

A CRITIC FOR SOFTWARE  
SPECIFICATIONS

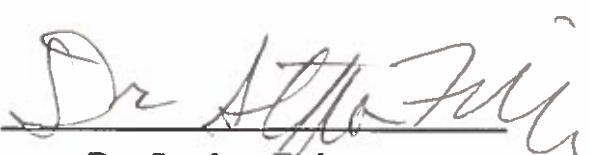
by

P. NAGARAJAN

A THESIS

Presented to 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 1988

Approved:   
Dr. Stephen Fickas

An Abstract of the Thesis of  
P. Nagarajan for the degree of Master of Science  
in the Department of Computer and Information Science  
to be taken June 1988

Title: A CRITIC FOR SOFTWARE SPECIFICATIONS

Approved:   
Dr. Stephen Fickas

This thesis proposes a knowledge based critic for software specifications. The critic is part of an environment where the users can build their specifications using a Petri net based language, ObjNPN. The system tries to make the development of the specifications an interactive process. The user can invoke the critic as soon as a primitive specification has been developed. The critic performs a domain-dependent analysis of the specifications. A case-based approach is adopted using examples from the domain. The analysis involves critiquing the specifications for the presence/absence of components related to the *policies* of the domain. The system also provides mechanisms for performing symbolic execution of the specifications.

VITA

NAME OF AUTHOR: P. Nagarajan

PLACE OF BIRTH: Thanjavur, Tamil Nadu, India

DATE OF BIRTH: September 13, 1964

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, Oregon, USA  
Indian Institute of Technology, Kanpur, India

DEGREE AWARDED:

Master of Science, 1988, University of Oregon  
Bachelor of Technology, 1986, Indian Institute of Technology

AREAS OF SPECIAL INTEREST:

Artificial Intelligence  
Software Engineering

PERSONAL EXPERIENCE:

Research Assistant, Department of Computer and Information Science,  
University of Oregon, Eugene, 1987-88

Teaching Assistant, Department of Computer and Information Science,  
University of Oregon, Eugene, 1986

## ACKNOWLEDGMENTS

I wish to express my sincere gratitude to Professor Stephen Fickas, my advisor. His encouragement and guidance throughout my study has been invaluable. Special thanks are also due to William Robinson for his support, friendship and insightful comments. I also express my gratitude to Ms. Swapna Trivadi for her editorial comments. I thank Kevin Looney for his skill and great effort; this document could not have met the requirements of the graduate school without him. Finally, the faculty, staff and colleagues in the CIS department, have all contributed in making my stay here at the University an educative and pleasant experience.

DEDICATION

To my parents

## TABLE OF CONTENTS

Chapter	Page
1. INTRODUCTION . . . . .	1
Goals of Analysis . . . . .	1
A History of our Approach . . . . .	4
Critic-1 . . . . .	4
Critic-2 . . . . .	5
Critic-3 . . . . .	5
Critic-4 . . . . .	7
Critic-5 . . . . .	9
How the Critic Works . . . . .	9
Related Research . . . . .	10
Scope of the Thesis . . . . .	12
2. OBJECT-ORIENTED NUMERICAL PETRI NETS . . . . .	14
Introduction . . . . .	14
Petri Nets . . . . .	14
Numerical Petri Nets . . . . .	15
ObjNPN . . . . .	19
Example of an ObjNPN . . . . .	21
Features of ObjNPN . . . . .	26
Limitations of ObjNPNS and Extensions . . . . .	28
Summary . . . . .	29
3. THE ANALYST . . . . .	30
Introduction . . . . .	30
The Domain Model . . . . .	30
Resources . . . . .	31
Resource Operations . . . . .	32
Constraints . . . . .	32
Policies . . . . .	32
Domain Behaviors . . . . .	36
Overview of the System Operation . . . . .	38

An Example . . . . .	39
4. CONCLUSION . . . . .	45
A Discussion . . . . .	45
Reaching Compromises . . . . .	47
More Automation ? . . . . .	48
Contribution of this Thesis . . . . .	49
APPENDIX	
A. DESCRIBING INITIAL MARKING AND AGENDA . . . . .	51
BIBLIOGRAPHY . . . . .	55



## LIST OF FIGURES

Figure	Page
1. Critic-1 and Critic-2 . . . . .	6
2. Critic-3 and Critic-4 . . . . .	8
3. A Petri Net . . . . .	16
4. A Marked Petri Net . . . . .	17
5. A Marked Petri Net showing Conflicts . . . . .	18
6. An ObjNPN in Construction . . . . .	23
7. A Complete ObjNPN . . . . .	24
8. The Policy Graph for Library Systems . . . . .	33
9. A Specification model of a Library System . . . . .	40
10. The Critic in use I. . . . .	41
11. The Critic in use II. . . . .	43

## CHAPTER 1

### INTRODUCTION

All modern software development models incorporate a specification process as the first step. Specification is essentially a way of expressing what we expect of a system. Since natural languages have ambiguities, informal specifications often fail to express correctly the client's intentions. This has necessitated the formation of a group in the software engineering circle which believes in the importance of formal specifications [19]. This research epitomizes this philosophy.

The problem of analysis of formal specifications as a part of the specification construction process is the primary focus of this thesis. The problem is treated as one in Artificial Intelligence. Barstow [2] has discussed the need for the application of A.I. techniques to Software Engineering. In keeping with Marr's tenet that a *result* in Artificial Intelligence consists of the isolation of a particular information processing problem, and the statement of a *method* for solving it [18], this work provides a *method* for analysis of software specifications. This work is a component of the knowledge-based system KATE proposed by Fickas [8] that will assist in the automation of software development. The approach adopted here was introduced by him in the 4th IWSSD [9] and is also described in [10].

#### Goals of Analysis

Before performing the analysis of specifications we should identify the goals we need to achieve. There are three major goals of analysis. Goal I is to make sure that the specification is free of syntactic errors. Given a *syntactically sound* specification, the next question that a specification analyst faces is whether the given specification is *programmable*. By *programmable* we mean that the specification does

not involve any problem for which a solution is not known. There are algorithms and resources available for each of the tasks mentioned in the specification to be performed. However, it is beyond the scope of this thesis to verify the availability of algorithms and resources for a given specification. Boehm [3] discusses some of the economic issues in software engineering and provides the COCOMO model to help people understand the cost consequences of software development. It is essential to discuss these and other related issues during the specification analysis phase before the actual development of the software. This would be Goal II.

Validating the intentions or the functionality of the specification forms the third goal. It is important to validate that the specification performs the intended function. In other words, we have to check that the intentions of the client are correctly and completely represented in the specifications. Stretching it to the extreme, we cannot call a syntactically sound and programmable specification for a book-keeping system, one which meets the client's requirements of word-processing, acceptable. It has to be validated that the specification addresses the given problem and is functionally correct. There are two levels of problems involved here:

- (a) To detect if the specification describes the correct problem.
- (b) To detect if the described problem is the one the designer(client) would like a solution to.

It might seem that the first problem is an integral part of the second one. This may not be the case. Even if the specification addresses a client's problem, that problem may not be *correct*. For example, consider the following specification for a resource management system :

We have a pool of resources and a group of users who might request for a resource. If a request comes in for a resource (say, A) and A is in the pool then lend it.

This specification is for a resource manager, but does the client want such a

system ? There are no constraints on the borrower(user) in the specification. The pool of resources may empty itself very soon and then the resource management system will fail.

It is considerably easier for human analysts to detect the problems of the above described nature. But why is that so? Undoubtedly because the human analysts have prior knowledge about resource management systems. They know that in general, in such systems, it is desirable that every user be given equal opportunity to borrow. Since the number of resources is limited, it is often desired that every resource borrowed be returned in a fixed time. Thus validation of intent, which forms the Goal III, needs knowledge of the domain. This does not mean that the Goals I and II can be accomplished without domain knowledge. That cannot be said unless *methods* to solve those problems are established.

Achieving Goal III involves checking if the specification defines the desired behaviors of the system. That is, as described in the above paragraph, the analyst should see if a set of *policies* applicable to the domain is met by the specification. *Policies* are statements which govern the domain behaviors. For example, "serve all patrons uniformly" was a policy in the resource management system. The policies also can be thought of as the goals<sup>1</sup> of the client. Thus the analyst needs to have prior knowledge of possible policies applicable to the domain. Given the requirement of the client, that is the policies desired by the client, the analyst looks for the presence of these policies in the specification. In the experiments conducted here [11], it was apparent that human analysts not only detect the absence/presence of policies but they also recognize conflicting policies. Hence, the relation between the policies should also be known to the analyst.

