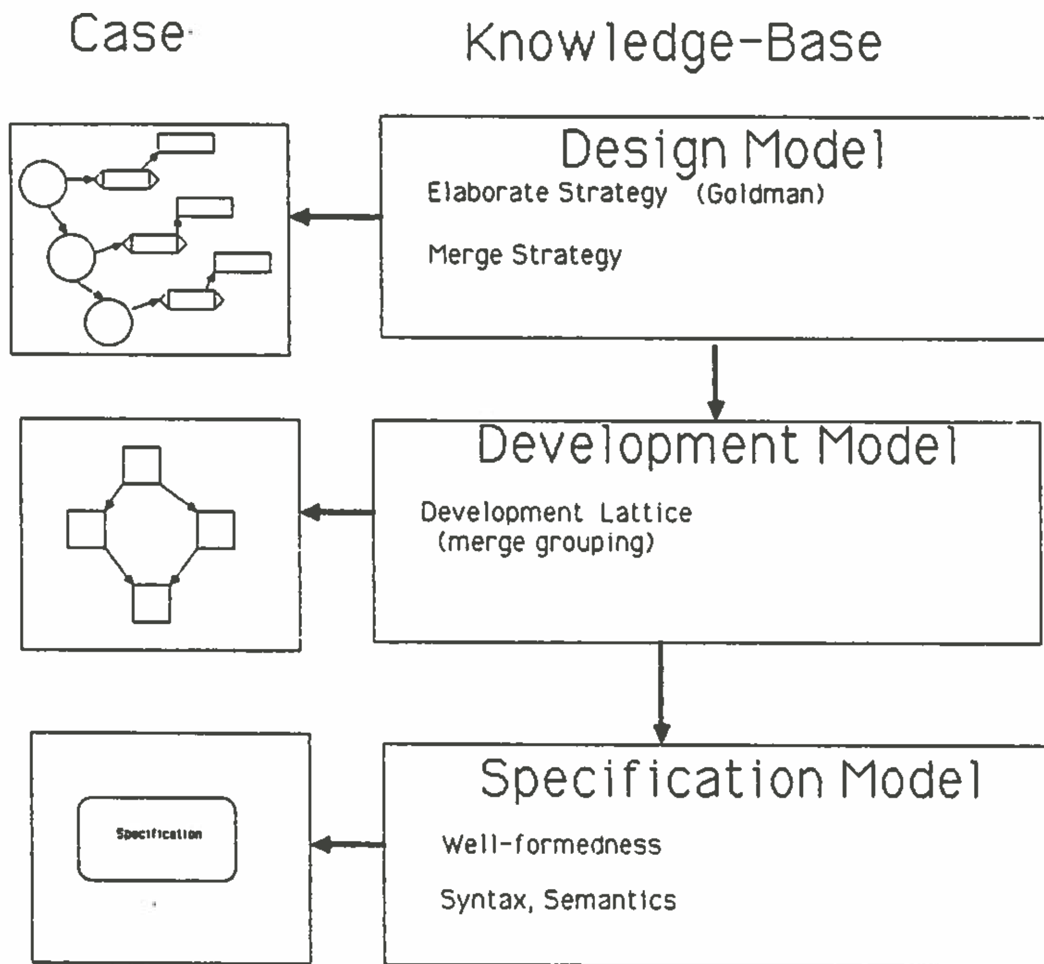


Figure 1. The KBPES Model.

general model of control and data flow. Domain specific knowledge specifies the syntax and semantics of the particular specification language being used. The syntax and semantics are used to determine language specific conflicts during a merge.

4. Overview

Requirements compilation facilitates the discovery and specification of goal and policy interactions. As the discovery of goal and policy conflicts and the specification of their resolution are the most difficult parts of the specification process, this thesis formalizes specification design as a requirements compilation process in order to support goal and policy reformulation.

Chapter II describes how the model of specification development, Parallel Elaboration of Specifications (PES), is similar, but distinct from other models of specification development. Chapter III presents a knowledge-based form of this model in detail, i.e., the development and specification knowledge models are described. Chapter IV illustrates the use of KBPES in the design of a simple specification. Related work and conclusions are discussed in chapters V and VI respectively. Appendices A and B describe the transformational operators used to develop the specification of chapter IV. Appendix C summarizes the original specification development on which the example of chapter IV is based.

CHAPTER II

FOUR MODELS OF DEVELOPMENT

This chapter explores four related models of development. Three of the models, step-wise refinement[54] , functional development[41] , and correctness preserving transformations (CPT)[16] are well established. The fourth model, parallel elaboration of specifications (PES) [17] is described as a model which subsumes the other three and is appropriate for investigation of a model of specification design; it is more speculative.

Figure 2 depicts these four models and their subsumption relations with the use of a Venn diagram with two discrete axes. As will be explained in greater detail in this

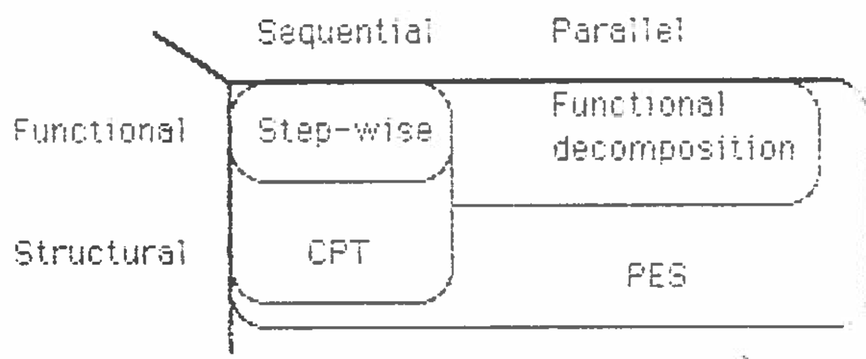


Figure 2. Four Models of Development.

chapter, step-wise refinement applies functional refinements sequentially to develop a specification. Function decomposition applies functional refinements in parallel to develop specifications. CPTs apply refinements sequentially, some of which may be functionally preserving. However, other refinements may alter the functionality of subcomponents, yet maintain the global meaning of the specification. This second group of refinements are referred to as structural refinements (cf. [5]). Finally, the PES model is introduced as a model which subsumes the other three. It is essentially a CPT model where refinements can be applied in parallel and also change the global meaning of the specification.

This presentation now commences with a discussion of the four models of development, followed by a summary of the salient development issues.

1. Step-wise Refinement and Functional Decomposition

Figure 3 illustrates the use of step-wise refinement and then functional decomposition to develop a patient monitoring system. In the first bubble of the figure, four components of the system are shown with their abstract dataflow relations. This diagram depicts multiple patients (P+) passing values to a single device (D) which then passes values to a single monitor (M) which communicates with a single nurses' station (N).

This patient monitoring example will be carried on throughout this thesis to provide illustrations and serve as the example specification to be developed in chapter IV. The monitor is the component to be implemented to react to its environment containing patients, devices, and a nurses' station, among other things. The basic task of the monitor is to notify the nurses' station when a patient becomes unsafe, where safeness is specified by the nurses' station.

The second bubble of figure 3 illustrates how the monitor can be functionally refined by the internal description of two subcomponents, a cache and a processor. The cache serves to accumulate values received from the device until all patient values

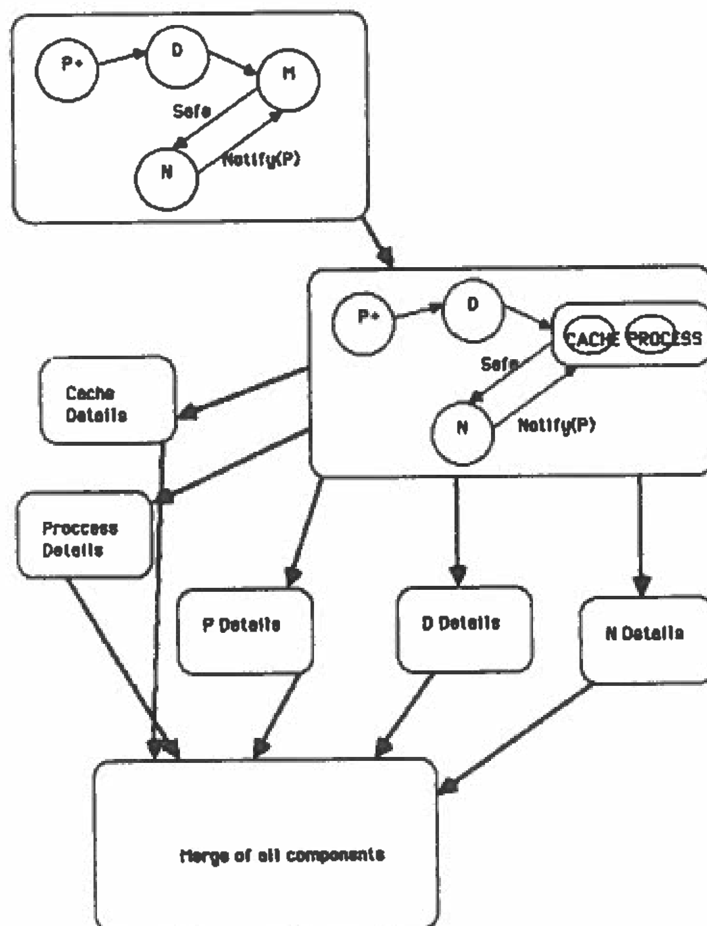


Figure 3. Step-wise Functional Development

have been accumulated. Then, the processor will do computations to determine whether the nurses' station should be notified.

The remaining bubbles of the figure illustrate how each of the components of the system can be further functionally refined in isolation and then merged into a single document. This is called functional decomposition.

Functional decomposition is just step-wise refinement where multiple refinements are allowed to be done in parallel. This model of development derives its benefits primarily from (1) verification, (2) multiple designers, and (3) a trivial merge scheme. Since refinements are restricted to be functionally preserving, verification is considerably simplified. Given any level of the specification, the immediate lower level can be verified as correctly implementing its functionality[31, 37]. Also, due to the sole use of functional refinements, interfaces are fixed and thus allow multiple components to be refined simultaneously in isolation of one another. Moreover, bringing these components back together, merging them into a single document, is trivial; simply consolidate the components into a single document.

At this point the concept of a merge may seem foreign or spurious. But, once the restriction on functionality preserving refinements is lifted, it will become all too apparent.

2. Correctness Preserving Transformations

Figure 4 shows a similar refinement to that of figure 2. The first bubble of the figure is the same as figure 2. But, the second bubble illustrates the introduction of the cache into the device instead of the monitor. Patient values are accumulated in the device and when all patient values have been collected they are all passed to the monitor at once.

This type of refinement is not functional but it is correctness preserving, i.e., a Correctness Preserving Transformation (CPT). Correctness preserving transformations maintain the global meaning or functionality of the specification, but not necessarily the local meaning.[1] Functional refinement is a subclass of CPT.

---

[1]In this case, even the form of the input into the component to be implemented (the monitor) was changed. However, the meaning it carries remains constant. CPTs may alter the functionality of a specification, but maintain its meaning.

The primary benefit of CPTs is that they can modify the functionality of multiple components of a specification while maintaining its correctness. For example, CPTs can change interfaces while functional refinements cannot. Verification of refinements is still possible with CPTs, but it is more problematic since it can be quite

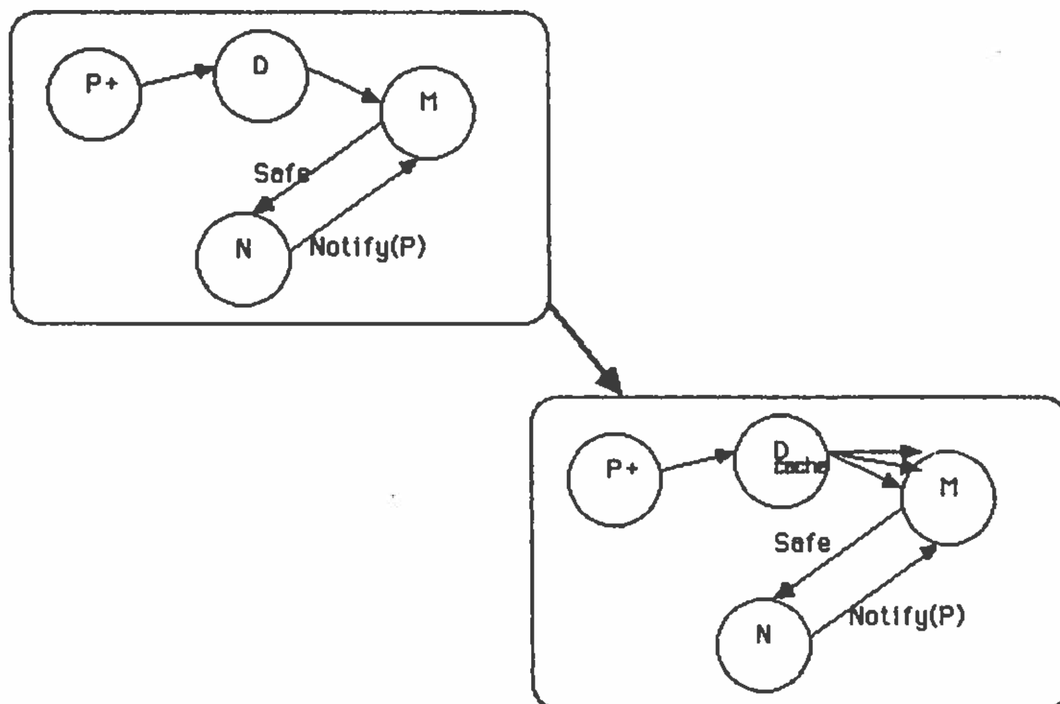


Figure 4. Correctness Preserving Transformations.

difficult to prove a transformation is correctness preserving.

CPTs are usually done sequentially. Hence, the specification is always represented as a single document and does not require a merge capability.[2] However, control in general does become in issue.

Using CPTs, refinements can be applied in many more places than that allowed using functional refinements. CPTs can be applied to components, interfaces between components (two components), and any combination of multiple components. Conversely, functional refinements can only be applied to single components. While it is also true functional refinement must determine where refinements are to be made and what they should do, it is considerably more acute using CPTs.

Intimately tied to where refinements are made is what they should do. Refinements compile requirement goals and policies into the specification. For example, moving the cache into the device as illustrated above was done to satisfy the goal of having the specification accurately reflect the actual devices that the monitor must interface with. Control of refinements must (1) determine which goal is to be compiled into the specification, (2) where the specification should be altered, and (3) which of many possible refinements should be applied.

3. Parallel Elaboration of Specifications

The proceeding two sections described three established development models, step-wise refinement, functional decomposition, and correctness preserving transformations, with respect to several issues that are important in all development models: (1) verification, (2) parallel refinements (multiple analysts), (3) merge strategies, and (4) control issues. This section continues this discussion with the introduction of a newer development model called, Parallel Elaboration of Specifications (PES)[17].

[2]CPTs applied in parallel, despite their correctness preserving nature, would require a complex merge scheme. Note that parallel CPTs could refine subcomponents which would not interface. Also, see the next section.

The primary features of PES is that it supports parallel refinements (multiple analysts) and modification of the meaning of the specification (noncorrectness preserving transformations). It is this second aspect, noncorrectness preserving, that requires us to call such transformations *reformulations* rather than refinements, as refinements imply functionality preserving developement. Figure 5 illustrates its use.

Figure 5 illustrates how the last bubble of figure 3 can have two reformulations applied in parallel to introduce device failure and patient factors. Neither of these reformulations are correctness preserving because they alter the functionality of the monitor, the component to be implemented.

Note that either one of these "reformulations" may actually involve the application of a sequence of reformulations for their achievement. So henceforth, the term elaboration will be used to mean the satisfaction of a requirement goal or policy by the application of a sequence of reformulations. Elaboration is the method of goal compilation. A reformulation is the application of a single transformation.

The second bubble of figure 4 illustrates the introduction of device failure. When a device fails the nurses' station must be notified. But, previously the nurses' station was notified when a patient was perceived as unsafe. The introduction of device failure requires the modification of two interfaces: device to monitor, and monitor to nurse station. In the second bubble the notify data arc distinguishes between device failure and patient illness.

The third bubble of figure 4 illustrates the introduction of patient factors; a set of values such as blood pressure, temperature, and pulse which represent the reformulation of the abstract value patients were modeled with previously. Introducing multiple values necessitates that the device read and accumulate multiple values for each patient and then pass them on to the monitor. Also, the notion of safeness, as defined by the nurses' station, must be reformulated to specify safeness for each patient factor.

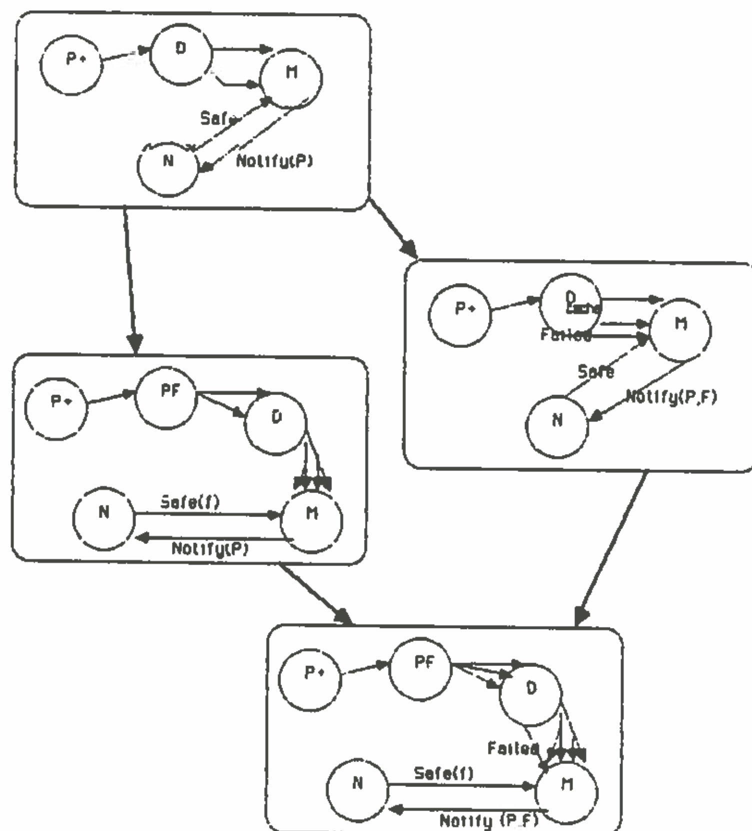


Figure 5. Parallel Elaboration of Specifications.