This thesis introduces an approach of analysis of specification using domain knowledge and policy implementations for fulfillment of Goal III. To start with, a new specification language is introduced. This language is an extension of Petri nets

---

<sup>1</sup> Henceforth we use *policy* and *goal* synonymously.

[20, 21] and is essentially a modeling language. This language is called ObjNPN (Object-oriented Numerical Petri Net). It seems to be a good candidate for a specification language. We shall discuss in the next chapter how this language upholds the principles laid down by Balzer and Goldman in [1] to a large extent. This will be followed by a discussion of the design of our analyst in detail through an example which will lead us to the conclusion of this work.

### A History of our Approach

Our interest is to develop a specification critic. This specification critic is a part of a larger project whose goal is to provide assistance to a software analyst in producing a formal specification. This project, called Kate [8], focuses on the following three components:

- (a) A model of the domain of interest. This includes the common objects, operations, and the constraints of the domain, as well as information on how they meet the types of goals or policies one encounters in the domain.
- (b) A specification construction component that controls the design of the client's emerging specification.
- (c) A critic that attempts to poke holes in the client's problem description.

This research focusses on the building of the third component, the specification critic, using the first component, the model of the domain. This attempt at designing the critic has taken us through several iterations. We present these in the form of five major versions in the following sections.

#### Critic-1

As an initial effort, an analysis tool called SKATE was constructed [9]. This basically consisted of two components; namely, a specification language and a rule-based critic. The critic, shown in figure 1a, was composed of a *model*, an *example*

and correspondence links between the components in *model* and *example*. The *model* was a representation of a particular application domain while the *example* was a representation of a particular problem specification in that application domain. The specification language NIKL [7] was used to represent both the *model* and the *example*.

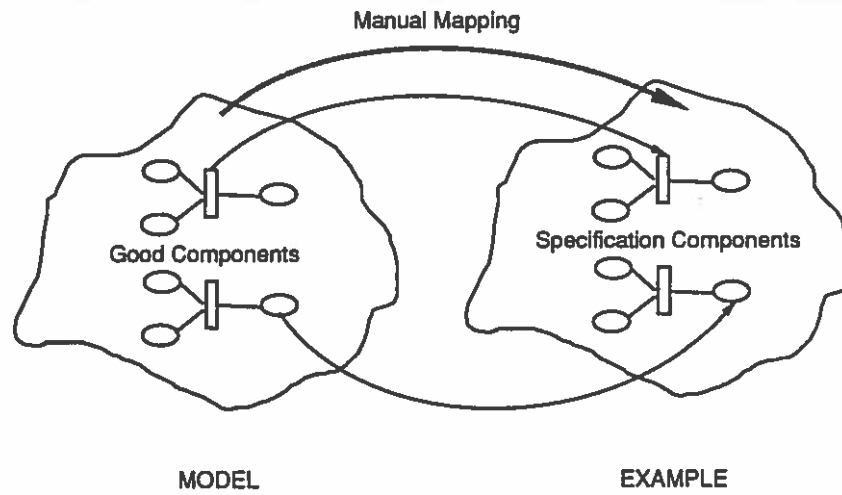
To use the critic several manual steps had to be carried out. First a representation of a domain had to be incorporated in the *model*. The second step was to translate the specification to be critiqued into the *example* format. The third step required the creation of correspondence links between the components of the *model* and the *example*. After these three steps were carried out, a rule-based critic, implemented in the ORBS language [7], was employed to find components of *model* that were not linked to components in *example*.

### Critic-2

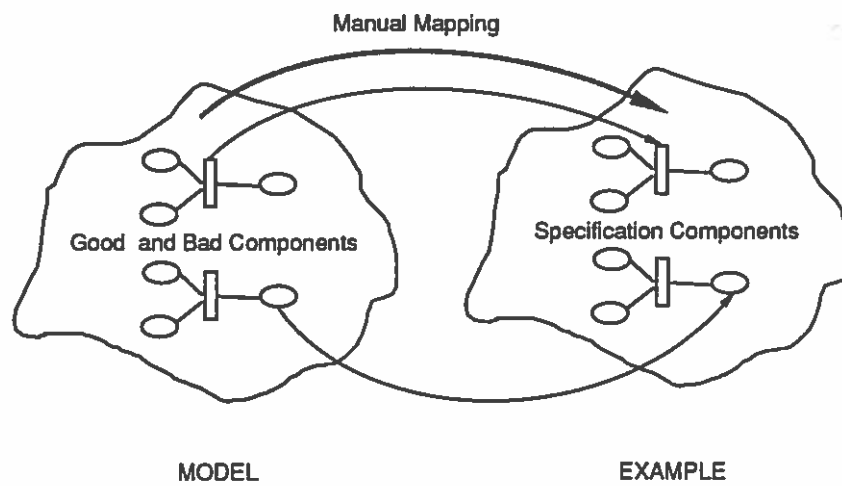
The major deficiency found in critic-1 was that it could only detect missing components usually found in real physical systems. It was important to find out if the specification included certain components which could be problematic. This feature was added to the design of the critic to form the second critic, Critic-2 (shown in figure 1b). This addition would produce warnings for components whose presence might cause adverse effects in real systems.

### Critic-3

In general, experiments conducted with expert analysts [11] revealed that, specifications are not *bad* inherently. They can only be judged on the basis of the intent of the clients. More specifically, the criteria which should form the basis of our judgement are the goals (or policies, as referred to in this work) and the resources available with the client. A two component critic, such as Critic-2, is too rigid and



1a: Critic-1



1b: Critic-2

Figure 1: Critic-1 and Critic-2

may not be of good use in real situations. Hence it is worthwhile to introduce one component which represents the policies of the domain and another which finds the relationship between the domain components and the policies.

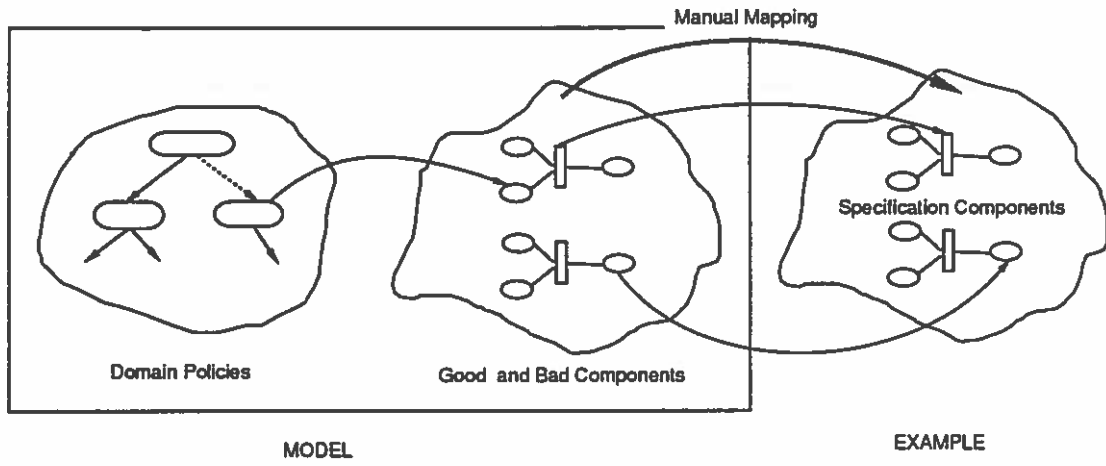
Ideally, the policies need to be distinguished as *important*, *unimportant* and *unknown*. As a result, we can find out from the clients, their prioritization of the policies for their usage. We also have the policies related to other policies by two types of links : *positive* - when one policy component supports the other policy; *negative* - when one policy component thwarts the other policy. The critic with this policy component forms our Critic-3, shown in figure 2a.