The device failure and patient factor elaborations both modify the meaning of the specification. Noncorrectness preserving changes are the result of compiling a goal or policy, which is not fully represented in the specification. Goals and policies may not be fully represented in a specification because (1) the design process has not

progressed to the point where they can be elaborated, or (2) introduction of other goals and policies (requirements modifications) necessitate further elaboration.

PES does facilitate multiple component modification (parallel elaborations) and meaning changes of the specification. But in so doing, it introduces complexity into verification and merging. Given a semanticly rich model of PES, this research aims to automate aspects of verification and merging, and support the directed control of elaborations.

For example, after the analysts introduces the failed arc between the device and the monitor in figure 4, the system will suggest to the analyst that the notification arc distinguish between device failure and patient illness. Similarly, when the analysts introduces patient factors it will be suggested that the safe relation should be reformulated to handle patient factors. This is a limited form of validation in which the analyst is asked to confirm each of his design actions. The system derives these queries from domain models as will be discussed in the next chapter.

The fourth bubble of figure 4, represents the merge of two divergent specifications. The merge must integrate the two compiled goals: that of properly representing and dealing with device failure, and that of properly representing and dealing with patient factors.

At the level of goal descriptions, it is difficult to recognize these two goals as interacting. But, in fact, they interact in such a way that a domain decision must be used to integrate them into a merged specification.

The analyst must determine whether the device should report failure after all patient factors are read, or after the first sign of device failure. If the later choice is made, then notify should distinguish which factor the device was reading when it failed. Figure 4 depicts the former choice; the device reports failure after all the factors of a patient have been read.

4. Summary of Issues

PES is the model of development which the knowledge-based model of specification design of chapter III is based. It was chosen for two reasons. First, the other models of development presented in this chapter can be simulated by constraining PES. Second, by being a less constrained model, PES provides the opportunity to explicitly examine important issues in specification design. These issues are addressed in the next two paragraphs.

Step-wise refinement is simply functional decomposition where the refinements are constrained to be sequentially applied. CPT can be constrained to just alter single components, and thus simulate step-wise refinement. If CPT were allowed to carry out refinements in parallel, it would be a constrained version of PES where the meaning of the specification could not be modified. Yet, parallel CPTs would require a complex merge scheme if refinements altered interfaces incompatibly along parallel lines. Constraining the parallel CPT model to refine single components results in the functional decomposition model. PES can be considered a parallel CPT model which also allows changes in the meaning of the specification. As such, by constraining PES properly, it can simulate any of the other three models of development.

Two of the difficulties of PES are: validation of reformulations and merging of specifications which were developed in parallel. Yet, the unconstrained nature of PES allows one to explore these issues and to provide automated support for their use. Below is a summary of the salient issues.

- Confirmation

It should be assured that each group of actions[3] results in moving the current state of the specification closer to its final acceptable state.[4] In terms of the PES

---

[3]By action, I refer to any operation available to the analyst within the specification design environment.

[4]Actions should be allowed to move away from local minima, cf. [18].

model, the results of each action should be in support of the compilation of some goal or policy of the requirements or the integration of the same. This is an instance of the frame problem. After any action one must determine which specification constraints are consistent and which have been invalidated by an action (cf. chapter III).

- Elaboration Strategy

  An elaboration strategy must: (1) determine which goal is to be compiled into the specification, (2) where the specification should be altered, (3) which of many possible reformulations should be applied, (4) when should a specification be split and modified in parallel, (5) how far should these specifications be allowed to diverge, and (6) when should they be merged back together.

- Merge Strategy

  A merge strategy must: (1) determine which specifications are to be merged, (2) the order in which they are to be combined, (3) notice conflicting specification components and the goals from which the conflicts stem, and (4) determine how to resolve the conflicts.

- Reuse

  How can analysis carried out on one part of a specification, be reused on a similar but different part? Also, figure 4 illustrates the merging of specifications that were developed in parallel from the same initial specification. Can one also merge in disparate specifications, i.e., ones developed from distinct roots?

- Multiple Analysts

  PES supports parallel elaborations, hence, sets of elaborations can be carried out by separate processors. If multiple analysts are allowed, the elaboration strategy must deal with issues such as: (1) who can modify the requirements (since this is global) and to what degree?, and (2) who makes the decisions of when to split off or merge specifications.

The rest of this thesis explores these issues in detail.

5. Why Parallel

At this point one may rightfully question the need for parallel elaborations, based on the inherent difficulties in merging. However, initial exploration of the merge problem suggests fruitful research will result. Moreover, arguments against parallel elaboration can be seen as arguments against the linearity assumption in planning. Here are the basic arguments for parallel elaboration:

- Multiple Analysts

  A group of analysts can apply reformulations at the same time. Difficulties are introduced if they are allowed to modify the meaning (global requirements) at the same time.

- Reduced Cognitive Demands

  Components can be designed in isolation of locally irrelevant parts of the specification. Hence, parallel elaboration is a form of abstraction.

- Reusability by Design

  By designing components in isolation, the designer is not as likely to include unnecessary assumptions which limit the reusability of the design.

- Goal Conflict Detection Unnecessary

  The linearity assumption assumes goals are independent. So, one need not attempt to detect conflicting goals. Instead, goal conflicts are determined by conflicts in their compiled behaviors.

The most powerful argument against parallel elaborations is that many resources can be wasted on creating a specification aspect that has diverged too far to be integrated. This can occur with a poor elaboration strategy. PES is not a panacea for development, but rather a rich speculative paradigm.

6. Summary

Parallel Elaboration of Specifications (PES), is a model of development that subsumes step-wise refinement, functional decomposition, and Correctness Preserving Transformations (CPT). However, as a less constrained model of development, it must explicitly address the issues of confirmation of action, parallel elaboration, and merging of divergent specifications.

CHAPTER III

KNOWLEDGE-BASED PES

This chapter describes a semanticly rich design model upon which an "intelligent" environment (Oz[1]) is being created to assist in the development of specifications. Knowledge-based Parallel Elaboration of Specifications (KBPES) is the result of integrating knowledge-based design techniques into the PES development model.

KBPES has three domain models from which it derives its knowledge of specification construction: the design model, the development model, and the specification model. The design model is still too speculative to be presented. However, the specification and development models are described. A brief discussion of the benefits of "power tools" precedes the summary.

1. Specification Model

In this section, the domain dependent and independent aspects of the KBPES specification model are presented. This model is discussed prior to the development model because the development merge operator makes explicit use of the analysis made available by this model. This specification model serves to (1) define a syntactically well-formed specification, (2) define reformulation transformations for a specification language, and (3) recognize conflicting specification constraints.

---

[1]Oz is not fully implemented, but its architecture is based directly upon the KBPES model described in this chapter and illustrated in figure 1.

1.1. State

A specification in KBPES consists of a set of relations as shown in figure 6. These relations can be derived from the specification language, and in fact, the analyst need never see them.

A set of such relations is a specification. It is these sets that the system modifies. However, the system must also know the semantics of these relations to enable it to recognize conflicting specifications. Figure 7 illustrates the simple specification of the semantics for four relations used in the Oz system.[2]

These relations are often called constraints. For example, the relation

Cardinality(P,**natural**)

constrains the number of patients (P) to be a natural number. Relations such as these constrain possible implementations and should not be confused with qualitative con-

---

Type(P,**process**)   Type(M,**process**)   Type(N,**process**)
Type(PM,**uniarc**)   Type(MN1,**uniarc**)   Type(MN2,**uniarc**)
Type(NM,**uniarc**)
Connected(P,PM,M)   Connected(M,MN1,N)
Connected(M,MN2,N)   Connected(N,NM,M)
ArcValue(PM,Pvalue)   ArcValue(MN1,Mvalue)
ArcValue(MN2,Notify)   ArcValue(NM1,Safe)
Type(Pvalue,**relation**)   Type(Mvalue,**relation**)
Type(Notify,**relation**)Type(Safe,**relation**)
Relation(Pvalue,(P,VALUE),**random**)
Relation(Mvalue,(P,VALUE),**random**)
Relation(Notify,(P),**random**)
Relation(Safe,**boolean**)
Cardinality(P,**natural**)   Cardinality(M,1)   Cardinality(N,1)
Effects(Mvalue,[Safe(**true**)|Safe(**false**)],nil))
Effects(Safe(**false**),Notify,nil)

Figure 6. A Set of Specification Constraints.

---

[2]These and other relations are used, but explicit representation of their semantics and consequently the automated support of merging has yet to be implemented.

```
(Type (symbol x oneof
        {process,uniarc,biarc,relation,integer,natural,boolean})
     (M x 1))
(Effects (relation x relation x oneof {relation,nil})
         (1 x M x M))
(Connected (process x arc x process)
           (M x 1 x M)
           (symetric))
(Ieffect (relation x relation)
         (M x M)
         (transitive
                 (If (Effect(x,y,z) & Effect(y,v,w))
                  Then (Ieffect(x,v)))))
```

Figure 7. Semantics of Specification Constraints.

straints.[3] Henceforth, constraints will be used synonymously with the term relations with regard to the representation of a specification.

Currently, in Oz both static and dynamic constraints are represented. Type, Connected, and Cardinality are examples of static constraints. Dynamic constraints constrain behaviors over time. For example, the Effects relation (see figures 6 and 7) specifies that when the first relation is true, the second and third must unify. This kind of rule is used to specify what must be true after an event. Constraints used in conjunction with symbolic execution determine the internal consistency of the specification[55].

As noted earlier, the analyst need never deal with constraints explicitly, rather the pretty print of them can be manipulated. Figure 8 illustrates the pretty print of an Oz specification. The Diagram and Constraint sections are pretty printed views of the set of relations which make up a specification.[4]

---

[3]Qualitative constraints constrain a variable to range over a value space. In qualitative analysis, processing of a problem solving component continually narrows value spaces of several such variables until a set of consistent solutions is found.

[4]Oz does not currently represent the relations explicitly, hence the pretty print of them is a manual process, i.e., the analyst formats the Constraints and Diagram sections.

Figure 8 also shows the two other parts of an Oz specification. Comments are contained in an unformated text section and are ignored by the system. The Motivation section is also currently unformated ignored text. But, it has a more formal meaning in terms of the KBPES model as will be described in the section on goals.

The development structure of a particular specification is depicted as shown in figure 9. Boxs denote specifications and diamonds denote elaborations. Boxes are created by the **Elaborate** operator which simply makes a copy of a specification and links the two. After which, reformulations are applied to alter the specification. This set of reformulations, an elaboration, is recorded in the diamond.

1.2. Domain Independent

Domain in this model refers to both the specification language and the problem domain of the specification. Independent of either of these domains is the syntactic "well-formedness" model of specifications and the general data and control flow model on which well-formedness is predicated.

The six following features of a specification, referred to as specification sins, represent an initial model of specification well-formedness. Meyer[36] , in arguing for formal descriptions, discusses problems that must be dealt with in any specification: noise, incompleteness, overspecification, inconsistency, ambiguity, and wishful thinking. Abstractions in the specification which have no correspondence in the real problem environment are noise. Incompleteness is the absence in the description of relevant entities in the problem environment. Features in the specification which do not deal with the problem, but rather constrain the possible software solutions are instances of overspecification. Internal specification inconsistency results if two or more abstractions are incompatible. Finally, wishful thinking describes problem features which cannot be a realisticly validated part of the solution.

Note that many of these errors have direct analogies in compiler terminology: unused data (dead variables), and disconnected code (dead processes); missing inputs

Figure 8. An Oz Specification.

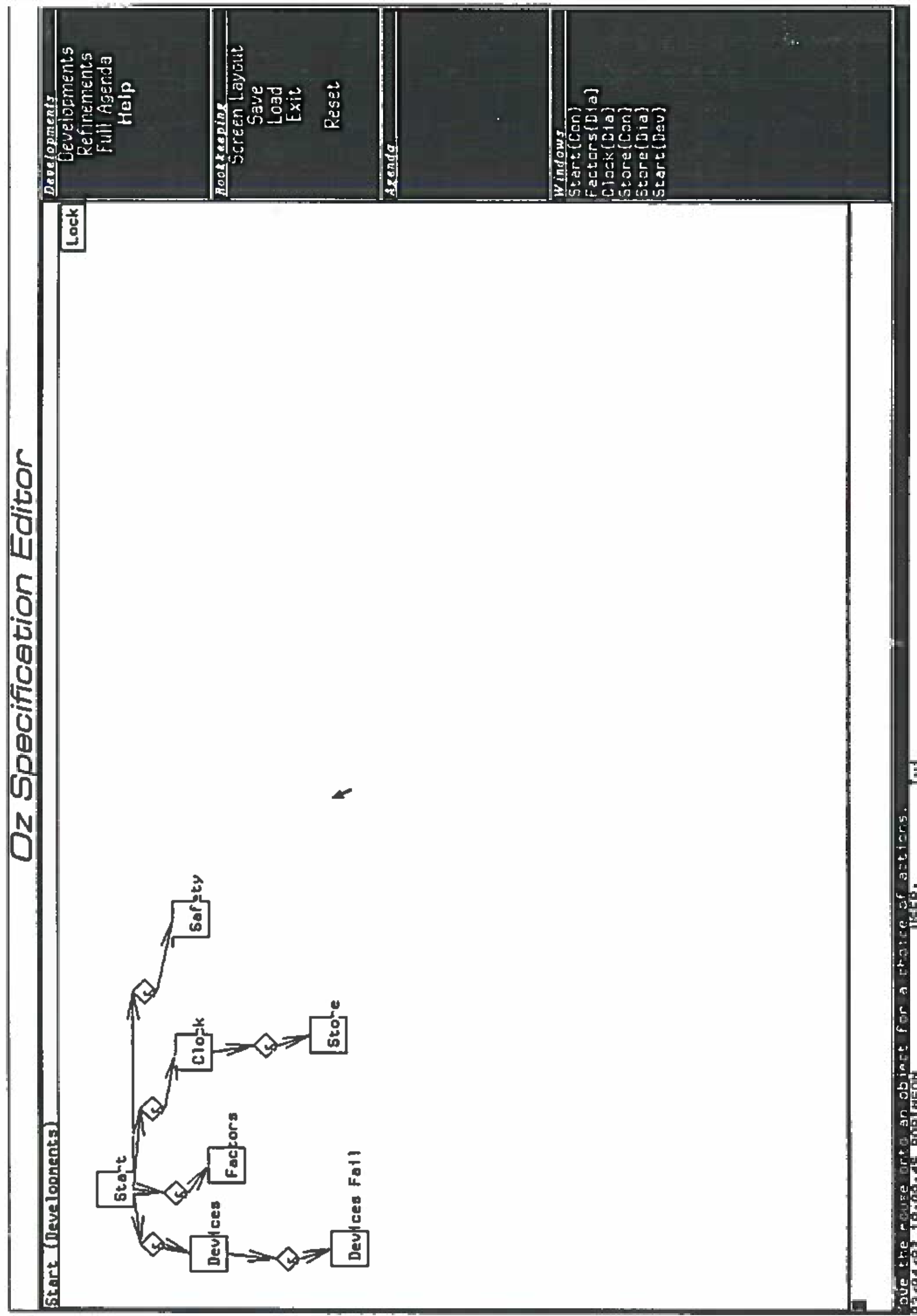


Figure 9. An Oz Specification Development Structure.

and outputs (parameters); interface mismatches (typing errors). Hence, recognizing such errors is predicated on analysis of data and control flow as often found in good compilers[2].

1.3. Domain Dependent

The domain dependent portion of the specification model is concerned with (1) the syntax and semantics of the specification language and (2) the semantics of reformulations applied to the language. Predicated on the above domain independent flow model, this partition serves to apply specification transformations and recognize specification conflicts.

The syntax of a specification language, in this model, is perceived as a way to view the specification as represented in a relational model. As shown in figures 6 and 7, specifications are represented as a set of relations, where the relations have explicit semantics. The recognition of specification conflicts is the process of discovering inconsistent relations.

Specification operators are transformations which (1) satisfy elaboration subgoals and (2) have attached procedures to maintain the consistency of the specification. An elaboration subgoal has the intent of changing a portion of a specification by introducing a compiled goal's behavior. But, due to the interdependent nature of the compiled goals represented in a specification, such changes may *effect* a large segment of the specification. Specification operators rely on the semantics of the relational representation of specifications and self-knowledge of modification effects to apply adjustment procedures which insure consistency of the specification after transformations.