#### Critic-4

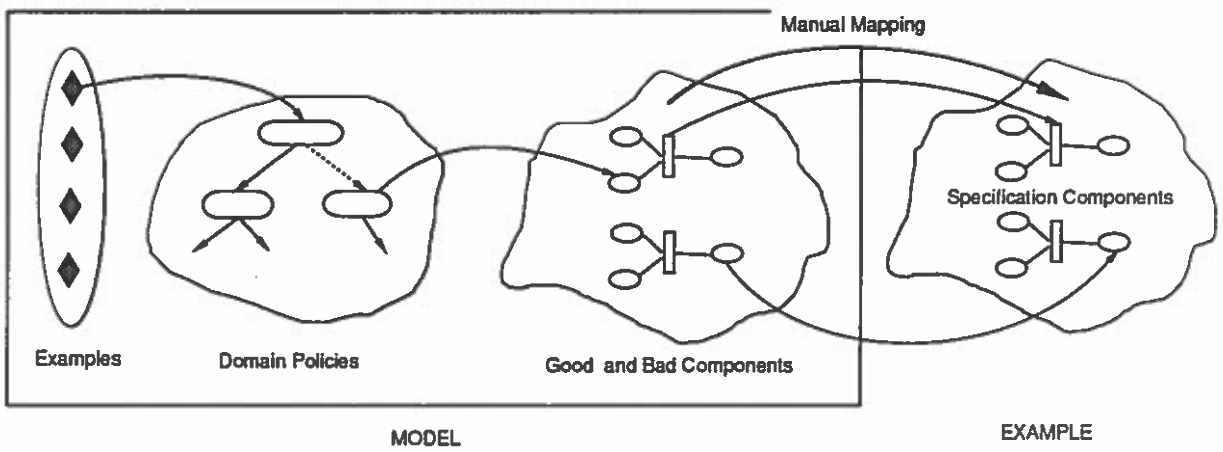
We realized that mere policies are not powerful enough to represent the goals of the clients, the resource constraints, and the human behaviors associated in the form of the users of the system. Basically some script-like representations of the domain behaviors are needed to associate them with each policy or *vice versa*. This made us consider works in legal reasoning systems such as [25, 24] where Rissland uses case-based reasoning to compare and contrast among legal cases. We employ similar *cases* or usage-scenarios to capture the domain behaviors. As we will see later, the cases act as justifiers for the existence of the policies in the system. Also, they open up avenues for analyzing the closed specifications as the cases encode the environmental interaction with the system. The use of cases significantly increases the scope of our critic. It is possible to explain through the cases the missing components in a specification in domain terminology, as opposed to describing dataflow type of syntactic errors which are often too generalized to be comprehended by the client. The critic can also be extended to include the capability of suggesting components for the specification during its construction. Further, it might provide comments on the effects of removing certain components from the specification.

We now have come to a version of the critic, Critic-4, which incorporates policies





2a: Critic-3



2b: Critic-4

Figure 2: Critic-3 and Critic-4

and cases to analyze a specification (figure 2b). The next version basically integrates these into an environment which allows us to build and analyze specifications.

### Critic-5

As observed by Barstow [2], building specifications is a problem solving process and should be treated as one. We have built a system which allows users to incrementally develop a specification, to call the critic when ready, to edit the specification if required, to reinvoke the critic and so on. The critic on its part makes use of the users if required in designating the component links between the *model* and *example*.

### How the Critic Works

As mentioned earlier, this research adopts a case-based approach in building the specification critic. The critic uses a knowledgebase of examples which display the behavior of the specifications. These examples are associated with specific domain policies. The policies include information about the facts that have relevance to their claims. That is, the prerequisites for the policies to be met by specifications, and the contribution of that policy to the weakness and the strength of specifications are part of the knowledgebase. Initially, the critic searches for the components of specifications which serve as the prerequisites for the policies. These components are expressed in the form of factual predicates (for example, *borrower-is-registered-user*). The critic looks at the user's specification from various perspectives. The perspective is defined by the policy the user is interested in. If the policy of interest in our domain of resource management is, *allow only registered users to borrow*, then the critic looks at the knowledge associated with this policy. The prerequisites for this policy are then looked for in the user's specification. To decrease the complexity of this part of the policy detection process, we restrict the users to define

their specifications only using some predefined factual predicates. In this manner the critic can accomplish a matching of these predicates in the specification, to detect the policy requirements.

If the specification fails to meet the requirements for a policy then the critic adopts a negative attitude. Assuming that the specification does not implement the policy of interest, the example case associated with the policy is mapped on the user's specification model. This mapping again involves matching of components. The critic seeks help from the user through queries to help in the matching process. The example case is then presented to the user in an interesting way. It was observed in our experiments [11] that human analysts have an ability to generate examples to explain problems. They use their deep understanding of the domain and their practical experiences with specific physical systems to produce both hypothetical and real examples. We tried to use a similar approach by presenting hypothetical scenarios. We attach to the cases in the knowledgebase, a description to generate examples. This can be looked upon as a transformation of the static cases to dynamic ones. By presenting a concrete example related to the domain, we make it easier for the user to decide on the presence or requirement of the policy in question. The executable nature of our specification language makes it easier to map the parameters of the matched case to the specification and demonstrate the execution. If the users feel that the specification does not behave in a desirable fashion, they can go back and modify the specification. Our system thus tries to make the specification development an interactive and iterative process as suggested in [28].

### Related Research

There has not been much work on domain related analysis of software specifications. Cohen [5] and Swartout [27] have built a symbolic evaluator for software specifications which explains the behavior of the specification defined in GIST. This

approach allows the user to test parts of the specification by designating particular Gist actions to the system. The system uses some symbolic inputs to run these actions and produces a trace of the execution for the user to see. Another part of the system tries to highlight the important aspects of the execution by looking at the trace produced. However, this system does not make use of any domain knowledge. Evidently their explanation revolves around the features of the language.

As far as the use of the domain knowledge is concerned the Requirements' Apprentice project [23] comes closer to ours. But it is basically directed towards building a system which will assist an analyst in the creation and modification of software requirements. It requires an extensive library of knowledge of the particular domain of the requirement to be constructed. The Apprentice describes the requirements in various levels of detail depending on the audience. It allows informality in the specifications and works with a library of *cliches* from the domain in which the requirement is to be constructed. It is claimed that the Apprentice improves both productivity and the quality of the requirement produced because of its following capabilities: support for cliches, propagation of information and contradiction detection. Most of the work on this project has been directed towards knowledge representation and reasoning (CAKE system) [6].

Work has been done in artificial intelligence which involves matching of concepts [4]. Detection of policy implementations in a specification also involves matching of concepts. The problem in Legal reasoning systems [12] is comparable to ours. The objective in such systems is given a case, to compare and contrast it with other cases in order to be able to make a claim for or against the case. Rissland [25] adopts a case-based approach for legal reasoning using the history of related cases. Rissland's HYPO system treats the case in hand as a hypothetical situation. It tries to make it stronger for a specified party by modifying it along specific dimensions. These modifications are guided by comparison with cases stored in the knowledge base.

### Scope of the Thesis

This thesis is more a proposal of an approach towards the analysis problem than an implementation of the approach. A very small system has been implemented as foundation work in the SIMKIT [26] environment on Symbolics. This thesis also presents a petri-net based language as a language for describing specifications.

Using ObjNPN as the specification language, a system has been developed to define the specifications interactively on SIMKIT's graphical interface. KEE Pictures [15] are used to represent the various elements of the net in the form of icons. This ObjNPN system has been extended with a system which shows the proposed approach of analyzing the specifications using knowledge about the policies and example cases applicable in the domain. The domain used here is Library Systems. This system does not solve the problem completely. As mentioned earlier, most of the difficulties involved are in matching cases. In this system, the knowledge base is currently very small with just a few example cases. The system does not try to find the *closest* match from the example cases. If an example fails to match then the system tries to use the next available example in the knowledge base. If examples are exhausted then the system just admits its failure. No attempt is made to analyze the failure. We try to finesse the matching problem in this system by seeking the user's help (like in finding corresponding elements between two cases).

In the absence of extensive domain dependent analysis of specifications, the major intent of this work is to introduce and examine some important issues for further research. Even though it may be *difficult* to develop a full-fledged system of this nature for analyzing specifications, we feel that this is a promising approach towards the analysis of specifications.

We introduce the ObjNPN language in Chapter 2 and follow it up with some examples. Chapter 3 provides a description of the implementation of the analyst. Chapter 4 concludes the thesis with a general discussion followed by suggested

extensions.

## CHAPTER 2

### OBJECT-ORIENTED NUMERICAL PETRI NETS

#### Introduction

In this chapter we look at Numerical Petri Nets (NPN) and the modifications made to derive Object-oriented NPNs (ObjNPN). We start by giving a short introduction to Petri nets. It is suggested that the reader first obtain a formal definition of Petri nets from [20, 21]. This chapter provides an example of an NPN model after introducing the notation used. A complete definition of NPNs is beyond the scope of this thesis, but can be found in [29, 30].

#### Petri Nets

Petri nets are a model for systems where information flow plays an important role. Here we present a very brief and informal introduction to Petri nets for readers who are not familiar with it.

Figure 3 shows a simple Petri net. This graphical representation models the static properties of a system. The Petri net graph is a bipartite graph with two types of nodes; namely, *places* and *transitions*. Places are represented as circles and transitions as bars in the figure. These nodes are connected by two types of directed arcs namely *input arcs* and *output arcs*. The arcs which goes from a place to a transition are the input arcs while the arcs from a transition to a place are output arcs. Nodes of the same type are not connected. Besides these elements for modeling the information flow, there are markers called *tokens* which reside in the places of a net. The distribution of the tokens in a Petri net constitutes the *marking*

of the net.

The dynamic properties represented by the net are a result of its execution. Execution of a Petri net is basically a change in its marking guided by the following *firing* rules of the transitions:

- (a) Tokens are moved by the firing of transitions.
- (b) A transition must be *enabled* in order to fire. A transition is enabled when all of its input places have a token in them.
- (c) A transition fires by removing one token from each of its input places and generating a new token at each of its output places.

Figure 4 shows a marked Petri net. In this net the transitions T1 and T2 are both enabled. In such situations we have a choice as to which transition to fire next. This feature is very useful in modeling nondeterminisms in physical systems. Figure 5 shows another marking for the net. Here firing either of transitions T3 and T4 disables the other. In such cases the transitions T3 and T4 are said to be in *conflict*.

The above, informal description of Petri nets, should give the reader enough background knowledge to understand the higher forms of Petri nets discussed hereafter.

### Numerical Petri Nets

Numerical Petri Nets are Place/Transition (P/T) systems [22] with some extensions as noted below.

- (a) Tokens are generalized to tuples of variables as in Predicate Transition Nets (PrT) [13].
- (b) Each input arc in an NPN has two inscriptions associated with it.