1.4. Analysis

The above analysis is used to insure the internal consistency of the set of relations that represent a specification. Insuring that specification transformation opera-

tors do indeed correctly compile goals and policies is one (difficult) way to insure the specification represents the desired behaviors. Other analysis techniques which attempt to elucidate the behaviors represented in a specification are: symbolic execution, rapid prototyping, and usage scenarios. The knowledge to carry out the first two of these would be represented in this specification model. A usage scenario is a form of symbolic execution in which a set of useful problem specific test cases are evaluated. These test cases would be attached to domain dependent goals and policies in the design model if they were represented in KBPES.

2. Development Model

The development model has three major tasks: (1) manage the development structure, (2) merge specifications, and (3) resolve conflicts.

2.1. Manage Development Structure

A belief maintenance system[5] manges the development lattice. All specification relations are stored in a global database. Each node of the development structure represents a complete specification and a view of this global database.

Attached to each relation is the set of worlds in which it is valid. Also attached are the goals and policies from which it was derived. Goals, policies, and reformulation operators justify the existence of relations. Upon the realization that goals or policies need to be modified, the justifications can be used to determine which relations in each node of the lattice will be effected.

Justifications also allow default assumptions to be retracted efficiently. Such assumptions can be created by the analyst or the automated assistant during design. Many of the suggestions generated by Oz in the next chapter depend on such default

[5]This analysis is based on the intended use (not implemented) of ATMS[11] , but other such systems could be used.

assumptions.

2.2. Merge

Specifications are merged by merging their views (worlds), at which time corresponding concepts and conflicts must be noted. Assuming that distinct names reference distinct concepts and common names reference common concepts (i.e., in divergent specifications), correspondence of concepts is a simple lexical operation. However, such an assumption places stringent demands on KBPES model and reduces the applicability of it to a multiple analyst environment.[6] On the other hand relaxing this assumption requires a general concept recognition capability if correspondences are to be recognized automatically. Recognition could be made a manual process, however, this can be an enormous task for the merge of disparate specifications.[7]

Once correspondences are found and noted, the specification model determines if there are any constraint conflicts. These are then passed to the development model where they are recorded and asserted into the belief system.

2.3. Conflict Resolution

Automation of conflict resolution is predicated on (1) the development structure, (2) reformulation operators, and (3) the *aggregation assumption.* For example, referring back to figure 4, the merge resulted in conflicting assertions about the source of values being examined by the device (D). In one line of elaboration, a process (PF) was spliced into the dataflow between the patient (P) and the device. The other line of elaboration did not modify this dataflow, but rather the dataflow from the device to the monitor. Using the aggregation assumption, that the merge of specifications is

[6]However, the system could alert the analyst of other uses of a name every time he creates one. But, this would likely be a long list.

[7]Oz currently employs a manual scheme.

intended to represent an aggregation of behaviors,[8] and the knowledge that splicing is a reformulation of a dataflow, the conflict is resolved by the removal of the constraint specifying the source of devices read as the patient.

3. Power Tools

Programmer's have recognized the power of interactive programming environments and have applied contemporary hardware technology in their creation. Sheil graphicly illustrates the need and use of these environments[43]. Program browsers depict the calling dependencies between subprograms. Sophisticated debuggers show various views of the run-time environment previous to the detection of an error, facilitate its fixing, and then allow continuation of processing. Sophisticated editors such as Teitelman's Programmer's Assistant[50] , use structured commands to view and modify programs. All of such tools use bit-mapped graphics to increase the bandwidth of information flow between the analyst and the tools.

Much of the example presented in chapter IV is motivated by my envisionment of more complex knowledge based tools such as PegaSys [37] and the programmer's apprentice[52] for specification design. But, before such a tool can be constructed, a formal model of design must be set forth. Only then can one construct power tools which understand the semantics of the model. This chapter represents my initial effort in this regard.

4. Summary

KBPES is a semanticly rich model of specification development which incorporates issues of knowledge-based design into its supportive design, development, and specification domain models. Next, the use of KBPES is illustrated via the development of the patient monitoring system. This example was constructed using the Oz environment which only partially implements the utility of the KBPES model.

---

[8] Additions or deletions of behaviors can be done prior to, or after a merge.

CHAPTER IV

CASE STUDY: A PATIENT MONITORING SYSTEM

This chapter illustrates the use of the KBPES model. The Oz environment, which partially implements KBPES, is used to develop the patient monitoring example referred to earlier.

Currently, the Oz implementation only keeps track of the development structure and applies simple specification operators. The consistency maintaining procedures associated with the specification operators are being developed. Their application has not been automated, but the few rules used in the development of the patient monitoring example of this chapter are shown in appendix A.

Similarly, the development model merge operator has not been automated. Instead, the rules of appendix B were manually applied. Appendix B also contains a description of the simple binary merge strategy used.

With the above caveats concerning the implementation status, the patient monitoring example is shown with the perspective that the system is actually applying rules of appendices A and B rather than reality of manual application by the analyst. It is shown this way in the hopes that it will be simpler for the reader to understand the intended demarcation of tasks between the system and the analyst.

The intent of this chapter is to show that automation of KBPES is feasible, even though it has not been completed. The existence of Oz demonstrates that aspects of KBPES can be automated and provide for a useful specification environment. However, future research goals do include the development of a robust merge operator.

The next section provides an introduction of the patient monitoring system to be specified. The following three sections describe Oz, its simple specification language, and the specification operator hierarchy. Section 5 summarizes the requisite KBPES

terminology of chapter III. Sections 6 and 7 are the heart of the example, first showing simple elaborations of the specification and then the merges of divergent specifications. Possible continuations and a summary are presented in sections 8 and 9, respectively.

1. Introduction

In [17] , Feather applied PES to a Patient Monitoring System which has also been used by other researchers in specification[46, 56]. In fact, Feather's transformational model of design inspired this part of our research.

The following case study is a recreation of Feather's development using the Oz specification system. (Feather's corresponding GIST specifications appears in appendix C.) His development does not complete the specification, and correspondingly, the Oz development does not result in a finished specification. This study was done to promote comparision and facilitate understanding of the KBPES model and is not meant to suggest an optimal development or specification.

Feather used the statement of specification appearing in [46] as his problem statement. It is reproduced here:

> A patient-monitoring program is required for a hospital. Each patient is monitored by an analog device which measures factors such as pulse, temperature, blood pressure, and skin resistance. The program reads these factors on a periodic basis (specified for each patient) and stores these factors in a data base. For each patient, safe ranges for each factor are specified (e.g., patients X's valid temperature range is 98 to 99.5 degrees Fahrenheit). If a factor falls outside of a patient's safe range, or if an analog device fails, the nurse's station is notified. [46]

Feather's specification was developed from a simple initial stage and elaborated[1] via transformational operators toward a more complete specification. The development of this specification by Oz will correspond to Feather's. The discussion will be concerned with the dimensions of name, motivation, transformational operators, and

[1]By elaboration, I mean an ordered set of transformations motivated by a requirements goal or policy.

software design knowledge:

(1) *Name*

For reference, each elaboration is named.

(2) *Motivation*

Each elaboration is justified in terms of domain needs according to the problem statement.

(3) *Operator*

The use of each transformational operator(s) and the support it provides at each elaboration is described.

(4) *Knowledge*

The software design knowledge needed by each operator to provide the support described will be discussed. However, as much of the knowledge is repeatedly used, it will only be described once, after which it will simply be referenced.

Feather also briefly characterized transformations in terms of domain independent modifications as described by Goldman[26] (e.g., structural, behavioral, temporal changes). This is an area which I am also pursuing, but is not discussed here.

Also, Feather described continuations which may be necessary to completely cover the problem description, but which were not further described. These are mentioned at the end of the example, see § Continuations.

2. Oz

Oz is a specification design tool which embodies the KBPES model. It is an interactive and graphical system. Figure 10 shows the display during a session. Its operation is roughly modeled after the ZMACS editor on the Symbolics Lisp Machine[53].

The large pane in the center can be divided into any number of graphic and text windows. Graphic windows display the Diagram part of a specification and text win-

dows display Comments, Constraints, and Motivations.

The pane at the bottom is used for limited forms of I/O. To its right, at the bottom, a pane displays windows which may not be currently visible; clicking on these mouse sensitive text items causes their windows to be displayed.

The Screen Layout command allows the analyst to completely specify the layout of specification windows within the central pane. The Agenda pane simply allows the user to postpone commands; clicking on these items causes the command to be executed. The uppermost right pane provides various views of the development structure.

3. Oz Language

The "language" used to illustrate my analysis of the patient monitoring system is a subset of GIST[33] chosen to highlight features that are used by the specification model, e.g., control and data flow relations. Familiarity with either language is not expected, as relevant details of both will be discussed. However, most of the discussion will be centered around the Oz specifications, so here I clarify the basics of the model.

An Oz specification consists of four parts: the diagram, the constraints, the comments, and the motivation. The diagram section is a graphical depiction of the agents (circles), objects (boxes), and dataflow relations (arcs) of a GIST specification; essentially a dataflow diagram. The constraints section specifies constraints on the graphical entities. There are four sorts of constraints and they are displayed in the order of: static, type, assertion, and assertion effect. Throughout the example, new and modified constraints are shown at the bottom. The four sorts of constraints are defined as follows:

- Static Constraints

  Static constraints describe nonchanging characteristics of entities across time. For example, to say that there is only one Monitor we can use the static con-

Figure 10. The Oz Design Tool.

straint, $|M| = 1$.[2]

- Typing Constraints

  Types are like static constraints in that they describe relations that are constant across time, but they are definitional (i.e., type defining). Some examples are:[3]

  VALUE | **type**;
  [PVALUE | **relation**(P,VALUE) || **random**];

- Assertion Constraints

  An assertion constraint is of the form <LHS> → <RHS>, where the LHS and RHS are relations. Whenever the LHS relation is asserted the RHS relation must also be asserted without resulting in inconsistency. In the following, MPVALUE results in either SAFE being true or SAFE being false, nondeterministicly.

  MPVALUE → [SAFE(**true**) | SAFE(**false**)];
  SAFE(**false**) → NOTIFY(P);

  Such a relation can be considered an abstraction of both control and dataflow relations. For example, the arrow (→) says that somehow, if SAFE(**false**) is asserted then NOTIFY(P) must happen (i.e., be asserted).

- Indirect Assertion Effect

  Indirect assertion effects summarize chains of assertion effects; they show the implicit flow of effect relations in the specification. Indirect effects are determined solely by the system via propagated control and dataflow relations.[4] An example (used later) is:

  (PVALUE, DVALUE, MPVALUE) →→ NOTIFY;

  This constraint is a summation of four constraints, each of which directly determine the next, thus forming a chain, i.e.,:

---

[2]This particular problem constraint is not part of the corresponding GIST specification, but it and several others were added to clarify the example.

[3]In Oz one uses brackets ([]) to enclose relations rather than the braces ({}) of GIST because braces are reserved for set notation.

[4]This part of the system has not been implemented yet.

PVALUE → DVALUE;
DVALUE → MPVALUE;
MPVALUE → SomeValue;
SomeValue → NOTIFY;

The value that actually determines NOTIFY (SomeValue), is not listed as an indirect assertion effect, but instead is a direct assertional effect, (i.e., SomeValue → NOTIFY;).

Constraints are defined in the constraint section, which is a ZMACS text window. My intent is to specify most constraints graphically. Those constraints not suitable for graphical notation will be modified by specialized operators in a way similar to that of a structured editor[49]. But for now, the constraint window is simply a text editor; the system relies on the user to form syntacticly correct statements.

The comments and motivation sections are also simply textual annotations. The Motivation section was described in more detail in chapter III.

4. Specification Operator Architecture

Specifications are a set of relations. When a relation is modified, one must determine which other relations are effected. Given a simple language, one can completely enumerate the types of modification operators and their effects to other parts of a specification (cf. [5]).

Figure 11 illustrates the modification of a complex specification. When part of a specification is modified, one does not wish to completely search through all the possible constraint interactions. Rather, each operator should know which constraints were likely to be violated, then the search for conflicts can be directed.

Figure 12 shows the hierarchy of specification operators used in the patient monitoring system. It is a composition hierarchy: each operator is composed of the operators below it in the hierarchy. Consequently, when an operator is applied, it must execute operators and rules below it in the hierarchy. By attaching these meaning maintenance rules to operators in the hierarchy, one need not check all relational

Figure 11. Specification Modification as a Set of Changing Constraints.

interactions for possible conflicts.

As described in chapter III, these rules attached to the specification operators serve to maintain consistency of the specification after the application of an operation. They are based on a general control and data flow model and may be specialized by the problem domain.

The set of operators and associated rules of figure 5 is sufficient to handle Feather's example, but it seems clear that it is only a small subset of a robust set. Moreover, I offer no cognitive justification for these operators. In fact, such justification may not be applicable since the analyst will eventually apply the goal intergration operators of the design model. Those operators will then map to these specification modification operators.

5. KBPES Terminology

Before commencing with the example, I would like to make a clear definition of each of the following terms: elaboration, transformation, operator, and adjustment.

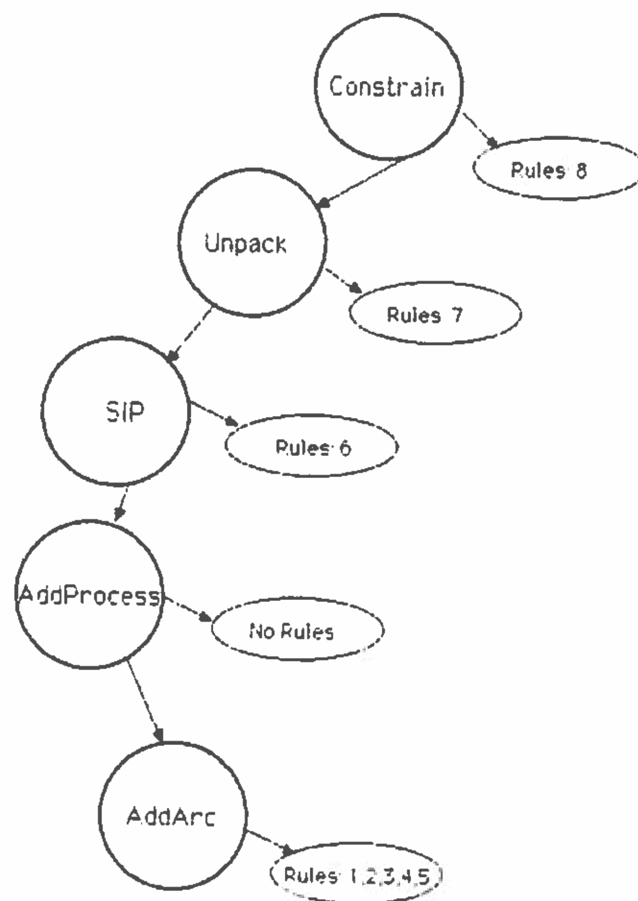


Figure 12. Initial Hierarchy of Specification Operators.

An *elaboration* is an abstract operator which directs modification of part of a specification in order to compile some goal or policy into it. A *transformation* is the way one modifies specifications in KBPES. Transformations may consist of several specification *operators*. Specification *operators* are alter the specification. After the application of an operator, an *adjustment* may be necessary to ensure the quality of the specification with regard to constraint conflicts and specification sins (cf. chapter III).

6. Simple Elaboration

In this section, each of the GIST elaborations found in[17] (summarized in appendix C) are presented as an Oz elaboration and discussed. Figure 13 shows the initial Oz specification. It represents the three major agents of the system:

Patients (denoted as an oval, labeled P)

There may be several patients, each of which has a value which is a crude measure of health.

The Monitor (denoted as an oval, labeled M)

There will be one monitor in the final implementation. This is the component which is to be implemented. The monitor watches the patient's values and notifies the nurses station if a value becomes unsafe.

The Nurses Station (denoted as an oval, labeled N)

The single nurses station defines what are the unsafe values for each patient and receives warnings from the monitor.

The initial specification is a simple abstraction of the system's intended behavior. Some of the complex details that have been ignored are: devices, patient factors, and periodic monitoring. We will explore each of these as the development proceeds. In particular, the six basic elaborations which will follow (in order) are:

(1) A device will be spliced into the dataflow between the patient and the monitor. This will retract the simplification of having the monitor directly read patient

Figure 13. Initial Oz Specification.

values by using an intermediate device.

(2) Devices may fail, and when they do, the patient values they pass to the monitor will no longer be valid. When a device fails, the nurse's station will be warned. This elaboration must sequentially follow the introduction of devices.

(3) A patient's value actually consists of a set of factors, each of which must be verified as safe or result in a warning to the nurse's station. Here, value is decomposed into its constituent factors.

(4) The continuous process of the Monitor is made periodic by the introduction of a clock.

(5) Safety will depend on the patient and the patient's value, rather than just a value.

(6) Storing of patient values, at this point, has been left out. Adding it illustrates how transformations can be used to maintain the specification.

In the next six subsections, each of the six elaborations will be described in terms of: (1) elaboration name, (2) elaboration motivation, (3) operator(s) used to carry out the elaboration, and (4) specification model knowledge used in adjustments after a specification operator application. The elaborations will: (1) add devices, (2) add device failure, (3) add patient factors (e.g., pulse, blood pressure), (4) introduce a clock, (5) make safety depend on a patient, and (6) store patient values in a database.

Figure 14 illustrates an overview of these elaborations. It shows the Oz depiction of the development structure. Boxes represent specifications and diamonds represent the operators used to elaborate a specification.