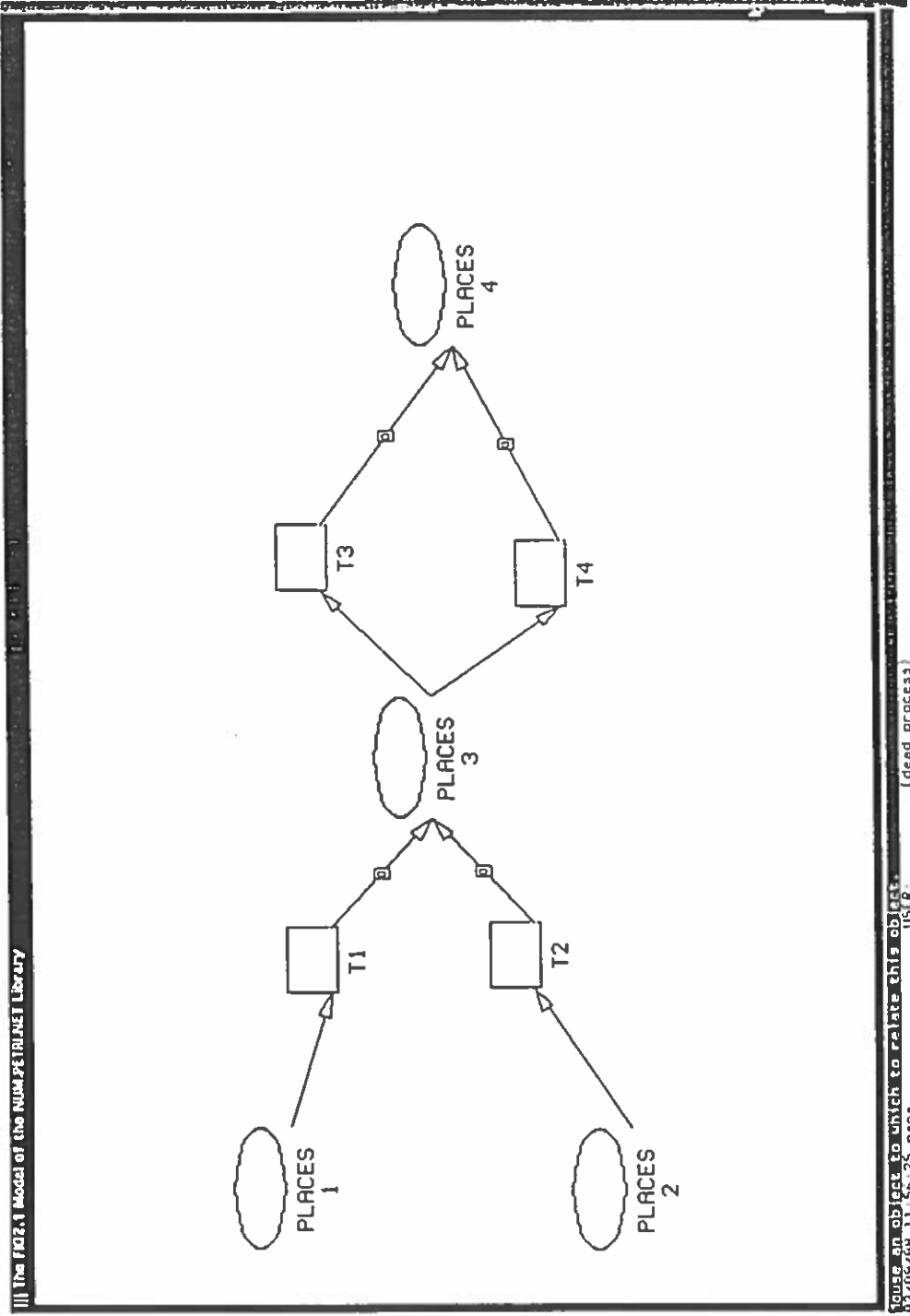


Figure 3: A Petri Net

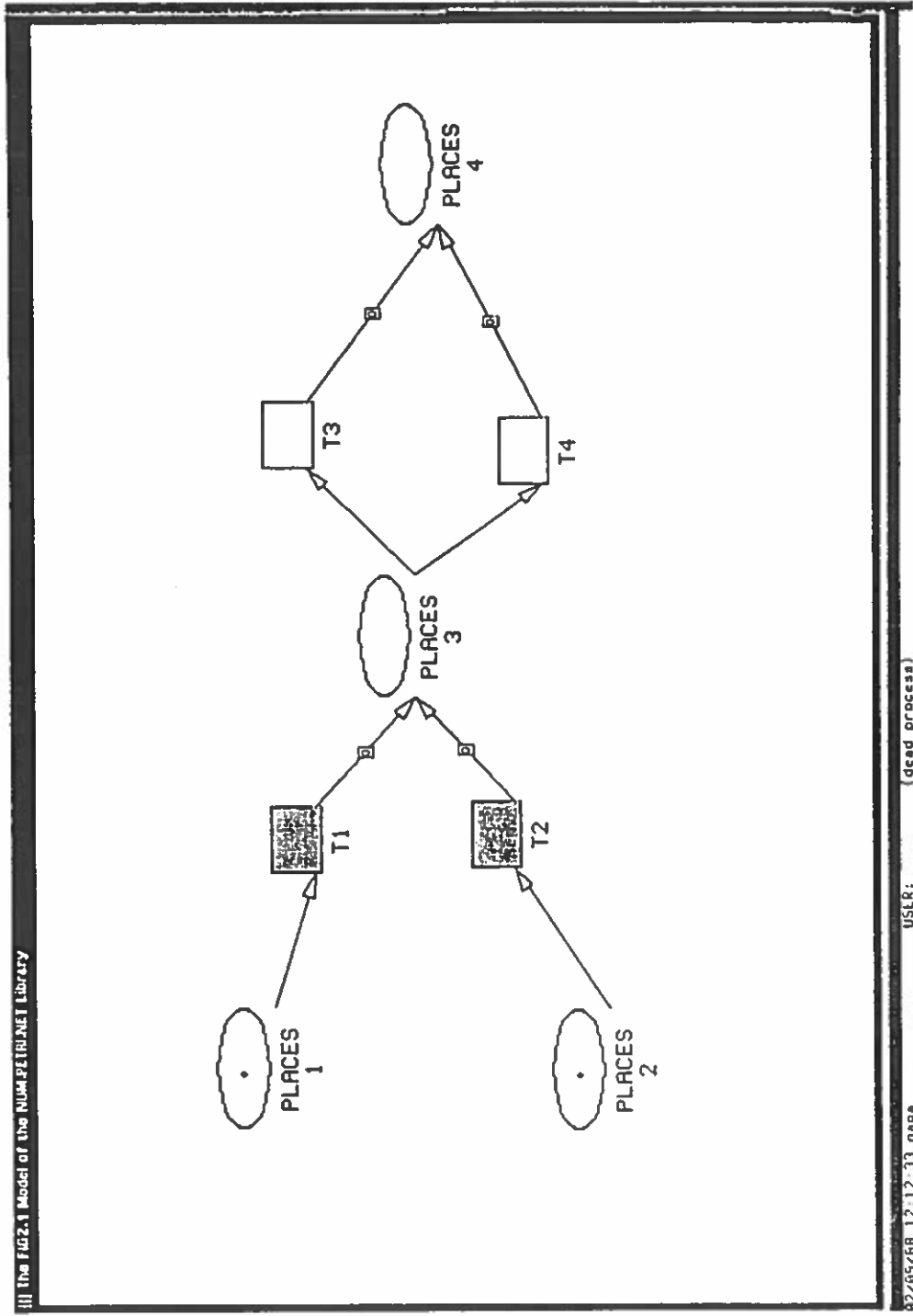


Figure 4: A Marked Petri Net

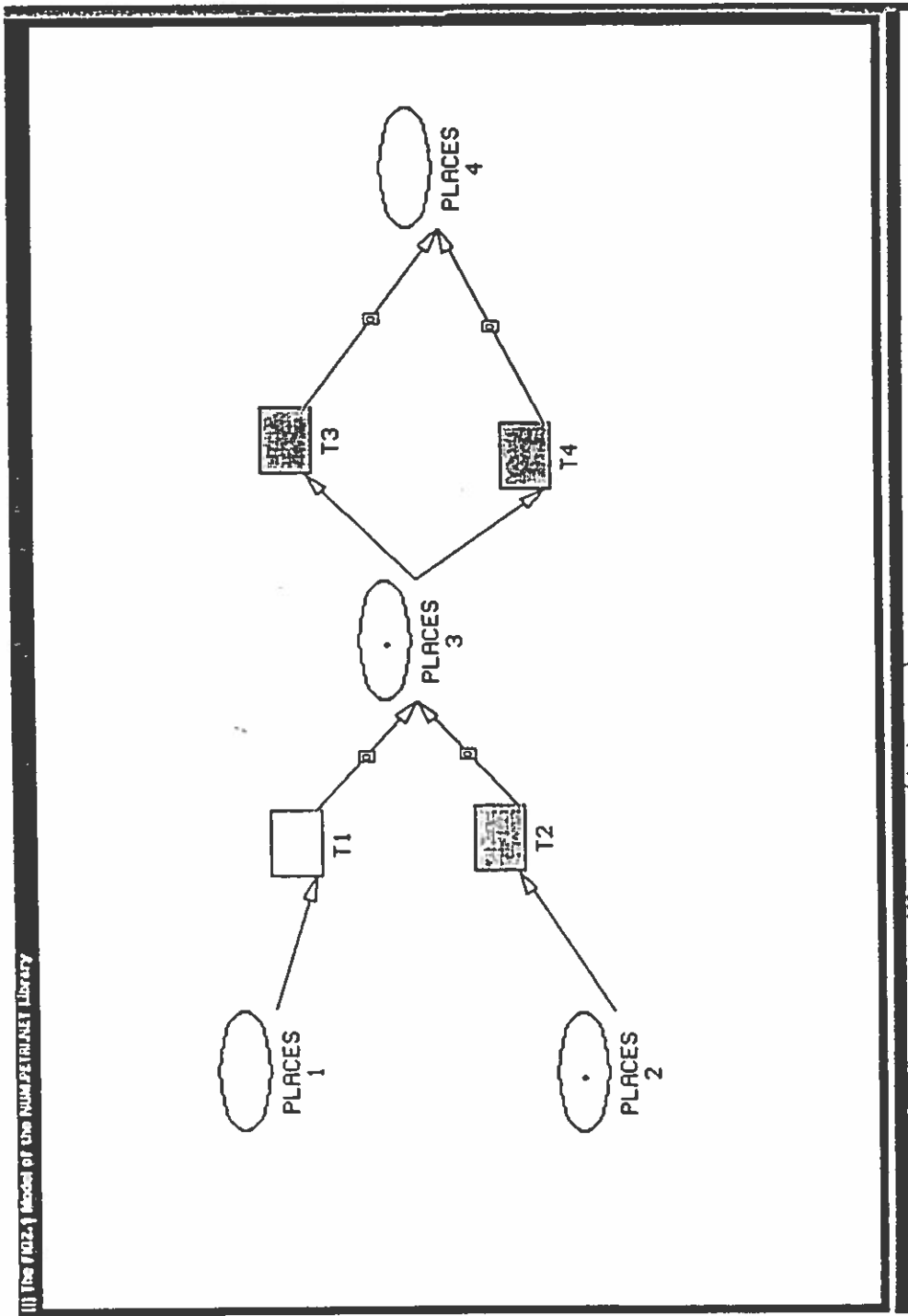


Figure 5: A Marked Petri Net showing Conflicts

- (i) An Input Condition (IC), defines a condition which may be satisfied by a collection of tokens in the associated input place.
  - (ii) The Destroy Tokens (DT), defines the bag<sup>1</sup> of tokens to be removed from the associated input place, when the transition fires.
- (c) Each output arc in an NPN has an inscription, Create Tokens(CT), associated with it. This defines the bag of tokens to be created/deposited into the output place, when the transition fires.
- (d) Each transition in an NPN has two inscriptions associated with it:
- (i) Transition Conditions (TC), define the conditions on the tokens resident in the transition's input places. These conditions have to be met for the transition to fire.
  - (ii) Transition Actions(TA), define the actions to be performed on the tokens when the transition fires.
- (e) An Initial Marking (M0) which defines the initial allocation of tokens to each place in the net.
- (f) Instantiable variables (Ivars), which are wild cards in the ICs. They can be used to specify DTs, CTs, TCs and TAs.
- (g) Place capacities are defined in a manner to enable restriction on the total number of tokens present at a place and also on tokens of specific types.

### ObjNPN

Object-oriented Numerical Petri Nets (ObjNPNs) derives its name from the representation of the elements of the net. The places and transitions are in the

---

<sup>1</sup> A bag is an extended form of sets. We allow repetition of elements in a bag. But for our purpose a bag may be interpreted as a set here.

form of objects. This allows them to have additional information in the form of their slot values.

In ObjNPNs the tokens are not mere tuples but objects. These objects have slots whose values correspond to the fields of the tuples in the NPNs. The object representation allows us to have a hierarchy of token classes and subclasses. The subclasses inherit the slots of their parents. This permits us to address the tokens at a place using their generic class name, if required.

In addition to the above mentioned differences from an NPN, in this implementation the ICs have been restricted to be just a bag of tokens as in Pr-Transition nets. It is interpreted as the condition that the bag specified is contained in the input place. Other forms of ICs can be added on this system by providing the operators defined for the NPNs. Similarly, the TCs and TAs are merely lisp expressions defined using some predefined operators. Again, these could be extended (or rather restricted) to the forms defined for the NPNs.

In petri nets the transition firing are thought of as instantaneous events and the places are viewed as processes. There is no temporal aspect attached to either of these. Since in most physical systems events are rarely momentary transitions, we need mechanisms to specify time associated with these events. ObjNPNs provides one such mechanism. With each transition we associate an attribute called *duration*, which represents the time taken by the event represented by the transition to complete. The provision of this concept of time will prove very useful for specifying scheduling systems and systems where certain events occur in concurrence with some others. It is useful to determine mutual interdependency of the transitions and the possibility of parallel firings of transitions. However, in this implementation, we do not provide for concurrent firing of transitions.

The notion of an *agenda* has been introduced in this system for the execution of ObjNPNs. An agenda basically describes the desired behavior of the net during execution. Currently we use an agenda only to describe the sequence of transitions

to fire and changes in marking in between the firings. We see a lot of promise for further research in the use of an agenda for constraining system behaviors. This might be an avenue to explore for introducing references to future markings of the nets. An agenda can also incorporate constraints which tie in more than one transition's input places. In a way an agenda can be a rule system, which can have rules of the form:

If place P1 has two tokens and transition T2 is not enabled then try transitions T1 and T3 in that order; else try transition T3.

Place P1 may not have been an input place for any of the transitions T1, T2 or T3. These can be looked upon as some form of global constraints too. The proper use of agenda might eradicate the deficiencies of this language for the specification process discussed in a later section.

#### Example of an ObjNPN

Let us consider a specification of a resource management system to represent in the form of an ObjNPN. Consider the following informal specification in English for the system :

Client A wants to have a system which manages resources of types say R1, R2 and R3. There are patrons (P) who would make requests for one resource at a time. Only patrons are allowed to borrow the resources. Any resource borrowed by a patron should be returned before he/she can borrow another one.

To express the above given specification in ObjNPN, we first identify the transitions, places and tokens in it. Here there are only two activities involved namely, borrowing and returning. Hence, we shall have two transitions, one for each activity. The resources can be assumed to be kept as a pool, and this pool can be represented as a place. Another place can be used to show the resources which have been

borrowed and are no longer in the pool. Since the specification does not mention anything about how to maintain records of the borrowings, a representation can be chosen for record maintenance. Let there be a place where a record about each patron is kept. This record will carry information regarding the resource borrowed by the patron. After taking care of the places and transitions, the tokens required have to be decided. In this case, the resources can be treated as tokens. Since there are three types of resources one could either have three different types of tokens or have a resource token with a name tag attached to it to identify its type. In this example, the former approach defines one resource class and then defines three subclasses, one for each resource type. This enables us to encode any generic properties of the resources in the parent class since these will be inherited by the subclasses. The properties particular to each resource type can be represented in the corresponding subclass. The requests for a resource made by the patrons are also to be shown. These requests can form another class of tokens. The patron-record can also be a class of tokens. This completes the identification of the various elements of the net to form the required specification. But the input and output arcs for the transitions and the inscriptions which go along with these arcs are yet to be identified. Finally, the transition conditions to represent the constraints on the two operations will have to be defined.

Figure 6 shows the places and transitions instantiated while forming the specification model.

The relation between these elements are defined in the form of the input and output arcs. For a borrowed transaction to take place, the following are the minimum requirements. There must be a request for a resource; the resource must be present in the pool; the person making the request must be a patron. This decides the three input arcs for the BORROW transition as shown in figure 7.

After a borrow transaction the resource borrowed has to move out of the pool into the new place, called BORROWED RESOURCES. Hence an output arc is