6.1. Elaboration 1: Introducing Devices

Our initial view of a monitor is too simple: a monitor cannot directly read patient values, but instead it must read values gathered by *intermediate devices.* Such devices continually observe the patients' values and display it unchanged to the moni-

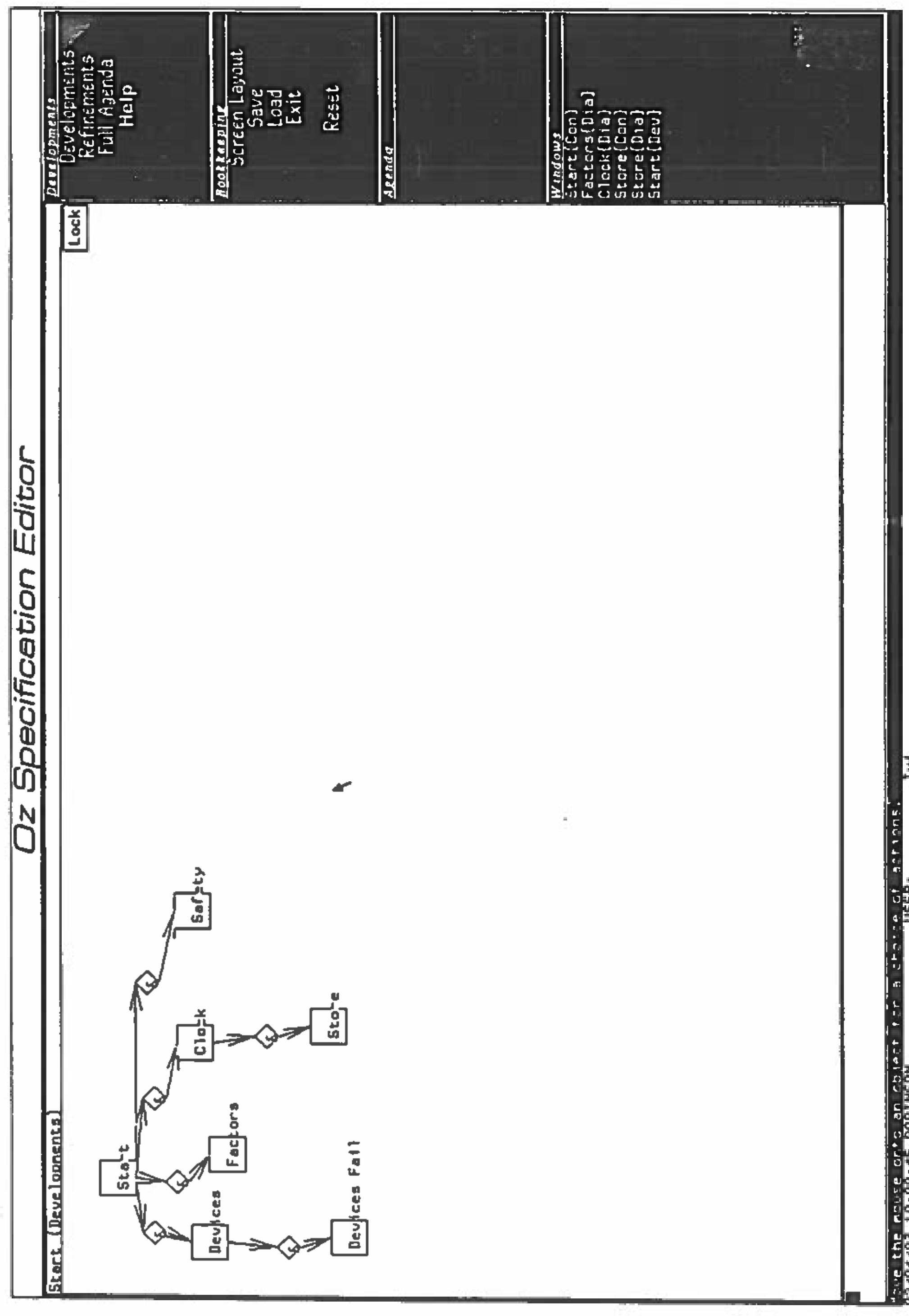


Figure 14. Illustration of the 6 Basic Elaborations.

tor. Hence, we will have to forego our simple view and add devices. Figure 15 shows the results of adding devices to the initial specification. The elaboration is one of splicing in a new agent (device) between the patient and monitor; the device passes on the PVALUE unchanged as DVALUE. The monitor is modified to use the DVALUE to pass as MPVALUE to SAFE rather than PVALUE. Changes are listed at the bottom of the constraint section.

(1) *Name:* Devices.

(2) *Motivation:* The dataflow between the Patient and the Monitor is overly simplistic; this elaboration refines the model of the Monitor-Patient interaction.

(3) *Operator:* The SIP (Splice In Process) operator was used to transform the initial specification to that shown in figure 15. SIP splices a process into a dataflow and then queries the analyst about the effect of this splice to the resulting specification. For example, after determining the name of the process (D) and the outward arc (DVALUE), SIP attempts to substitute DVALUE for PVALUE through the rest of the dataflow, so as to maintain the meaning of the specification as much as possible.

Preserving the meaning of the specification may seem at odds with the meaning changing operators which are allowed in KBPES. However, while most operations are meant to *change* just a small part of the specification, they *effect* a large part. By showing the analyst what must be done to maintain the meaning of the specification, he is made aware of the implications of each change made. Oz queries the analyst to confirm that larger effects are understood.

(4) *Knowledge:* SIP relies on knowledge of physical devices connected by communication lines, i.e., abstract data and control flow. It does seems possible to generalize this notion to any two communicating agents. In this case, SIP proposed preservation constraints to the analyst in the form of the following six queries.

Figure 15. Oz Modifications to Introduce Devices.

(1) $|D| = |P|$; ?
(2) PVALUE $\rightarrow$ [DVALUE = PVALUE]; ?
(3) DVALUE $\rightarrow$ [MPVALUE = DVALUE]; ?
(4) PVALUE $\twoheadrightarrow$ MPVALUE; ?
(5) (PVALUE, DVALUE) $\twoheadrightarrow$ SAFE; ?
(6) (PVALUE, DVALUE, MPVALUE) $\twoheadrightarrow$ NOTIFY; ?

They can be paraphrased as:

(1) *Does every patient have his own device?* Or is there a single device that reads all patient values as illustrated in the example of chapter II?

(2) *Does DVALUE carry the same information as PVALUE?* Or does the device alter the form and content of it?

(3) *Does MPVALUE use DVALUE in the same way as PVALUE previously?* Or does the monitor alter MPVALUE in some way?

(4) *Does PVALUE indirectly determine MPVALUE?* Yes, if 2 and 3 are confirmed.

(5) *Does PVALUE determine DVALUE which indirectly determines SAFE?* Yes, if 2 and 3 are confirmed.

(6) *Does a chain of effect that start with PVALUE, go through DVALUE and MPVALUE, and indirectly (through SAFE) determine NOTIFY?* Yes, if 2 and 3 are confirmed.

In this example, each constraint was confirmed by the analyst.

Operators such as **SIP** determine which constraints should be added, deleted, or modified via rules of typicality based on notions of abstract data and control flow. For instance, the above proposed constraint, $|D| = |P|$, was determined by *Corresponding Processes*, the others by *Propagate Spliced Relations.* Once such constraints are determined, operators call **Constrain** with the suggested constraints. **Constrain** allows the analyst to confirm, add, delete, and modify constraints. It can also be called directly by the analyst. Once more of the specification model is represented, **Constrain** will carry out more of the confirmation process autonomously.

The above proposed constraints were generated by two rules which are part of the specification model. *Corresponding Processes* suggests that when a process P is spliced into a communication flow between two sets of processes, the number of processes spliced in, $|P|$, should be equal to the number of process at the source. This rule, and others listed in the appendix, are specific to the example presented, e.g., SIP assumes a single direction of flow. It is obvious that such rules can be generalized. For example, to handle bidirectional flow one could specify that if both source and sink had the same number of processes, then splice in that number; otherwise, splice in the greater of the two numbers. Domain specific rules could also be used at this point to further refine Oz's notion of the typical number of processes which should be spliced into the flow based on the types of agents and other relevant problem environment concerns. I am considering the addition of such knowledge to subsequent versions of the system.

Propagate Spliced Relations simplity states that any relations existing previous to a splice across the flow of communication should exist after the splice. This is arranged by substituting the new spliced in values into the existing relations. In this case above, originally PVALUE → [MPVALUE = PVALUE], but after the splice DVALUE → [MPVALUE = DVALUE].

6.2. Elaboration 2: Introducing Device Failure

Devices may fail, and when they do, DVALUE may no longer reflect PVALUE accurately. Thus, one must introduce the notion of device failure into the specification. This elaboration necessarily follows the introduction of devices (see figure 14). As described in appendix C, Feather does this elaboration in two steps, whereas I did it in one as shown in figure 16. This is because, unlike Feather's derivation, in Oz the application of an operator and its adjustment are encapsulated into a

```
Oz Specification Editor

Devices Fail (Diagram)                                  Lock
  P   Pvalue   D   FAILED   Dvalue   SAFE   M   MPvalue   N   NOTIFY

Devices Fail (Constraints)
|M| = 1; |N| = 1; |P| ≥ 1; |D| = |P|;
VALUE | type;
[PVALUE | relation(P,VALUE) || random];
[MPVALUE | relation(P,VALUE)];
[SAFE | relation(VALUE)];
MPVALUE → [SAFE(true) | SAFE(false)];
SAFE(false) → NOTIFY(P);
PVALUE → [DVALUE = PVALUE];
DVALUE → [MPVALUE = DVALUE];
PVALUE ↠ MPVALUE;
(PVALUE,DVALUE) ↠ SAFE;
(PVALUE,DVALUE,MPVALUE) ↠ NOTIFY;

[FAILED | relation(boolean,VALUE,D) || random];
[NOTIFY | relation(P,FAILED)];
FAILED(true,VALUE,D) → NOTIFY;
FAILED(true,VALUE,D) → [DVALUE ≠ PVALUE];

ZWEI (Fundamental)

Developments: Developments, Refinements, Full Agenda, Help
Bookkeeping: Screen Layout, Save, Load, Exit, Reset
Agenda
Windows: Devices(Con), Start(Dia), Devices Fail(Dia), Devices Fail(Con), Start(Dev)

Move the mouse onto an object for a choice of actions.
03/04/87 23:34:20 ROBINSON        USER:        Tyi
```

Figure 16. Oz Modifications to Introduce Device Failure.

single elaboration.[5]

(1) *Name:* Devices_fail.

(2) *Motivation:* The modeling of real devices is more precisely modeled as devices which may possibly fail.

(3) *Operator:* The analyst applies **AddArc** to add failure behavior to the devices specification. The operator adds the arc, and then it calls **Constrain** with the following proposed modifications to the constraint set:

(1) [FAILED | **relation**(...) || ...]; ?
(2) FAILED(...) → [MPVALUE | NOTIFY]; ?

(4) *Knowledge:* Constraint 1 was generated based on the knowledge that all arcs are represented as a relation. Constraint 2 was generated by the *Determine Effect of Input* rule, which assumes that an in going arc determines the value of one of the arcs in the out going set. The analyst fills in these constraints giving:

(1) [FAILED | **relation**(**boolean**,VALUE,D) || **random**];
(2) FAILED(**true**,VALUE,D) → NOTIFY(P);

AddArc, via the *Distinguish Output* rule, now suggests the constraint:

(3) NOTIFY | **relation**(P,FAILED); ?

In essence, it notes that before only DVALUE was used to NOTIFY, but now both DVALUE and FAILED contribute to notification. It suggests that NOTIFY distinguish between the two in its output. (An alternative would be to have two outputs from M, but it has been determined that FAILED → NOTIFY.)

AddArc does not presume any relation between FAILED and DVALUE (other than they originate at the same source), so the analyst must provide the environmental constraint

(5) FAILED(**true**,VALUE,D) → [DVALUE ≠ PVALUE];

This constraint is used to correspond to the GIST specification, even though

---

[5]An elaboration is made immutable and hence can be reused by other specifications (see § Merge). One need not reuse inconsistent specifications, so Oz incorporates the adjustment before encapsulation.

(5) FAILED(**true**,VALUE,D) → [DVALUE = **random**];

is probably more accurate in this case.

6.3. Elaboration 3: Introducing Patient Factors

Next, I return to the initial specification and elaborate it to introduce the composite of patient factors. Note that this is a case of parallel elaboration: instead of adding a new elaboration to devices_fail in figure 16, the initial specification is decomposed into another line of elaboration. Figure 17 shows the development space after the initial specification has been copied in preparation for modification.

While I could have elaborated the devices_fail specification to account for patient factors, I returned to the initial specification to begin elaborating so as (1) not to confuse the two modifications, and (2) simplify the factors introduction. This elaboration is our first case of a diverging specification; the merge section IV.7 discusses the integration of divergent specifications.

(1) *Name:* Factors. See figure 18.

(2) *Motivation:* Patients do not consists of just one value. Rather, each patient's safety is determined by the conjunction of several factors: pulse, temperature, blood pressure, and skin resistance.

(3) *Operator:* **Unpack** is a specialized sort of **SIP**; it splices in a process into a dataflow in order to refine a data structure by unpackaging it within the specification. After determining the arc and process names, **Unpack** calls **Constrain** with these proposed constraints:

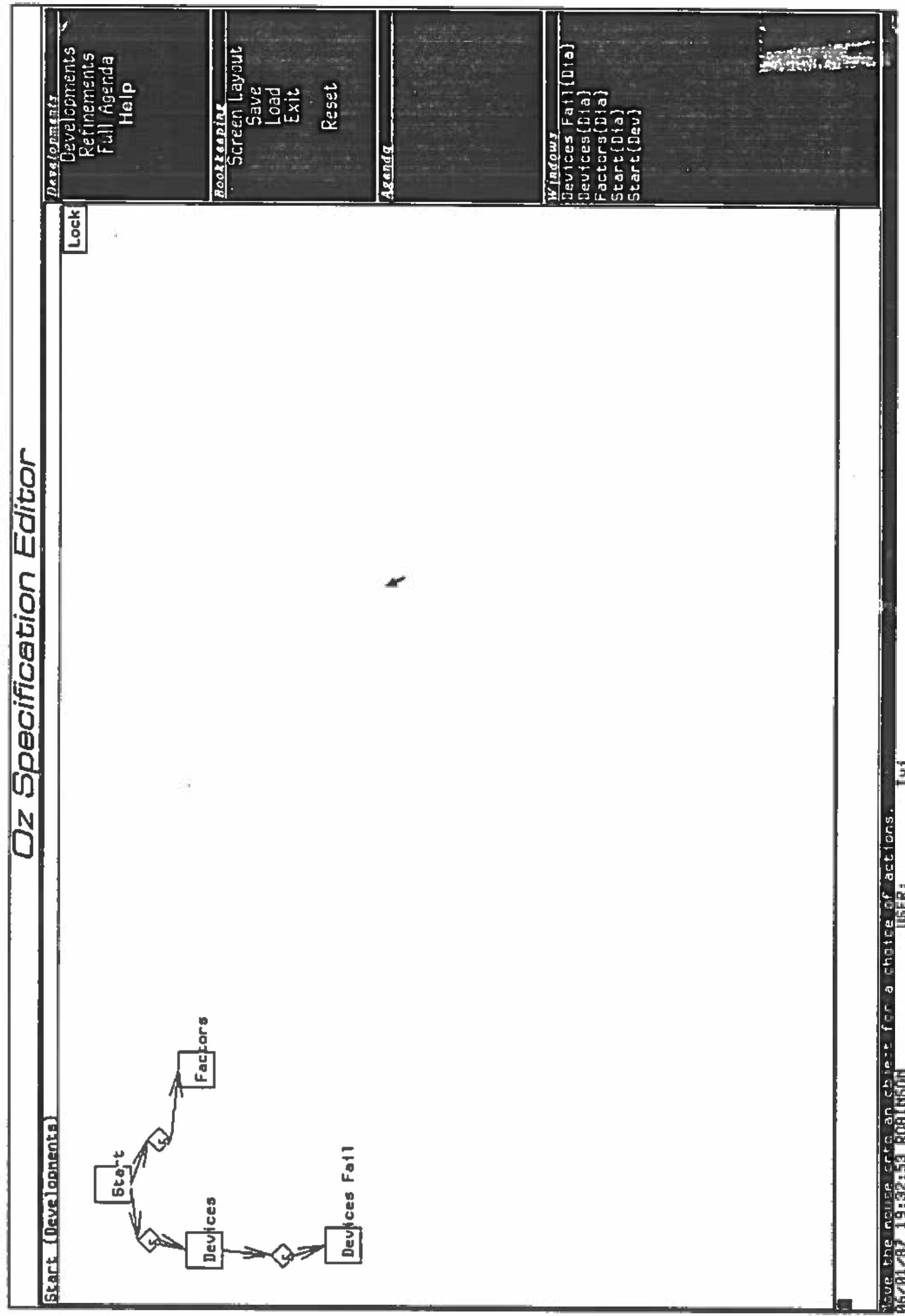


Figure 17. Elaborating Patient Factors in Parallel with Devices.

Figure 18. Oz Modifications to Introduce Composite of Factor Values.

```
 (1) |PF| = |P|; ?
 (2) Set | type = {...}; ?
 (3) |PFVALUE | relation(P,∈ Set,VALUE)|; ?
 (4) |MPVALUE | relation(P,∈ Set,VALUE)|; ?
 (5) |SAFE | relation(boolean,∈ Set,VALUE)|; ?
 (6) PVALUE → |∪ PFVALUE∈ = PVALUE|; ?
 (7) PFVALUE → MPVALUE = PFVALUE; ?
 (8) MPVALUE →
       |SAFE(true,∈ Set,VALUE) | SAFE(false,∈ Set,VALUE)|;  ?
 (9) SAFE(false,∈ Set, VALUE) → NOTIFY(P); ?
(10) (PVALUE, PFVALUE) →→ SAFE; ?
(11) (PVALUE, PFVALUE, MPVALUE) →→ NOTIFY; ?
```

(4) *Knowledge:* The first constraint (1) states that each patient process P should have a corresponding patient factor generator, PF. (Patients are modeled as having an attached process which generates factors.) *Corresponding Processes* suggested this constraint.