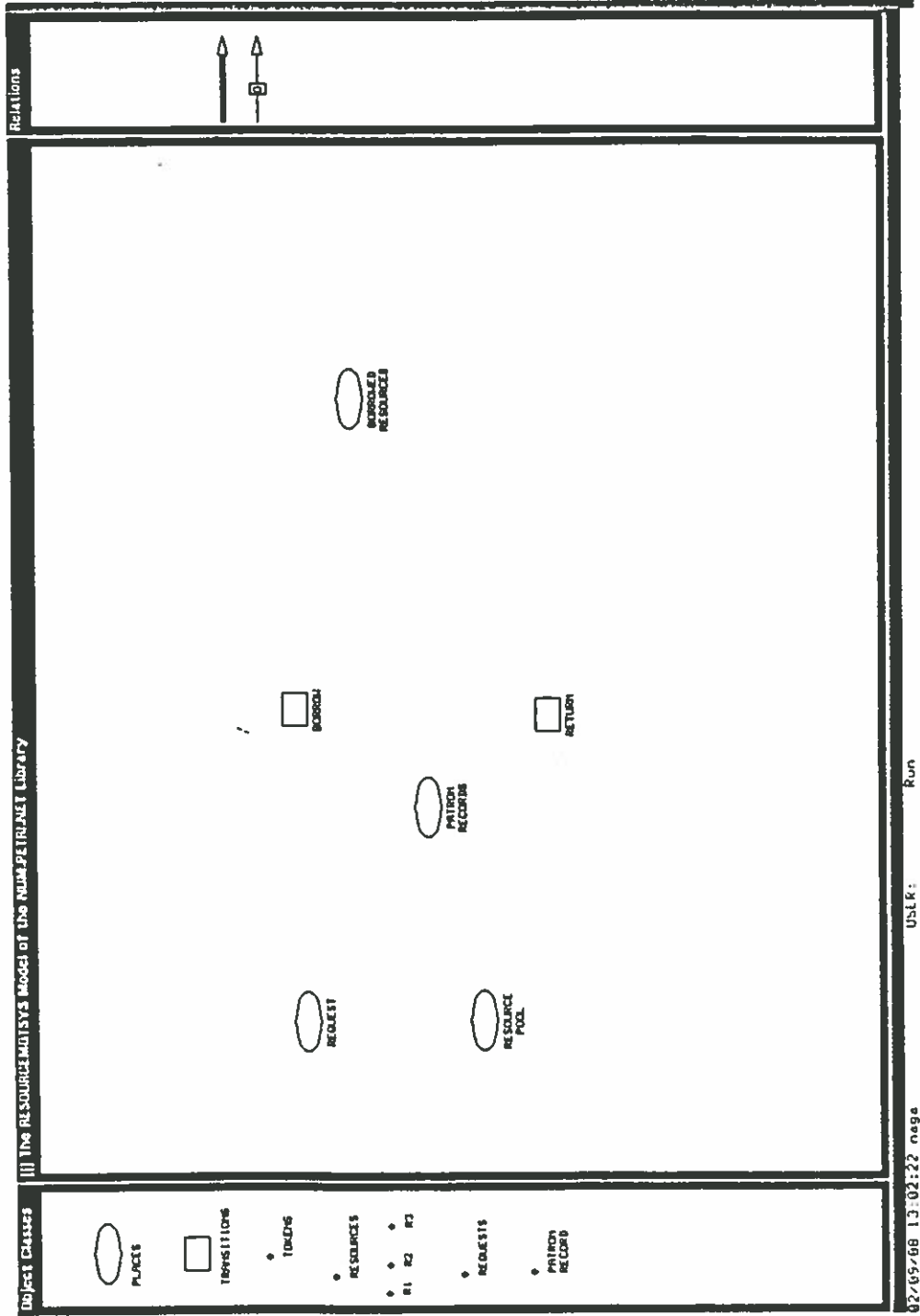


Figure 6: An ObjNPN in Construction



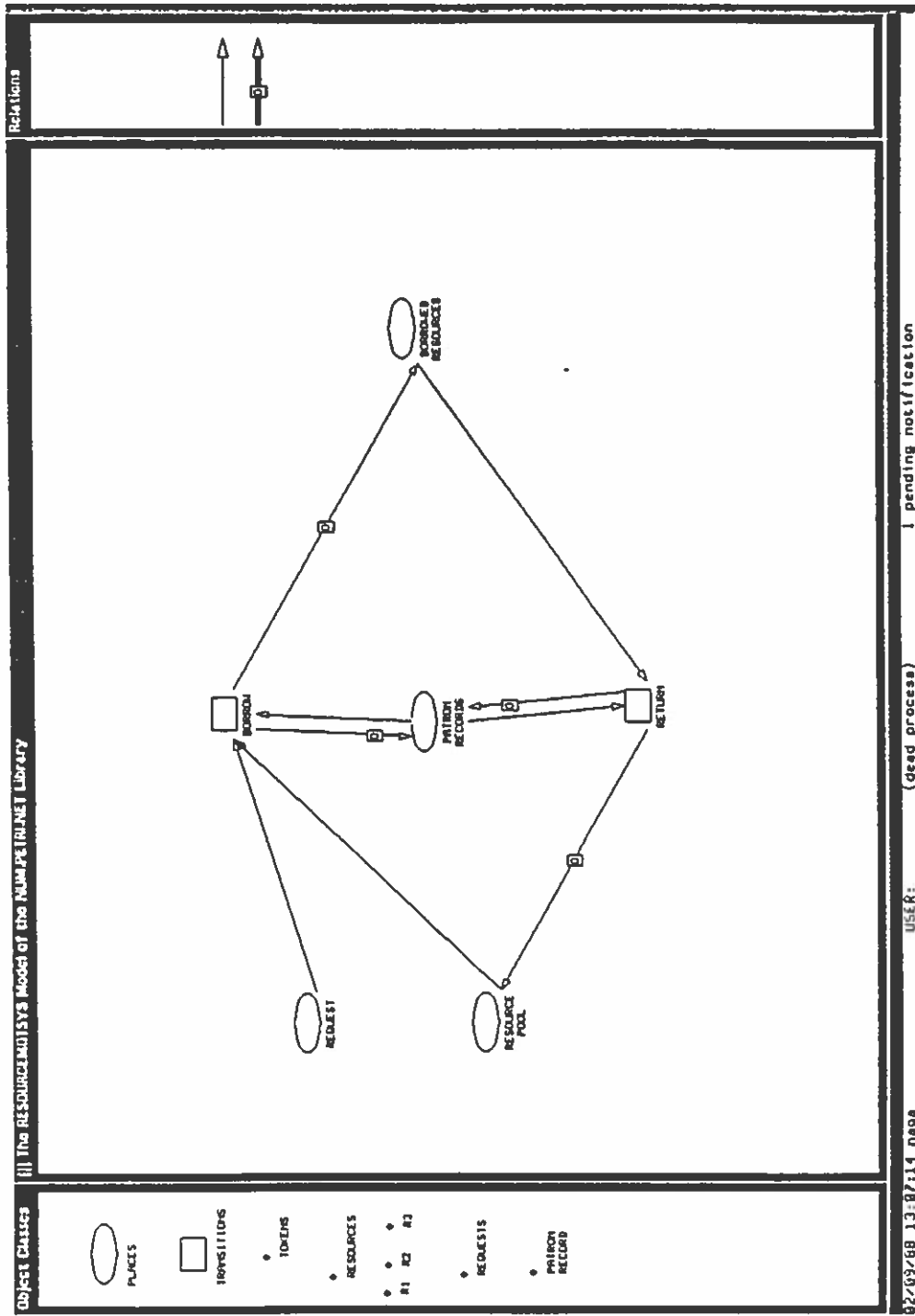


Figure 7: A Complete ObjNPN

present from BORROW to BORROWED RESOURCES. To capture the notion of updating the borrowing patron's record, another output arc from BORROW to PATRON RECORDS is added. In a similar way, the input and output arcs for the RETURN transition is added as shown in figure 7.

The inscriptions for these arcs are as follows :

<i>Arcs</i>	<i>Input Conditions</i>
REQUEST to BORROW	(R1 REQUEST)
RESOURCE.POOL to BORROW	(Res1 RESOURCE)
PATRON.RECORDS to BORROW	(Pr1 PATRON.RECORD)
PATRON.RECORDS to RETURN	(Pr2 PATRON.RECORD)
BORROWED.RESOURCES to RETURN	(Res2 RESOURCE)

In this example the DTs for each of the input arcs are the same as their corresponding ICs. That is, the enabling tokens are the only ones which are to be removed from the input places for both transitions. For example, the request token R1 picked for firing BORROW is the only request which will be deleted from the place REQUESTS.

The transition conditions(TCs) for the two transitions are as follows :

(a) BORROW

- (i) R1.Type = Res1.Type
- (ii) R1.ID = Pr1.ID
- (iii) Pr1.Status = "Can.Borrow"

(b) RETURN

- (i) Pr2.ID = Res2.Borrower

The transition actions for the two transitions are as follows :

(a) BORROW

- (i) Res1.Borrower := R1.ID
- (ii) Pr1.Status := Res1.Type
- (b) RETURN
  - (i) Pr2.Status := "Can.Borrow"

The variables R1, Res1, Pr1, Res2, and Pr2 are all Ivars (cs-variables in NPN terms) which get unified to token instances when the ICs are tried.

This model can be executed using an animated mode, which shows the tokens moving in and out of the places. If executed in an interactive mode the system prompts the user at the choice points to decide on the transition to fire. If an agenda is specified then the system gets this information from the agenda.

### Features of ObjNPN

This section will bring out the significant features of ObjNPNs, which makes it a useful candidate for a specification language. The advantages of having a formal language for software specification are many. Before looking at the advantages let us define what a formal specification means. A specification is formal if it is written entirely in a language with explicitly defined syntax and semantics. As described in [17], the advantages of formal specifications are as follows:

- (a) It can be studied mathematically.
- (b) It can be meaningfully processed by a computer. This is because formal specifications are well-defined and unambiguous.
- (c) Any inconsistency or incompleteness in a specification which is syntactic in nature can be detected automatically.

Having seen the advantages of a formal specification language, now let us examine ObjNPNs as one such language. ObjNPNs have most of the properties [1],

required for producing a good software specification. ObjNPN is a process-oriented system specification language. The transitions can be used to specify the processes/events in a system, while the places may represent the states of the system.

Balzer and Goldman explain that a system specification must be a cognitive model. Since it is possible to describe the system with ObjNPNs as a user would perceive, I feel that ObjNPNs would be an aid to satisfy this requirement. We can have net representations of a system at various levels. Leiden et al. [16] use Petri nets to describe an expert system architecture divided into three levels namely, Meta-level, Middle-level and Lower-level; each level serving a different purpose but linked to the Middle-level. We could also have different perspectives of a system described using the nets.

Next, it seems almost essential to have specifications which are operational. Since it is hard to imagine a specification which is *in toto*, it is necessary that our environment allows partially defined specifications to be executed. ObjNPNs seems to serve this purpose, as one of the major significance of petri nets in general happens to be their executability. Any petri net model provided with a mechanism to handle non-determinism is executable. Given an interactive, graphical petri net system, this execution can be visually seen by the user. This will help the user to detect errors in his/her specification and then to modify the specification.

It has been suggested that specification languages must have constraint capability. In ObjNPNs, constraints are represented in various forms such as ICs and TCs. Data typing can be introduced in ObjNPNs by having various classes of tokens. Further type constraints can be added to the places in the form of their capacity description. In other words, we can specify that no more than a prespecified number (which could be 0) of tokens of a particular type ever be at a place.

While it is true that we need some global constraints for describing most systems, there should be a mechanism to make the constraints localized as well. This falls in line with the principle of having a specification localized. It makes interpre-

tation of the constraints easier at most times if they are associated with a specific process or event than having them globally grouped together with constraints which have no meaning in the context. Further, if the specification changes are associated with specific processes then we only need to make changes in constraints of these processes. Thus, having localized constraints is advantageous in the specification maintenance. In ObjNPNs constraints are always localized.

Another significant feature of petri nets is their ability to express nondeterminism. Since it is natural for most systems to have a nondeterministic behavior, specification languages should allow description of these choice points in the system behavior. As mentioned earlier, in this system an agenda can be specified before execution of a model(net). This agenda describes the choices to be made at the nondeterministic choice points thus deciding the transition to be fired. The agenda could be used as part of the specification as a global controller of the behaviors of the system. Another use of agenda comes during the testing of the specifications. The user can set up various agendas and then execute the specifications to check if they result in the desired behavior.

Existing specification languages like GIST are very complex. It is cumbersome for the user to write a specification document and the specification is often too complex to analyze. In ObjNPNs we have tried to avoid this complexity.

#### Limitations of ObjNPNS and Extensions

Despite the attractive features of our language for specification there are some limitations to it. First of all the basic drawback lies in representing a problem in the form of a ObjNPN. A net representation more often than not is used to model states of a system and to make changes in them. But if we want to write a problem specification, like the famous *dining philosophers* problem, this language seems inadequate. Though it is more suitable for representing the solution for the philosophers problem, it is cumbersome to state that no philosopher should ever

starve in a petri net specification. Our notion of an agenda could be extended to perform this in a simpler way.

There are no means for specifying global constraints on a model. Though this supports the theory of localization, there are times when it is useful to make global restrictions on the behavior of a system. But this drawback can be removed by extending the definition of the ObjNPN, to allow execution of these nets in a closed world containing global constraints on the net elements.

Finally there seems to be no way of expressing future references or result specifications in this language. We can probably avoid these by describing constraints based on the current state if temporal operators are available to describe the constraints. In addition to the capacity constraint, we require a time constraint on a token to make it stay at a particular place. Another useful constraint type is to have a token generated at a place in a finite time. These constraints will simplify defining of the dining philosophers problem in the net form.