The next two constraints (2,3) suggest that PFVALUE is actually a set of arcs which correspond to each element of some yet to be specified set. Constraint 2 is an unnamed set relation which is used in 2, 3, 4, 5, 8, and 9. When I enumerated this set, I also supplied the name, FACTOR.

Constraints 2, 3, and 6, were suggested by the *Determine Composite Set* rule. This rule states that when a process, P, is spliced in to distribute parts of a value in the source: (1) a set of subcomponents should be named, (constraint 2), (2) this set should be used as names of the outgoing arcs of P, (constraint 3), and (3) the sum of these outgoing values is equal to the composite of them in the source, (constraint 6).

The rest of the constraints (4, 5, 7, 8, 9, 10, 11) are just the propagation of 1, 2, 3, and 6 as suggested by *Propagate Spliced Relations.* In this example, the analyst confirmed all the constraints.

6.4. Elaboration 4: Introduction of Periodic Monitoring

The next elaboration, that of introducing periodic monitoring, is also done in parallel with the elaborations of devices and patient values. Figure 19 shows the development space after the Clock specification has been created in preparation of its

modification. The resulting specification is shown in figure 20.

(1) *Name:* Clock.

(2) *Motivation:* This elaboration is motivated by the impossibility in this problem domain of implementing software that reacts instantaneously to changing patient values. Instead, a monitor that periodicly reads device values will be introduced. The period may eventually be based on needs of the monitor, individual patients, and factors. However, this elaboration reads periodic values specific to patients from a database.

(3) *Operator:* The **AddProcess** and **AddData** operators are used in this elaboration, during which a small dialogue between the analyst and the system takes place. First, the analyst uses **AddProcess** to add the clock process, C, and its arcs, after which **AddProcess** suggests that there be a single clock for each monitor, (constraint 1 below), and the more tentative constraint that CLOCK_TIME should somehow be reflected in the Monitor's output (constraint 2 below). These were determined by the rules *Corresponding Processes* and *Determine Effect of Input* respectively.

(1) |C| = |M|; ?
(2) CLOCK_TIME →→ [MPVALUE | CLOCK_TIME →→ NOTIFY]; ?

Both are confirmed, but the second constraint is not specific enough. The analyst uses **AddData** to create a database for patient values, DB, and its arcs. Similarly, the system notes that START_TIME and PERIOD should be reflected in M's output, and warns that there is no in comming arc which is being used as input to generate the outgoing arcs.

(3) |DB| = |M|; ?
(4) START_TIME →→ [MPVALUE | NOTIFY]; ?
(5) PERIOD →→ [MPVALUE | NOTIFY]; ?
(6) Warning: {} → START_TIME.
(7) Warning: {} → PERIOD.

The analyst ignores these warnings (6 and 7), as their resolution is not part of this

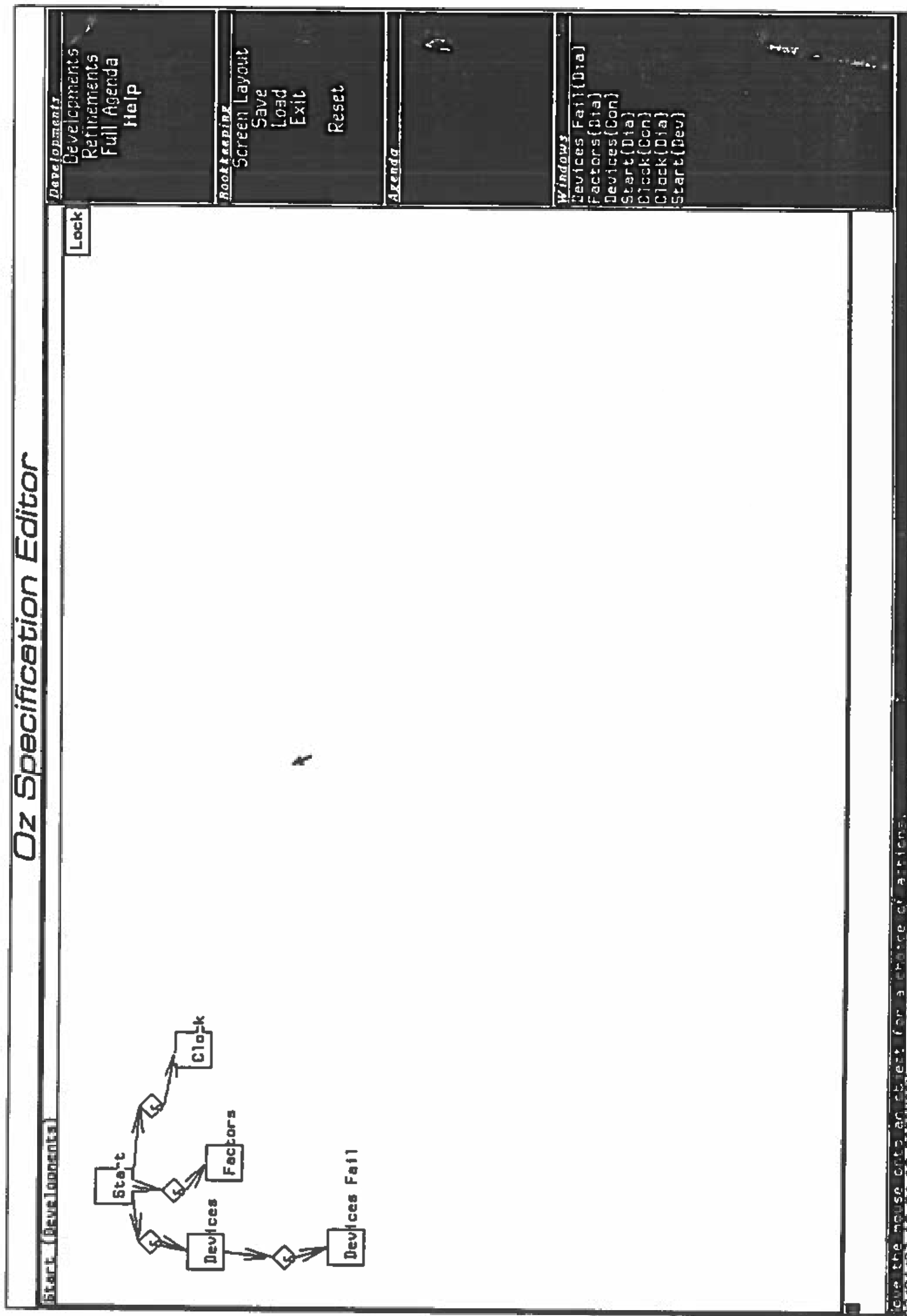


Figure 19. The Development Space After Starting the Clock Elaboration.

Figure 20. Oz Modifications to Have the Monitor Read Periodically.

example.[6]

Now, the analyst is prepared to describe the specific relation between CLOCK_TIME, START_TIME, PERIOD, and M's outputs. Recall that the analyst is presented with the above constraints via the **Constrain** operator. In this case, the analyst specifies that when the clock time is at the end of a patient's period, MPVALUE should be updated. But, rather than call it MPVALUE, the analyst uses **Rename**[7] to rename MPVALUE to be LAST_VALUE_READ, (constraint 11 below).

```
 (8) [CLOCK_TIME | relation(integer)];
 (9) [START_TIME | relation(P,integer)];
(10) [PERIOD | relation(P,integer)];
(11) LAST_VALUE_READ was MPVALUE;
(12) [LAST_VALUE_READ | relation(P,VALUE)];
(13) {PVALUE, CLOCK_TIME(t) = START_TIME(P,t)
                    (an integer) * PERIOD(P,?)}
        → [LAST_VALUE_READ(P,v) = PVALUE(P,v)];
```

Constraints 8-10 are type constraints. Constraint 11 is an annotation linking the previous specification to the current one. In the current specification all occurrences of MPVALUE have been renamed LAST_VALUE_READ. Constraint 11 makes a note of this to: (1) facilitate the analyst's understanding, and (2) provide the explicit renaming relationship which is to be used during the merge of divergent specifications. Constraint 12 is the substitution of LAST_VALUE_READ into the type constraint for MPVALUE.

The analyst makes explicit the relationship between the CLOCK_TIME, START_TIME, PVALUE, and LAST_VALUE_READ in constraint 13. It states the LAST_VALUE_READ should be updated when the CLOCK_TIME is the correct multiple of the START_TIME as specified by PERIOD.

---

[6]During the creation of the initial specification, Oz also gave the warning, Warning: {} → PVALUE, which was ignored. The *Effector of Output* rule is the originator of such warnings.

[7]**Rename** is currently the only structured command that is part of the constraint section editor. Otherwise, constraints are modified in a ZMACS buffer.

(4) *Knowledge:* After the system parses the constraints, it suggests:

(14) LAST_VALUE_READ → [SAFE(**true**) | SAFE(**false**)]; ?
(15) (PVALUE, {CLOCK_TIME, START_TIME, PERIOD})
→ LAST_VALUE_READ; ?
(16) (PVALUE, {CLOCK_TIME, START_TIME, PERIOD},
LAST_VALUE_READ) →→ NOTIFY; ?
(17) (PVALUE, {CLOCK_TIME, START_TIME, PERIOD})
→→ SAFE; ?
(18) Removed: PVALUE → [MPVALUE = PVALUE];
by confirmation of (13).

Constraint 14 is the result of substituting LAST_VALUE_READ into the same relation involving MPVALUE. It is further confirmed by the analysis of constraint 13; it is determined that LAST_VALUE_READ is a processed form of PVALUE, and thus should directly effect SAFE, explained below. Given that 14 is confirmed, 15, 16, and 17 are just propagations of effects. Constraint 18 shows the removal of a previous constraint due to its subsumption by 13.[8]

Constraint 14 was suggested by substitution of MPVALUE by LAST_VALUE_READ and further confirmed by a complex rule dealing with the splicing of processed data into data flow. The rule *Use Processed Data* suggests that if an agent produces a newly modified and/or subset form of a value (e.g., LAST_VALUE_READ is a subset form of what MPVALUE contained) which was previously used by an effector relation (SAFE), then the processed form (LAST_VALUE_READ) should be substituted in the previous relation (e.g., LAST_VALUE_READ → SAFE). More concretely, the example substituted into the rule yields:

[8]Removals due to substitution are not shown.

```
Use Processed Data (paraphrased):
IF      **old relation.
        MPVALUE → [SAFE(true) | SAFE(false)];
        **old effect relation.
        PVALUE → MPVALUE;
        **modified form.
        {PVALUE, CLOCK_TIME(t) = START_TIME(P,t)
                (an integer) * PERIOD(P,?)}
                → [LAST_VALUE_READ(P,v) = PVALUE(P,v)];
        **new effect relation.
        (PVALUE, {CLOCK_TIME, START_TIME, PERIOD})
                → LAST_VALUE_READ;
THEN LAST_VALUE_READ → [SAFE(true) | SAFE(false)];
```

Part of the rule also contains the constraint that the modified form (LAST_VALUE_READ) and the original form (MPVALUE) used must be determined by the same effector (PVALUE). This makes certain that the new value is indeed a modified or subset value of the original.

6.5. Elaboration 5: Link Safety to Patients

At this point, we are ready for the last of the four parallel elaborations. In the initial specification, the safety of a patient is determined solely by a patient's value, without regard to which patient that value is from (i.e., [MPVALUE → SAFE(**true**) | SAFE(**false**)]). It needs to be modified to take into account which patient the value comes from (i.e., [{MPVALUE, P} → SAFE(**true**) | SAFE(**false**)]). Figure 21 shows the resulting specification after the elaboration.

(1) *Name:* Safety.

(2) *Motivation:* Different patients may have different levels of SAFEty depending on such things as their age, gender, etc.

(3) *Operator:* In this case, neither agents, data, nor arcs are modified. Instead, the constraints are modified directly by the analyst via the **Constrain** operator. After the analyst adds

{MPVALUE,P} → [SAFE(**true**) | SAFE(**false**)];

the system resolves a constraint subsumption by removing

Figure 21. Oz Modification to Elaborate Safe to Depend on which Patient It Is.

Removing: MPVALUE → [SAFE(**true**) | SAFE(**false**)];

and then adds the indirect effect:

(PVALUE, {MPVALUE, P}) →→ NOTIFY(P);

(4) *Knowledge*: No new knowledge is needed here; just control and data flow knowledge.

6.6. Elaboration 6: Add Overlooked Storage of Factors

Finally, note that the specification fails to "store these factors in a data base". Such maintenance is as simple as the previous elaborations. One simply transforms the clock specification to STORE patient values after each read.

(1) *Name:* Storage.

(2) *Motivation:* Add coverage of an overlooked behavior to the specification. Figure 22 shows the results of modifying the specification.

(3) *Operator:* **AddArc** is applied to specify the storing of LAST_READ_VALUEs. It calls **Constrain** with:

(1) [STORE | **relation** (...) || ...]; ?
(2) STORE →→ [PERIOD | START_TIME]; ?

which the analyst modifies to be:

(3) [STORE | **relation** (P,VALUE)];
(4) {LAST_VALUE_READ, CLOCK_TIME(t) = START_TIME(P,t)
(**an integer**) * PERIOD(P,?)
→ [STORE(P,v) = LAST_VALUE_READ(P,v)];

Constraint 2 was suggested by *Determine Effect of Input*. But, it was not confirmed since STORE does not effect either of these outgoing arcs. In fact, the specification up to this point does not specify the use of stored values, so constraint 2 is eliminated.

After parsing 3 and 4, the system suggests the following propagated effects which are confirmed.

(5) (LAST_VALUE_READ, {CLOCK_TIME,START_TIME,PERIOD})
→ STORE; ?
(6) PVALUE →→ STORE; ?

Figure 22. Oz Modification to Store Patient Values.

(4) *Knowledge:* This elaboration did not introduce the use of any new knowledge.

7. Merging

At this point I have completed the basic elaborations and hence will continue with the four merges described in[17]. Figure 23 shows all the previous elaborations described and the merges of them which will be described in this section. It shows the 6 basic elaborations which have been completed and the four merges which are described in this section. Figure 24 summarizes the interactions of the merges.[9] I will not describe all the possible merges, but rather just the four which are not independent (shaded in figure 24) and so are of greater interest. The four merges could have been done in any order, but will be described as follows:

(1) Factors merged with Devices. The specification which models patients as having multiple values and the specification which models an intermediate device between patients and the monitor will be merged into a single specification. The analyst uses problem domain knowledge to decide whether there should be a device for each patient, one for each factor, or one for each group of factors.

(2) Clock merged with Factors. Here, the analyst must decide which, if any, factors should have their own distinct clock period, rather than reading all factors at the same START_TIME and PERIOD.

(3) Clock merged with Device Failure. The analyst must decide whether the monitoring of device failure should be separate from that of patient values, and if it should be periodic.

(4) Storage merged with Device Failure. The analyst must decide if device failure should be stored.

---

[9] This figure summarizes two figures Feather presented in[17].

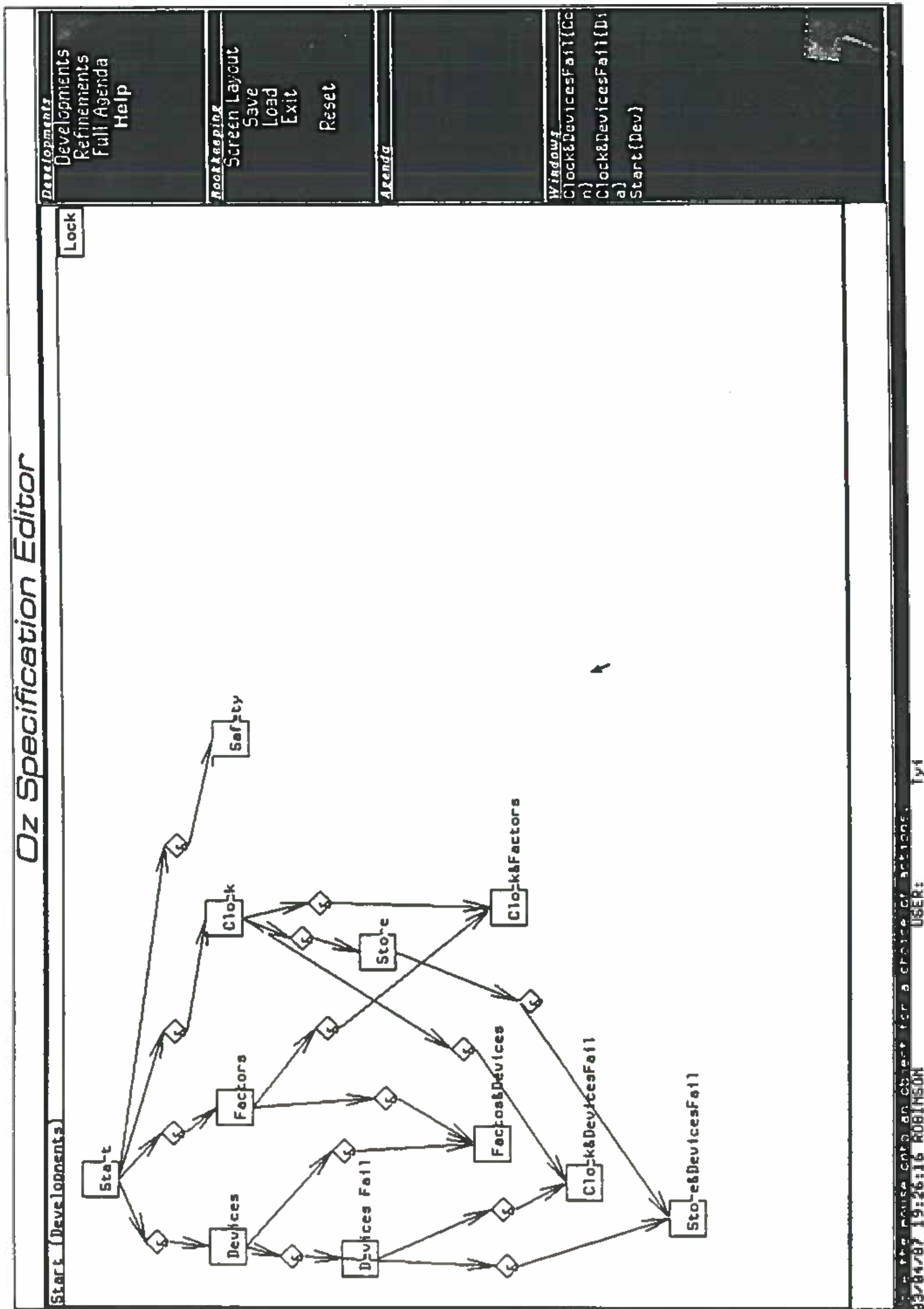


Figure 23. Illustration of the 6 Simple Elaborations Followed by 4 Merges.

	Safety depends on patient	Periodic monitoring	Factor values	Devices fail	Devices	Store read values
Safety depends on patient						
Periodic monitoring	Independent					
Factor values	Independent	Period depends on factor?				
Devices fail	Independent	Periodic monitoring of failure?	Independent			
Devices	Independent	Independent	Independent	Considered sequentially		
Store read values	Independent	Considered sequentially	Device for each factor?	Store record of failure?	Independent	

Figure 24. Summarization of the Interactions Between Specifications.

7.1. Merge Strategy

Before continuing with the example, a more detailed discussion of merge is in order. As mentioned in chapter III, merging of divergent specifications is a difficult task, whose major subproblems are:

- identification of corresponding concepts (a labeling problem),
- identification of conflicting constraints, and
- resolution of conflicting constraints

As mentioned in chapter III, I will assume throughout the example that the analyst determines correspondences explicitly.[10] Identification of conflicting constraints is carried out by the specification model. The more difficult tasks of constraint resolution is done via application of the merge rules of the development model (appendix A) and the rules associated with the specification operators used during the merge (appendix B).[11]

To simplify the merging process, I assume that the result of merging two specifications is the sum of the two, less constraint conflicts. That is, merging does not remove components. Removal of parts of a specification can be done prior to or after a merge; called the *aggregation assumption* in chapter III.

A subproblem of merging is the number of specifications which can or should be merged at once. Also, in which order should they be merged. To look at this problem in more detail, consider the case where only two specifications will be merged at a time. One approach would be to simply take the union of the two specifications and then resolve conflicts. An alternative would be to start with one specification, and apply operators to it (e.g., **AddProcess**) to add unrepresented components of the second specification, resolving conflicts as they occurred. The second approach uses a more incremental analysis of constraint conflicts and their resolution. It allows one to

---

[10]For example, by using a mouse to click on corresponding pairs of processes, arcs, and data objects.

[11]The application the these rules is currently a manual process.

use the operator rules to deal with many constraint conflicts.

However, using the incremental approach to merge very divergent specifications may necessitate the redundant resolution of constraints, whereas one large grained step might be more efficient in such a case. For example, one could add a component of another specification, say a process, to the current specification, which would require a tentative assumption to resolve a conflict. Later, as other components are added, the assumption is invalidated by a new constraint. But, this new constraint resolves the original conflict. Thus, if the union were taken first and then constraint checking applied, the unnecessary assumption and constraint modifications would not have been necessary.

The problem with simply appending the specifications and then resolving conflicts is one of: (1) where to start resolving conflicts and (2) how to specify such resolving rules. I have found it simpler to understand and specify merges if one considers merge as the addition of subcomponents of one specification into another specification.

This same notion of incremental resolution of constraints vs. global resolution turns up when one considers the number of specifications which should be merged at a time. Given N specifications to merge, one could attempt to lump them altogether and then resolve conflicts, or incrementally merge each specification to a growing aggregation, or merge pairs of specifications like a binary tree, or some variant thereof. Merging more specifications at a time gives one more knowledge of what the final specification should be. However, it also introduces the danger of information overload and combinatorial explosion in the number of constraint conflicts. Incremental merge offers the dual benefits of simplicity and maximum opportunity for reuse; reuse of intermediate specifications and the processing it took to create them. On the other hand, using a N-way merge, one can only resue the larger grained specifications embodied in those merges. I speculate that more complex merging schemes may be necessary. For example, merge order and number conditionalized on the type and number

of constraint conflicts that are expected. Appendix B describes our simple merge scheme.

During the discussion of the four merges, I shall take the perspective of an incremental merging methodology. Unlike the previous elaborations, I shall just mention constraints directly involved in a conflict. Other constraints that are added to the specification without conflict will not be commented on, as they have been described previously in the simple elaboration section.

7.2. Merge 1: Patient Factors and Devices

First I will consider the merging of patients factors and devices. The analyst must determine whether the devices are before or after the generation of PFVALUEs. This is because both elaborations separately spliced in a process into the data flow between the patients and the monitor. Oz can not determine whether one process should be before the other, or the two should be in parallel. However, it does know to ask, (see the *Merge Splices* rule of appendix B). It suggests the following Diagram constraint:

P -> PF -> D -> M ?

This constraint is confirmed. It is just a dataflow relation to determine where to place PF and D, and is not part of the Constraint section. *Merge Splices* simply places the process of the first specification before that of the second in order to generate a query.

Next the analyst must decide whether there should be a device for each patient, or a device for each factor, or a device for a subset of factors. This is also decision requiring domain knowledge, and cannot be determined by the system without a more extensive knowledge of patient monitoring. However, the default for splicing a process into a dataflow is to have a single process for each arc in that dataflow. So, in this

Figure 25. Oz Merge of Factors and Devices.

case the default is what the analyst wants, a device for each factor.[12] The following constraints are suggested and confirmed after the ordering of the splices is determined. See figure 25 for the resulting specification.

(1) |D| = |PF|; ?
(2) [DVALUE | **relation**(P,∈ FACTOR,VALUE)]; ?
(3) PFVALUE → [DVALUE = PFVALUE]; ?
(4) DVALUE → [MPVALUE = DVALUE]; ?
(5) (PVALUE, PFVALUE, DVALUE) →→ SAFE; ?
(6) (PVALUE, PFVALUE, DVALUE, MPVALUE) →→ NOTIFY; ?

The first of these constraints says that each factor has a device to read its value. The second is used to distinguish the different arc outputs in the DVALUE relation. The next two (3,4) just maintain the flow of effected values. While, the last two (5,6) are propagated indirect effects.

The constraints were determined by starting with the factors specification and then adding the components of the devices specification which were missing in factors. I used **SIP** to add the devices. The merge rule *Merge Splices* noted that D and PF probably should be part of a sequential dataflow, rather than parallel.

7.3. Merge 2: Periodic Clock and Factors

Next, I examine the interaction between a periodic clock and patient factors. Here, the analyst must resolve the decision as to whether the clock period should depend on a particular factor, rather than just on the patient as it currently does. The system has no knowledge of how START_TIME and PERIOD come into existence (i.e., how they are effected), so it has no basis on which to suggest they be modified. The system queries with:

---

[12] I would rather have the system give the analyst a choice of the possibilities, but most of the time this involves problem domain knowledge, which is not yet part of Oz.

```
(1) [LAST_VALUE_READ | relation(P,∈ FACTOR,VALUE)]; ?
(2) {PVALUE, CLOCK_TIME(t) = START_TIME(P,t)
      (aninteger) * PERIOD(P,?)}
      → [LAST_VALUE_READ(P,f,v) = PVALUE(P,f,v)]; ?
(3) (PFVALUE,{CLOCK_TIME, START_TIME,PERIOD})
      → LAST_VALUE_READ; ?
(4) LAST_VALUE_READ(P,f,v) → [SAFE(true,f,v) | SAFE(false,f,v)]; ?
(5) (PVALUE, PFVALUE, DVALUE,
             {CLOCK_TIME, START_TIME,PERIOD})
      →→ SAFE; ?
(6) (PVALUE, PFVALUE, DVALUE,(PVALUE,
      {CLOCK_TIME,START_TIME, PERIOD},LAST_VALUE_READ)
             →→ NOTIFY;  ?
```

The constraints simply distinguish the LAST_VALUE_READ by which factor was read, and summarize the propagation of those effects. The analyst then adds the following to specify that each factor has a different PERIOD and START_TIME.[13]

```
(7) [PERIOD | relation(P,∈ FACTOR,integer)];
(8) [START_TIME | relation(P,∈ FACTOR,integer)];
(9) {PVALUE, CLOCK_TIME(t) = START_TIME(P,f,t)
      (aninteger) * PERIOD(P,f,?)}
      → [LAST_VALUE_READ(P,f,v) = PVALUE(P,f,v)];
```

The result is shown in figure 26.

7.4. Merge 3: Clock and Device Failure

When combining the clock and devices_fail specifications, one must consider whether the monitoring of failure should be continuous or periodic, and should it be done separately from the read of DVALUEs. The following constraints handle the interaction between the two specifications. They specify periodic monitoring of failure at the same time as monitoring of DVALUEs.

[13]Not part of Feather's elaboration.

Figure 26. Oz Merge of Clock and Factors.

(1) {DVALUE, CLOCK_TIME(t) = START_TIME(P,t) (an integer)
* PERIOD(P,?)}
→ [LAST_VALUE_READ(P,v) = DVALUE(P,v)];
(2) (DVALUE,{CLOCK_TIME,START_TIME,PERIOD})
→→ LAST_VALUE_READ; ?
(3) (PVALUE, DVALUE,{CLOCK_TIME,START_TIME,PERIOD})
→→ SAFE; ?
(4) (PVALUE, DVALUE, {CLOCK_TIME,START_TIME,PERIOD},
LAST_VALUE_READ)
→→ NOTIFY(P); ?

These constraints are determined by the system. Given the clock specification and adding the devices_fail specification to it, one can see that device failure can be added by SIP-related rules, (i.e., *Merge Splices, Propagate Arc Relations*). These rules maintain the flow of PVALUE to LAST_VALUE_READ through D. The failed constraint

FAILED(**true**,VALUE,D) → [DVALUE ≠ PVALUE];

is just carried over from the device_failure specification. It is shown in figure 27.

7.5. Merge 4: Storage and Device Failure

In the final merge, that of the storage specification and devices_fail specification, one must consider whether one should store values of failed devices. The constraints involved in the interaction between these two specifications is along the flow of PVALUE to STORE. If the analyst simply continues to allow this flow unbroken, as suggested by the system, failed DVALUEs will be stored. Here are just those constraints dealing with the flow of PVALUE to STORE:

(1) PVALUE → [DVALUE = PVALUE]; ?
(2) DVALUE → [LAST_READ_VALUE = DVALUE]; ?
(3) {PVALUE,CLOCK_TIME(t) = START_TIME(P,t)
(an **integer**) * PERIOD(P,?)}
→ [LAST_VALUE_READ(P,v) = DVALUE(P,v)]; ?
(4) {LAST_VALUE_READ,CLOCK_TIME(t) = START_TIME(P,t)
(an **integer**) * PERIOD(P,?)}
→ [STORE(P,v) = LAST_VALUE_READ(P,v)]; ?
(5) (PVALUE,DVALUE,{CLOCK_TIME,START_TIME,PERIOD},
LAST_VALUE_READ) →→ STORE; ?

Figure 28 shows the diagram and part of the constraints. All of the constraints are shown in figure 29.

Figure 27. Oz Merge of Clock and Devices_fail.

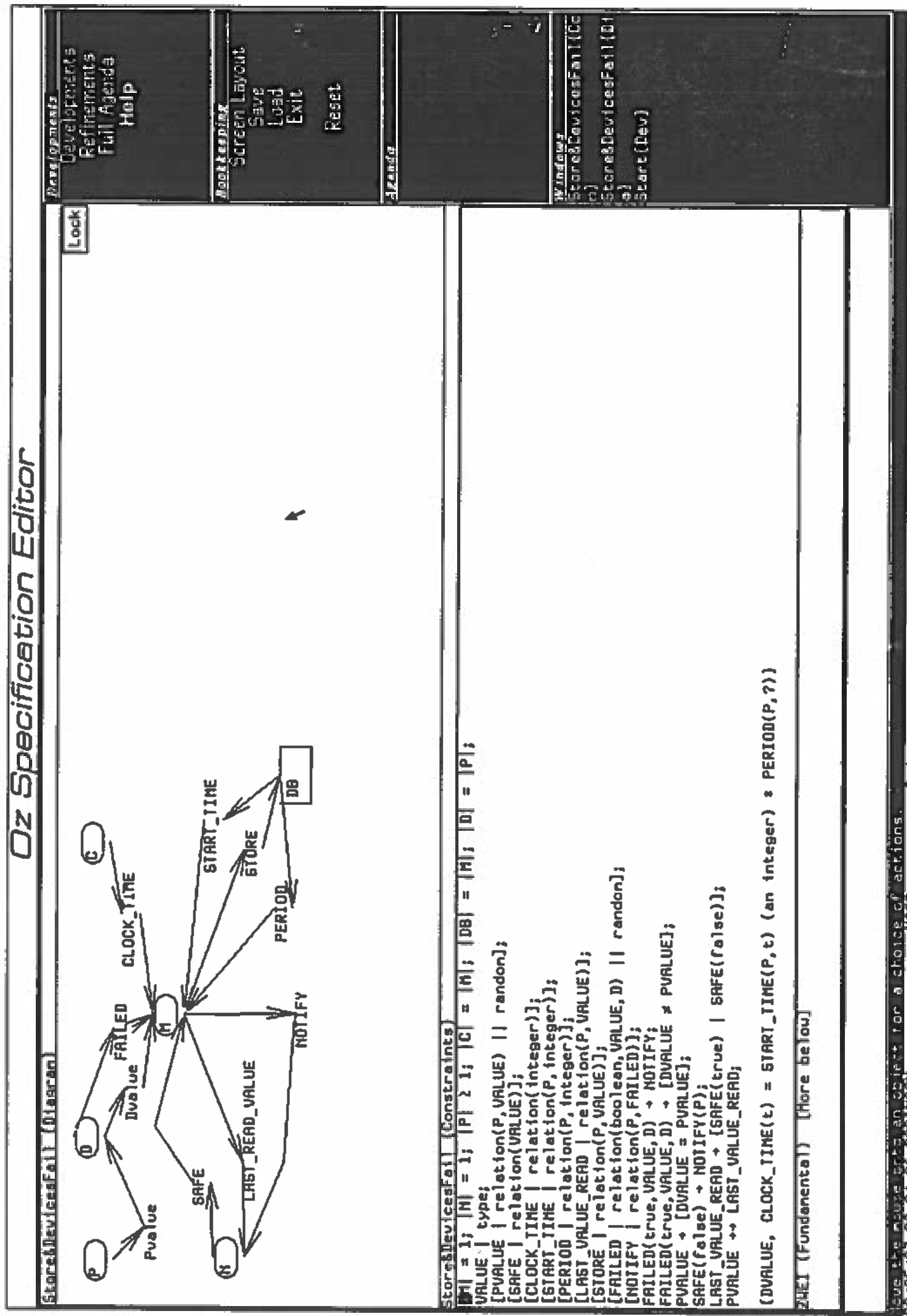


Figure 28. Oz Merge of Storage and Devices_fail, Diagram Section.

```
Oz Specification Editor

Store&DevicesFail (Constraints)
|M| = 1; |N| = 1; |P| ≥ 1; |C| = |M|; |DB| = |M|; |D| = |P|;
VALUE | type;
[PVALUE | relation(P,VALUE) || randon];
[SAFE | relation(VALUE)];
[CLOCK_TIME | relation(integer)];
[START_TIME | relation(P,integer)];
[PERIOD | relation(P,integer)];
[LAST_VALUE_READ | relation(P,VALUE)];
[STORE | relation(P,VALUE)];
[FAILED | relation(boolean,VALUE,D) || randon];
[NOTIFY | relation(P,FAILED)];
FAILED(true,VALUE,D) → NOTIFY;
FAILED(true,VALUE,D) → [DVALUE ≠ PVALUE];
PVALUE → [DVALUE = PVALUE];
SAFE(false) → NOTIFY(P);
LAST_VALUE_READ → [SAFE(true) | SAFE(false)];
PVALUE ↔ LAST_VALUE_READ;

(DVALUE, CLOCK_TIME(t) = START_TIME(P,t) (an integer) * PERIOD(P,?))
        → [LAST_VALUE_READ(P,v) = DVALUE(P,v)];
(LAST_READ_VALUE, CLOCK_TIME(t) = START_TIME(P,t) (an integer) * PERIOD(P,?))
        → [STORE(P,v) = LAST_VALUE_READ(P,v)];

(DVALUE,(CLOCK_TIME,START_TIME,PERIOD)) → LAST_VALUE_READ;
(LAST_VALUE_READ,(CLOCK_TIME,START_TIME,PERIOD)) → STORE;
(PVALUE,DVALUE,(CLOCK_TIME,START_TIME,PERIOD)) ↔ SAFE;
(PVALUE,DVALUE,(CLOCK_TIME,START_TIME,PERIOD),LAST_VALUE_READ) ↔ NOTIFY;
(PVALUE,DVALUE,(CLOCK_TIME,START_TIME,PERIOD)) ↔ STORE;

ZWEI (Fundamental)

Developments
  Developments
  Refinements
  Full Agenda
  Help

Bookkeeping
  Screen Layout
  Save
  Load
  Exit
  Reset

Agenda

Windows
Store&DevicesFail(Co
n)
Store&DevicesFail(Di
a)
Start(Dev)

:Move point, L2:Move to point, M:Mark thing, M2:Save/Kill/Yank, R2:System menu.
03/05/87 00:33:39 ROBINSON        USER:        Tyi
```

Figure 29. Oz Merge of Storage and Devices_fail, Constraint Section.

8. Continuations

There are two dimensions in which the development of the patient monitoring specification could be continued. The first is simply specifying more of the finer details of each component. For example, the safe ranges for patient factors needs to be specified. The other dimension is to bring together the divergent specifications into a single document. At this point there are four specifications (clock&devices_fail, factors&devices, clock&factors, storing&devices_fail) that need to be merged into one (see figure 23). Using the binary merge strategy of appendix B, this would involve three more merges. However, I stop here, as did Feather.

9. Summary

In this chapter I have illustrated the use of an interactive KBPES system and its use in the development of an example previously described by Feather in[17]. I hoped to have shown that much of a KBPES model can be automated and incorporated into an interactive specification environment. Also, due the complexity of this semanticly rich specification design model, such automation is desirable.

CHAPTER V

RELATED WORK

This thesis represents an initial effort to develop a formal model of the process of specification construction, with the intent of building a semiautomated interactive environment which supports specification construction. Unfortunately, there has been little research devoted to studying the *process* of specification creation. However, because specification construction is a kind of formal description evolution, many ideas of program construction are relevant.

Most of the research in the development of formal descriptions can be categorized as either theoretical or cognitive. I consider the theoretical research on description as the study of: (1) the definition of a representation language, (2) the definition of an associated meta-language of operators which can modify the representation language, and (3) the formal properties involved in both (e.g., what each of the languages can describe, optimality of operators in terms of efficiency, etc.). Computer science theory deals with these theoretical aspects in general. In section 1, I describe some research more specific to specification descriptions.

I categorize research that studies the aspects of how people currently describe things as cognitive. Cognitive studies of description attempt to: (1) determine abstractions people use in descriptions, (2) operators they use to evolve a description, and (3) the adequacies of both the abstractions and the associated operators. Such research particular to specification is described in section 2.

Given the above specialized uses of the terms *theoretical* and *cognitive*, I consider language theory to define the space of languages and operators on those languages; this is the *space of development descriptions*. The space of development descriptions defines what can be described and the possible modifications which can be applied to

a given description. Cognitive studies are used to determine which abstractions and operators humans find useful. One may look at currently used operators, or introduce operators found to be powerful theoretically and test for their cognitive usefulness. The PES model includes this second sort of operator; ones which are theoretically powerful, but may not yet be used by analysts. I speculate that studies will reveal that it is also powerful in terms of its cognitive aspects.

Section 3 and 4 highlights related research in specification tools and analysis respectively. Finally, section 5 describes research related to the development of specification construction models.

1. Theoretical Aspects (Languages)

Languages define what can be described, while meta-languages describe how the associated language can be changed. In[5] , Balzer describes a frame based language which is self-described, i.e., structural and functional enhancements are completely derived from the structure of the language. Such languages are useful in that the modification operators are apparent from the structure of the language. Similarly, in the RLL language when an operator is applied, the underlying system alters the semantics of the representation language based on its self-knowledge[29]. Self description in a language is useful in dealing with the modification of the language; it formally describes the space of development descriptions. However, such languages are currently not expressive enough for the needs of complex software specification.

Although not self-described, languages which have well understood semantics with various orthogonally parameterized abstractions may be suitable for complete enumeration of their possible meta-operators and their effects. For example, Goguen[24] describes a hierarchy of operators which can be applied to specifications in Clear[9] with defined effects. While not exhaustive, such a hierarchy is an initial categorization of the sorts of operators that are useful to apply in order to develop a specification. Using operators such as Combine, Instantiate, Enrich, Aggregate, and

Compose (to name a few), Goguen is able to compose specifications. It appears that these operators are sufficiently constrained that they could be applied to similar programming languages (e.g., Poly[35] , Apple[32] , Russell[15]).

A survey of more conventional languages reveals some languages which do facilitate reuse of descriptions (e.g., ADA[14] and Smalltalk[25]), which is necessary when using an evolutionary model of specification. However, I speculate that it would be difficult to develop a useful set of meta-operators which could be used to develop programs in these languages, due to the complexity of their semantics. But, these languages may use attached axioms such as those found in Clear and Meld [30] to support reuse.

I feel specification language design, with regard to evolutionary descriptions, should seek expressive languages in which structural and functional enhancements are completely described. As with many language design goals, solving one tends to defeat the other.

2. Cognitive Aspects

Empirical studies of the operators analysts use to evolve a specification are sparse, and consequently, so is this section. The work of Soloway[45] and Adelson[1] are empirical studies which show the use of plans in program and software design construction. It seems plausible to assume that similar planning operators are used in specification construction.

3. Tools

Three tools which embody the plan composition paradigm are KBEmacs, IDeA, and DRACO. KBEmacs[52] assists programmers in program construction. It does so by providing a library of abstract program plans and then facilitates their application to the programming task at hand. Plans form a common language between the assistant and the programmer. Operators apply plans to the program task and fill in their

abstractions with program constructs.

IDeA[34] and DRACO[40] are similar to KBEmacs, but assist in specification implementation. All three systems take the perspective that operators instantiate plans or their parts. IDeA is similar to KBEmacs in that is assists analysts through an editing interface. In IDeA, one constructs a traditional *design specification* from a high level specification. DRACO, on the other hand, is a set of tools which use domain models in the transformational implementation of specifications.

Like KBEmacs, PegaSys[37] also provides the supporting editor paradigm for program design, but does not provide a library of plans. Instead, it verifies consistency between levels of design diagrams.

Oz takes a more basic perspective of operators: that operators reduce differences between the way the specification is and the way it should be, as determined by the progress of goal and policy compilation. Such operators can involve plan instantiation, combination of plans, and combination of instantiated plans. Each of these system varies with the degree in which it provides: (1) domain modeling, (2) expressive operators, and (3) automation. All concur that the automation of combining plans is a difficult task.

4. Analysis

The four tools mentioned above all provide some level of analysis primarily to integrate plans, and secondarily to inform the analyst of errors in the designed artifact, i.e., program or design specification. Other researchers have concentrated more fully on the analysis issue.

Much of the syntax error detection stems from Petri net analysis[42]. In particular, Tse describes the formalization of data flow diagrams and their analysis based mapping them to Petri nets[51]. Rapid prototyping[3, 7] , paraphrasing[47] , and symbolic execution[10, 55] facilitate the detection of specification sins (cf. chapter II). Finally, analysis of errors due to goal conflicts have received less attention. Fickas[20]

shows good preliminary results.

5. Models

Presently, there is not an adequate model of design that explicitly addresses the compilation of goals and policies of the requirements and also facilitates changes in meaning of the requirements. Changes is the meaning of the requirements, and hence the current state of the specification, must be addressed due to the intertwining of levels of abstraction and their interface with physical devices[48].

Many issues and techniques of Mostow's work on operationalization of task heuristics into procedures (FOO)[38] appear directly transferable to requirements compilation. More recently he has applied such techniques to the rederivation of MYCIN's therapy selection algorithm[39].

This thesis has been primarily concerned with the development of a semanticly rich specification design model. This model is based on the model being developed at ISI by Feather[17] and Goldman[26]. In this approach, one is concerned with the specification language and the operators that apply to it. Goguen[23, 24] and Burstall[8] are similarly motivated, but with more emphasis of the theoretical aspects and less of the cognitive.

Yue[55] and Greenspan[28] are concerned with how reality is represented in specifications. They describe knowledge based models which facilitate the construction of an initial description.

In the domain of knowledge base construction, Freiling[21] describes a methodology by which a knowledge base can be constructed. However, it also addresses initial construction and not subsequent modification.

Dietterich[12] does address modification of design in his abstract model of mechanical design. Figure 30 shows a modified version of his model applied to specification design. It illustrates how analysis of the problem leads to a synthesis of the specification. After which, the specification is compared with the problem

environment to determine its adequacy.

This design process is cyclic and increasingly more detailed. Soloway[45] shows how more experienced programmers use composition of plans to create programs. With regard to program design, Adelson[1] notes that experts expand their mental

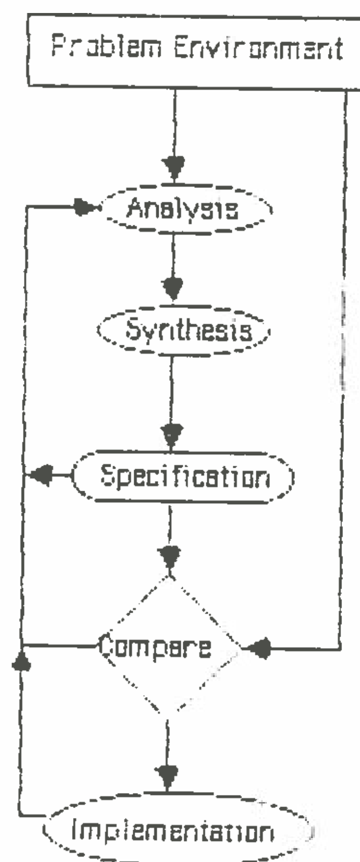


Figure 30. Cyclic Model of Specification Design.

models of the design, one level of abstraction followed by more detailed levels of description. Finally, Goldman[26] notes that slowly evolving the detail of a specification is desirable, if not for design, at least for explanation.

Analysis leads to a new synthesis of the problem description which results in modifications to the specification. Each new specification is then compared against the problem environment which can motivate further analysis and another cycle of design.

The need to modify the meaning of the specification stems from three sources: (1) changing user requirements, (2) unforeseen interactions, and (3) interfacing with physical devices. One would like to reduce the effects of these causes so as to minimize the need to change the specification.

Users may often change their decisions regarding the specification of requirements. However, a good analyst knowledgeable in the problem domain can minimize such changes.

An analyst using the proper development model and analysis tools can curtail the number of unforeseen interactions which are likely to result in specification modification.

Finally, I speculate that analysts use their knowledge of implementations during the analysis phase, but this knowledge and its application can be inadequate. Swartout and Balzer argue that it is impossible to construct a satisfactory design without the feedback of an actual implementation in all but the simplest of descriptions[48]. Thus, the necessity for a continuation of the design process after implementation is depicted.

Dietterich notes that:

> ... little attention has been paid to such strategies as (a) using rough or approximate designs to assess the feasibility of the specification, (b) applying the method of failure-driven patching to develop designs more quickly, and (c) employing an existing design as a guide to designing a new device[12].

It appears that the KBPES model can be used to exploit all three of these design stra-

tegies. In particular, rough approximations are the starting point of Oz specifications. Using analysis techniques such a rapid prototyping, symbolic execution, and formal analysis, one can use early specifications to determine the feasibility of their final implementation. Such analysis or actual implementations indicates some sort of error which then leads to patching. As shown in the example, IV.6.6, patching of specifications is as easy as the initial development. This is not by accident: failure-driven patching is the normal way by which specifications are created.

Finally, note that while the examples of chapter II and III showed that the development space as a lattice containing a single specification, it need not be so. The development space could contain disparate specifications, and transformations could merge these designs to develop new specifications stemming from multiple ancestral roots. While the model allows their transformations, automating such merges will be difficult.

CHAPTER VI

CONCLUSIONS

This chapter describes the current status of this research and the future directions which are being explored.

1. Methodology

This thesis describes my efforts in understanding models of the process in which descriptions of computational systems are created. I have constrained this problem by just considering the construction of specifications. My methodology has been one of surveying current models of specification design. The development of KBPES is the result of formalizing Feather's model[17] and integrating it with knowledge-based design ideas. Feather's model was chosen because it addressed many of the issues of specification design (e.g., multiple analysts, meaning changes, design strategies) and could easily be integrated with a knowledge-based design environment.

The KBPES model, as put forth is chapter IV, is a paradigm for specification construction. As such, it still remains to be justified in its cognitive aspects. The overall theme of developing specifications in an evolutionary style has been justified[26, 48]. But, a robust set of operators which use parallel elaboration for goal compilation must be created and then empirically justified as being cognitively useful. In a completed KBPES, analysts will not apply specification operators such as **AddProcess**, rather, they will apply the more abstract compilation operators.

2. Results

This thesis has presented an initial formalization of a model of the specification design process. This model appears to be useful in that it captures much of the pro-

cess of specification construction. In particular, KBPES explicitly addresses the evolutionary development of specifications. It does so by providing a new development operator, parallel structural elaboration, in a structured environment. Chapter IV shows that significant parts of KBPES can be automated, although the details of an efficient implementation have not been explored.

Besides satisfying many of needs of a specification design model, KBPES facilitates the use of various design strategies. As described in section V.5, KBPES supports such strategies as: (1) evolving simple designs into satisfactory ones, (2) failure-driven patching of designs, and (3) reuse of previous designs in the current one. However, KBPES is an experimental model and many issues remain to be explored.

3. Future Directions

The most pressing problem in KBPES is the formalization of a merge operator. Other issues are: formalization and automation of goal compilation operators, cognitive justification, demands on the specification language, and reuse of disparate specifications.

Given cognitive studies of specification development, I presume that formalization and automation of many of the syntactic operators found will be relatively simple. However, the knowledge associated with such operators (i.e., the rules of appendix A) which are used to maintain the quality of the specification will be much more difficult to express and automate. Such operators are part of the specification model. It will be more difficult to glean from such studies the goal compilation operators which are part of the design model.

The automation of the application of both the operators and their associated rules will likely place new demands on the specification language, thus leading to the development of a specification language tailored to KBPES. The view taken here is that one should start with a simple language in which KBPES can be effectively applied and then expand the expressiveness of both the KBPES operators and the

language based on the experience.

An alluring aspect of KBPES is the prospect of reusing components from different specifications to develop a new specification. In the merge section of chapter IV, it was shown how specifications from the same ancestral root could be merged. More complex specifications will require more complex merging schemes and constraint resolution knowledge. However, I speculate that a merge operator which could be applied to specifications with different ancestral roots would be a natural extension of merge, rather than an order of magnitude leap. This is based on the limited role the development history currently plays in the merging process. It is used to recognize correspondences between specifications and record the series of operators applied which resulted in goal compilation. One can envision slowly extending the merge to capture a larger class of constraint conflicts, using it to merge disparate specifications. However, first a satisfactory merge should be constructed for commonly rooted specifications.

APPENDIX A

OPERATOR SUMMARY

This appendix summerizes the operators used in the example of chapter IV. The operators are only informally described. In the rules, I use the following notational conventions that:

(1) Processes start with P, e.g., P1, P2,

(2) Arcs start with A, e.g., A1, A2. Their direction is determined by an <, or >, e.g., P1 A1> P2, A1 provides a flow from P1 into P2.

As explained in section IV.4., rules are organized into a hierarchy and attached to operators. After a operator and its rules are executed, suboperators and their rules are also executed, if any exist. These are also noted for each operator.

1. AddArc

AddArc adds an arc between two processes and determines the arc name. This is a primitive operator; no subordinate operators are executed.

Rule 1: *Corresponding Processes*

IF P1 A1> P3, and
P2 A2> is appended to form P1 A1> P2 A2> P3, or
P1 A2> P2 A1> P3,
THEN suggest |P2| = |P1|;

Rule 2: *Distinguish Output*

IF A3 is an input arc added to process P2 from P1, and
P1 A1> P2, and
A1 → A2, and
P2 A2> P3, and
it is confirmed that A3 → A2, and
[A2 | **relation**(...)];
THEN suggest [A2 | **relation**(...,A3)];

Rule 3: *Propagate Arc Relations*

IF P1 A1> P2, and
A2 is appended to form P1 A1> P2 A2> P3, and
A1 → A2,
THEN suggest existing relations containing A1, but with A2
substituted for A1.

Rule 4: *Determine Effect of Input*

IF A1 is an input arc added to process P, and
An is the set of output arcs of P,
it has not been determined how A1 effects An,
THEN ask the analyst which value(s) (if any) in An
are effected by A1.

Rule 5: *Determine Effector of Output*

IF A1 is an output arc added to process P, and
An is the set of input arcs of P,
it has not been determined how An effects A1,
THEN ask the analyst which value(s) (if any) in An
effect A1.

2. AddProcess

AddProcess adds a process to the specification and determines any in or outgoing arcs. **AddArc** rules are executed as secondary rules.

3. AddData

AddData adds a data entity to the specification and determines any in or outgoing arcs. **AddArc** rules are executed as secondary rules.

4. SIP

SIP splices a process into a dataflow. It determines the name of the process and the outgoing arc. After application, **SIP**'s own rules are executed, followed by those of **AddProcess** and **AddArc**. The work of *Propagate Spliced Relations* can be done by *Propagate Arc Relations*, but is inlcuded for added clarity.

Rule 6: *Propagate Spliced Relations*

IF P1 A1> P3, and
P2 A2> is appended to form P1 A1> P2 A2> P3, or
P1 A2> P2 A1> P3,
THEN suggest existing relations containing A1, but with A2
substituted for A1, or vice versa in the second case.

5. Unpack

Unpack is a specialize sort of **SIP** which splices a process into a dataflow for purposes of unpackaging a composite value. The spliced process serves as an intermediary between the original primitive value form and its composite representation. **Unpack** determines the process and arc names, then its rules determine the set of composite values.

Rule 7: *Determine Composite Set*

IF P1 A1> P2 A2> P3, and
P2 and A2 are the splice,
THEN then suggest [A2 | **relation**(...,∈ Set)];
A2 → [∪A2∈ = A1];
Set | **type** = {...};
determine the value for Set = {...}
determine the name for Set

6. Constrain

Constrain simply allows the user to alter the *Constraints* section. As a result, all constraints must be checked for inconsistencies after this operation. The rule presented here goes beyond simple constraint conflict recognition. It is explained in section IV.6.4.

Rule 8: *Use Processed Data*:

IF A2 → A3, and
A1 → A2, and
A2' is added (A2' ⊆ A2), and
A1 → A2' is also added, and
THEN suggest A2' → A3, and
A2 →→ A3

APPENDIX B

MERGE RULE SUMMARY

This appendix summerizes the rules used to merge the example of chapter IV. The rules are only informally described. In these rules I use the following notational conventions that:

(1) Processes start with P, e.g., P1, P2,

(2) Arcs start with A, e.g., A1, A2. Their direction is determined by an <, or >, e.g., P1 A1> P2, A1 provides a flow from P1 into P2,

(3) Specifications start with S, e.g., S1, S2.

The merging scheme I followed was generally: start with the more complex of two specifications and add components of the simpler to it, until it embodies both specifications. More concretely:

(1) Merge only two specifications at a time.

(2) Alter the more complex specification S1, by adding components of the other specification S2, until S1 has all the processes, arcs, and data sources of S2.

(3) During each addition in the above step, resolve constraint conflicts using the rules of Appendix A and those found here.

1. Merge Order

This rule determines the complexity of a specification, which is used to determine which specification will be copied and then modified. It does a simple complexity analysis based on the number of processes, arcs, and data sources. If neither specification is more complex than the other according to the analysis, then one is arbitrarily picked as more complex than the other.

Rule 9: *Complexity of Specifications*

IF the complexity of S1 is greater than that of S2
THEN start with S1 and modify it using S2, else vice versa.

2. Flow Rules

There rules show how conflicts along flows of effect were resolved in the example.

Rule 10: *Chained Input*

IF S1 has P1 A1> P2 A2> P3, and
S2 has P4 A4> P5, and
A1 → A2, and
A1 and A4 correspond,
P1 and P4 correspond, and
P3 and P5 correspond
THEN the merge is S1 (and add any of S2's constraints)

Rule 11: *Merge Splices*

IF S1 and S2 correspond except for the sets of processes Px in S1 and Py in S2, and
Px forms an effect chain between P1 and P2 in S1, and
Py forms an effect chain between P3 and P4 in S2, and
P1 corresponds to P3, and
P2 corresponds to P4,
THEN it is likely that one chain should precede the other in the merged specification, so the analyst should be asked.

APPENDIX C

GIST SPECIFICATION

This appendix presents the four simple GIST specifications in the order of parallel development found in[17]. Each development is only briefly commented on. For more details of these specifications and their merge, see[17].

1. Initial Gist Specification

The initial specification, figure 31, is the same of the Oz starting specification. The three major components are the patients, monitor, and nurse station.

2. Introducing Devices

Devices are introduced by the addition of a device agent and linking device values to patient values. See figure 32.

```
{ value | type
,
patient | agent ||
 { PVALUE | relation(patient,value) || random }
,
monitor | agent ||
 { NOTICE_UNSAFE | activity(patient) ||
   if start not SAFE(PVALUE(patient,?))
   then NOTIFY(patient)
 }
,
nurse's_station | agent ||
 { SAFE | relation(value) , NOTIFY | procedure(patient) }
}
```

Figure 31. Initial GIST Specification.

```
device | agent ||
{ DVALUE | relation(device,value) || random

'DEVICE_VALUE_EQUALS_LINKED_PATIENT_VALUE | invariant
  || all d|device || DVALUE(d,?) = PVALUE(LINK(d,?),?)

'LINK | relation(device,patient)
 }


'monitor | agent ||
 { NOTICE_UNSAFE | activity(patient) ||
   if start not SAFE(DVALUE(LINK(?,patient),?))
   then NOTIFY(patient)
 }
```

Figure 32. Modifications to Introduce Devices.

```
device | agent ||
{ DVALUE | relation(device,value) || random

' FAILED | relation(device) || random

'DEVICE_VALUE_EQUALS_LINKED_PATIENT_VALUE | invariant
  || all d|device || not FAILED(d) implies
                 DVALUE(d,?) = PVALUE(LINK(d,?),?)

'LINK | relation(device,patient)
 }
```

Figure 33. Modification to Introduce Device Failure.

3. Device Failure

Device failure is introduced as an implied nonequivalence of PVALUE and DVALUE, figure 33. Unlike the Oz development, Feather does this elaboration with three changes. First, the nonequivalence between PVALUE and DVALUE. Then, the next two figures adjust the Monitor to make use of the failed relation. Namely, figure 34 modifies the monitor to NOTIFY in case of failure. Next, figure 35 adjusts the NOTIFY relation to distinguish between patient safety notification and device failure

```
{ NOTICE_UNSAFE | activity(patient) ||
  if not FAILED(LINK(?,patient))
              and start not SAFE(DVALUE(LINK(?,patient),?))
      or start FAILED(LINK,(?,patient))
  then NOTIFY(patient)
}
```

Figure 34. Modification of NOTICE_UNSAFE to Respond to Device Failure.

notification.

4. Patient Factors

Patient factors, figure 36, are introduced by adding the factor relation and modifying the monitor to check each factor separately for safety.

5. Periodic Monitoring

Time is modeled by a increasing integer valued clock. The Monitor then, only updates LAST_VALUE_READ when the correct period comes to pass for each patient value. See figure 37.

```
{ NOTICE_UNSAFE | activity(patient) ||
  if not FAILED(LINK(?,patient))
              and start not SAFE(DVALUE(LINK(?,patient),?))
      or start FAILED(LINK,(?,patient))
  then NOTIFY(patient, FAILED(LINK(?,patient)))
}
...
NOTIFY | procedure(patient,boolean)
```

Figure 35. Modifications to Adjust NOTIFY Invocations.

```
factor | type = { 'pulse, 'temperature, 'blood_pressure, 'skin_resistance }
,
composite | type
,
CFV | relation(composite,factor,value)
...
PVALUE | relation(patient,composite) || random
...
NOTICE_UNSAFE | activity(patient) ||
 if start not all f | factor || SAFE(CFV(PVALUE(patient,?),f,?))
 then ...
...
SAFE | relation(value,factor)
```

Figure 36. Modifications to Introduce Composite of Factor Values.

```
clock | agent ||
{ CLOCK_TIME | relation(integer)

, TICK_TOCK | activity ||
   choose { CLOCK_TIME(?) := CLOCK_TIME(?) + 1, null }
}

,
monitor | agent ||
 { LATEST_VALUE_READ | relation(patient,value)

 , READ_PERIODICALLY | activity(patient) ||
    if CLOCK_TIME(?) = START_READING_TIME(patient,?) +
                         (an integer)*PERIOD(patient,?)
    then LATEST_VALUE_READ(patient,?) := PVALUE(patient,?)

 , NOTICE_UNSAFE | activity(patient) ||
   if start not SAFE(LATEST_VALUE_READ(patient,?))
   then NOTIFY(patient)
 }

,
nurse's_station | agent ||
 { SAFE | relation(value) , NOTIFY | procedure(patient) }

,
 START_READING_TIME | relation(patient,integer)

,
 PERIOD | relation(patient,integer)
 }
```

Figure 37. Modifications to Change the Monitor to Read Periodically.

6. Saftey Depends on Patient

In figure 38, the SAFE relation in the nurse's station and monitor are modified to take the added patient parameter. This allows safeness to vary beteween patients.

7. Store Values

Storing patient values is the last of the simple elaboration and was added to show maintenance via transformations. The periodic monitoring specification was modified so values are stored at the same time they are read, see figure 39.

```
monitor | agent ||
{ NOTICE_UNSAFE | activity(patient) ||
  if start not SAFE(PVALUE(patient,?), patient)
  then NOTIFY(patient)
}
,
nurse's_station | agent ||
{ SAFE | relation(value,patient) ... }
```

Figure 38. Modifications to Elaborate Safe to Depend on Patient.

```
monitor | agent ||
{ LATEST_VALUE_READ | relation(patient,value)
,
  READ_PERIODICALLY | activity(patient) ||
    if CLOCK_TIME(?) = START_READING_TIME(patient,?) +
                          (an integer)*PERIOD(patient,?)
    then LATEST_VALUE_READ(patient,?) := PVALUE(patient,?);
               STORE(patient,LATEST_VALUE_READ(patient,?))
,
  NOTICE_UNSAFE | activity(patient) || ...
}
,
nurse's_station | agent ||
{ ..., STORE | procedure(patient,value)}
```

Figure 39. Modifications to Store Read Values.

BIBLIOGRAPHY

1. Adelson, B. and Soloway, E., "The Role of Domain Experience in Software Design," *Transactions of Software Engineering* **SE-11** (November 1985) 1351-1360.

2. Aho, A., Sethi, R., and Ullman, J., *Compilers: Principles, Techniques, and Tools,* Addision-Wesley (1986).

3. Balzer, R., Goldman, N., and Wile, D., "Operational Specification as the Basis for Rapid Prototyping," *Sigsoft Software Engineering Notes* **7** (December 1982) 3-16.

4. Balzer, R., "A 15 year perspective on automatic programming," *Transactions on Software Engineering* **11** (November 1985) 1257-1267.

5. Balzer, R., "Automated Enhancement of Knowledge Representations," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* **2** (August 1985) 203-207.

6. Barstow, D. R., "Domain-Specific Automatic Programming," *Transactions on Software Engineering* **11** (November 1985) 1321-1337.

7. Boehm, B., Gray, T., and Seewaldt, T., "Prototyping Versus Specifying: A Multiproject Experiment," *Transations on Sofware Engineering* **SE-10** (May 1984) 290-303.

8. Burstall, R.M. and Goguen, J.A., "Putting Theories Together to Make Specifications," *Proc. 5th Int. Joint Conf. on AI,* (1977) 1045-1058.

9. Burstall, R. M. and Goguen, J. A., "An Informal Introduction to Specifications Using CLEAR," in: Eds. J. Strothers Moore, *The Correctness Problem in Computer Science,* Academic Press (1981) 185-213.

10. Cohen, D., "Symbolic execution of the Gist specification language," *Proc. Eight Int. Joint Conf. Artif. Intell.,* (1983) 17-20.

11. DeKleer, J., "An Assumption-Based TMS," *Artificial Intelligence* **28** (March 1986) 127-162.

12. Dietterich, T. and Ulman, D., "Artificial Intelligence Approaches to Design," *Artificial Intelligence in the Northwest,* (October 22-24 1985) 8/3.

13. Dietterich, T. G. (Ed.), *Proceedings of the Workshop on Knowledge Compilation,* Oregon State University, Otter Crest (September 24-26, 1986).

14. DoD, "Reference Manual for the ADA Programming Langauge," in: Eds. Ellis Horwitz, *Programming Languages: A Grand Tour,* Computer Science Press (1976) 417-752.

15. Donahue, J. and Demers, A., "Data Types Are Values," *Trans. on Prog. Lang. and Systems* **7** (July 1985) 426-445.

16. Feather, M.S., "A Survey and Classification of some Program Transformation Approaches and Techniques," *TC2 Working Conf. on Program Specification and Transformation,* (April 1986)

17. Feather, M. S. , "Constructing specifications by combining parallel elaborations," Transactions on Software Engineering, IEEE (1987).

18. Fickas, S., "Automating the Transformational Development of Software," *Transations on Software Engineering* **SE-11** (November 1985) 1268-1277.

19. Fickas, S., "A Knowledge-Based Approach to Specification Acquisition and Construction," CIS-TR-85-13, University of Oregon (November 1985).

20. Fickas, S., "Automating the Analysis Process: An Example," *Proc. 4th IWSSD,* (April 1987) 58-67.

21. Freiling, M., Alexander, J., Messick, S., Rehfuss, S., and Shulman, S., "Starting a Knowledge Engineering Project: A Step-by-Step Approach," *The AI Magazine* **6** (Fall 1985) 150-164.

22. Gane, C. and Sarson, T., *Structured Systems Analysis: Tools and Techniques,* Printice-Hall (1979).

23. Goguen, J.A. and Burstall, R.M., "CAT, A System for the Structured Elaboration of Correct Programs from Structured Specifications," CSL-118, SRI (October 1980).

24. Goguen, J.A., "Reusing and Interconnecting Software Components," *Computer* **19** (February 1986) 16-28.

25. Goldberg, Adele and Robson, David, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley (1983).

26. Goldman, N., "Three Dimensions of Design Development," ISI/RS-83-2, ISI (July 1983).

27. Green, C., Luckham, D., Balzer, R., Cheatham, T., and Rich, C., "Report on a Knowledge-Based Software Assistant," RADC-TR-83-195, Rome Air Development (1983).

28. Greenspan, S.J., "Requirements Modeling: A Knowledge Representation Approach to Software Requirments Definition," CSRG-155, University of Toronto (March 1984).

29. Greiner, R. and Lenat, D.B., "A Representation Language Language," *Proceedings*, (1980) 165-169.

30. Kaiser, G., "Composing Software Systems from Reusable Building Blocks," *Twentieth Hawaii International Conf. on Systems Sciences*, (January 1987)

31. Kemmerer, R., "Testing Formal Specifications to Detect Design Errors," *Transations on Software Engineering* **SE-11** (January 1985) 32-43.

32. Kieburtz, Richard B. and Nordstrom, Bengt, "The Design of Apple--A Language for Modular Programs," *Computer Lang.* **10** (1985) 1-22.

33. London, Philip E. and Feather, Martin S., "Implementing Specification Freedoms," *Science of Computer Programming* **2** (1982) 91-131.

34. Lubars, M.D. and Harandi, M.T., "Intelligent Support for Software Specification and Design," *Expert*, (Winter 1986) 33-41.

35. Matthews, David C.J., "Poly Manual," *SIGPLAN Notices* **20** (September 1985) 52-76.

36. Meyer, B., "On Formalism in Specifications," *Software* **2** (January 1986) 6-26.

37. Moriconi, M. and Hare, D. F., "PegaSys: A System for Graphical Explanation of Program Designs," *Proc. ACM SIGPLAN 85 Symposium on Languages Issues in Programming Environments* **20** (June 25-28, 1985) 148-160.

38. Mostow, D.J., "Mechanical transformation of task heuristics into operational procedures," CS-81-113, Computer Sci. Dept, CMU (1981).

39. Mostow, J. and Voigt, K., *Explicit Incorporation and Integration of Multiple Design Goals in a Transformational Derivation of the MYCIN Therapy Selection Algorithm*, IJCAI87 (January 1987).

40. Neighbors, J. M., "The Draco Approach to Constructing Software from Reusable Components," *Transactions on Software Engineering* **SE-10** (September 1984) 564-574.

41. Page-Jones, M., *The Practical Guide to Structured Systems Design,* Yourdon Press (1980).

42. Peterson, J. L., "Petri Nets," *Computing Surveys* **9** (September 1977) 223-252.

43. Sheil, B. A., "Power Tools for Programmers," in: Eds. E. Sandewall, *Interactive Programming Environments,* McGraw-Hill (1984) 19-30.

44. Smith, D. R., Kotik, B., and Westfold, S. J., "Research on Knowledge-Based Software Environments at Kestrel Institute," *Transations on Software Engineering* **SE-11** (November 1985) 1278-1295.

45. Soloway, E. and Ehrlich, K., "Empirical Studies of Programming Knowledge," *Transactions on Software Engineering* **SE-10** (September 1984) 595-609.

46. Stevens, W.P., Myers, G.J., and Constantine, L.L., "Stuctured Design," *Systems Journal* **13** (1974) 115-139.

47. Swartout, W., "Gist English generator," in: *Proceesing of AAAI-82,* AAAI (August 1982) 404-409.

48. Swartout, W. and Balzer, R., "On the Inevitable Intertwining of Specification and Implementation," *CACM* **25** (1982) 438-440.

49. Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," in: Eds. E. Sandewall, *Interactive Programming Environments,* McGraw-Hill (1984) 97-116.

50. Teitelman, W., "A Display-Oriented Programmer's Assistant," in: Eds. E. Sandewall, *Interactive Programming Environments,* McGraw-Hill (1984) 240-287.

51. Tse, T.H. and Pong, L., "Towards a Formal Foundation for DeMarco Data Flow Diagrams," TR-A6-86, University of Hong Kong (June 1986).

52. Waters, R. C., "The Programmer's Apprentice: A Session with KBEmacs," *Trans. on Software Engineering* **SE-11** (November 1985) 1296-1320.

53. Weinreb, Daniel and Moon, David, *Lisp Machine Manual,* Symbolics, Inc., Chatsworth, CA (1981).

54. Wirth, N., "Program Development by Stepwise Refinement," *CACM* **2** (April, 1971) 221-227.

55. Yue, Kaizhi, "Constructing and Analyzing Specifications of Real World Systems," STAN-CS-86-1090, Stanford University (September 1985).

56. Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems," *Transations of Software Engineering* SE-8 (1982) 211-236.