### Summary

In this chapter we have introduced the specification language ObjNPN. ObjNPN is an extension of Petri nets. We have looked at the various features of this language and how it is superior to the basic P/T nets. These features can be used to our advantage in the specification process. Finally, we have seen the limitations and extensions required for this modeling language to make a more powerful specification language.

Having introduced ObjNPNs, we now move on to the description of the implementation of the Critic in the next chapter.

## CHAPTER 3

### THE ANALYST

#### Introduction

The application domain chosen for the demonstration system is resource management systems. This domain covers a wide range of applications including management of physical, human and information resources. We come across a variety of problems involving user interaction, resource overflow and underflow, security of resources, etc.

Resource management systems are generally built around a set of policies, e.g., resource overflow. During analysis of specifications, it would be helpful to use knowledge about the domain policies. Since the main purpose of an analyst involves finding the intent of the clients expressed in the specifications, it is essential for the analyst to detect the policies the clients desire to be implemented in their system. Our system criticizes a given specification for the presence or absence of the components which support the policies applicable to the domain that have been either marked important by the client or left unknown. We use Library Systems as a representative of resource management systems.

#### The Domain Model

Concepts from a Library domain have been used to form the domain model. We use a frame-based representation to depict the domain components. We have tried to use an approach similar to Greenspan's RML [14], using the KEE [15] units as our representation. Since we are using ObjNPN as our specification language, we have constructed our domain model in terms of the basic elements of the language.

In the following subsections we will discuss the components which make up our domain model.

### Resources

We have introduced the main class of resources of the domain, namely books as a subclass of tokens. The tokens are the elements which flow through the net and hence are good for representing the flow of resources in a management system like the library. Since the library has several types of books (e.g., journals, texts, proceedings) the specification should distinguish among these types of books. We are basically seeking *object typing* here. The hierarchical nature of token classes in our ObjNPN allows us to do exactly this. We define each of these types of books as subclasses of books. If there are other types of resources besides books in a library, like newspapers, they can be also be created as a subclass of tokens. This allows us to refer to books and newspapers separately in our specification. At the same time we can refer to the journals, texts or proceedings as books if needed as in the following example :

Books can be checked out of the library, while newspapers can only be read on site. Texts can be borrowed on a regular basis while the journals can be borrowed only for two hours.

In the first statement we use *books* as a common term representing journals, texts and proceedings. However, in the next statement we distinguish between the texts and journals. Such specifications can be easily translated into our representation, using the token classification.

Other resources include library space, library staff and finances that can also be represented by defining new types of tokens.



## Resource Operations

The main resource operations in the library system are checkin and checkout. These operations are quite elegantly represented in the form of transitions in an ObjNPN. This knowledge is encoded as part of the domain knowledge base. We do not have explicit transitions predefined in our system. Their addition will prove helpful.

## Constraints

We would like to add some knowledge about the constraints on the resources and the patrons in this system. But currently, the constraints themselves have not been represented in the knowledge base. Instead we have provided operators and predicates for constraint implementations. These are represented in the form of KEE units. These operators are intended for use while defining the constraints in higher level domain operations like CHECKOUT. An example of one such predicate is IS.BORROWER.A.PATRON? which can be used to establish the policy that only registered patrons of the library be allowed to borrow the library resources.

## Policies

The policies of the domain form the most significant part of the domain knowledge. These policies are stored in the form of a directed dependency graph, as shown in figure 8. We have identified the following broad classes of policies which apply to resource management systems in general :

- (a) Allow users to have access to a large selection of resources to choose from.
- (b) Allow users to gain access to a useful working set and keep it as long as necessary.
- (c) Maintain privacy of users.

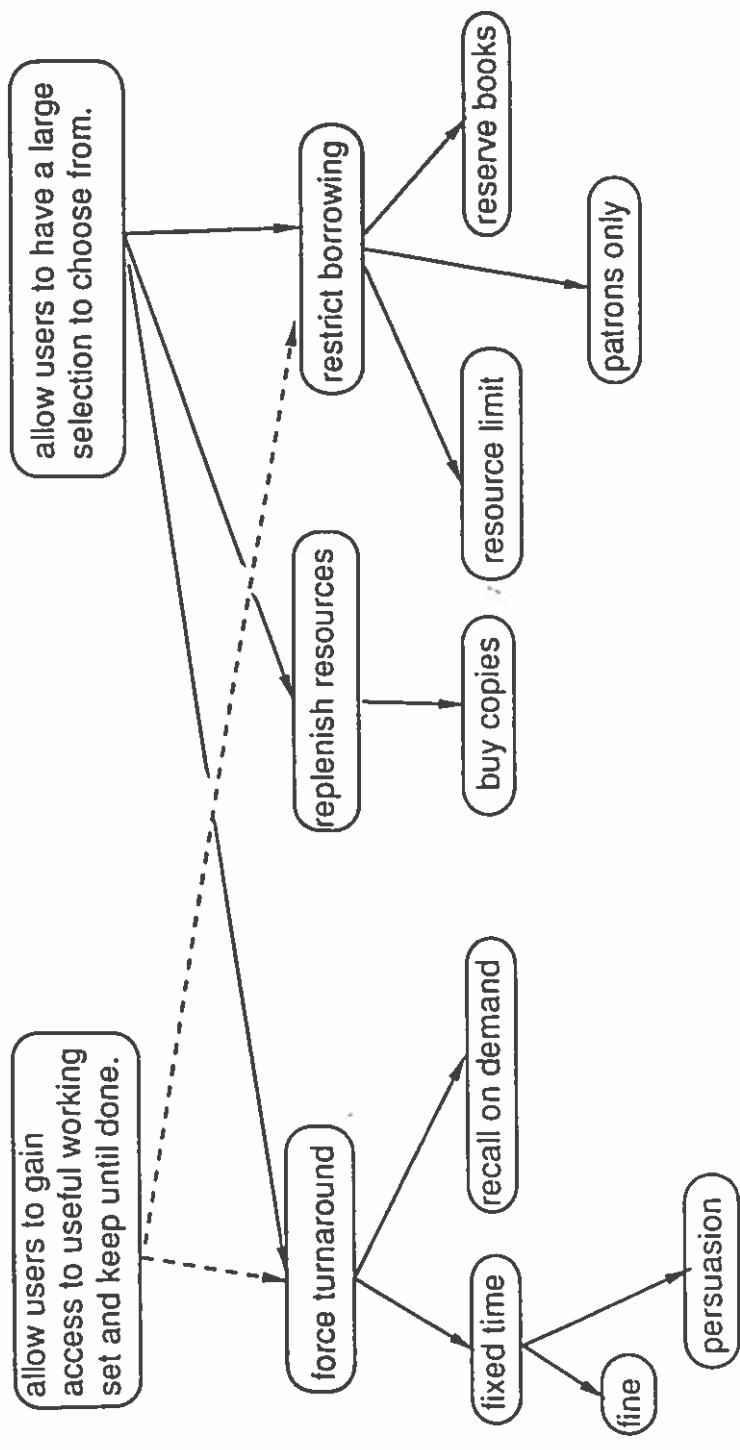


Figure 8: The Policy Graph for Library Systems

- (d) Account for human foibles e.g., losing items, forgetting.
- (e) Account for resource constraints e.g., money, staff, and time available to run and maintain the system.

Each of these can be further refined as shown in the policy graph. The directed policy graph(DPG) has two types of arcs :

- (i) Positive dependencies : If *Policy1*  $\longrightarrow$  *Policy2* is a positive dependency arc it means that implementing sub-policy *Policy2* will help in adhering to the *Policy1*. For example, in the library system, if the basic policy(*Policy1*) is to allow borrowers to have a large selection of books to choose from, then this policy can be adhered to in many ways. One could restrict the borrowing of books out of the library (*Policy2*); borrowed books could be forced to turnaround quickly (*Policy3*); or one could prevent theft of books from the premises (*Policy4*). Each of these three subpolicies, namely *Policy2*, *Policy3* and *Policy4*, form a positive dependency of the parent policy, *Policy1*. The interesting aspect to be noted here is that all the three subpolicies contribute both individually as well as collectively towards the parent policy. This may not always be so. For example, to implement the policy of forcing the turnaround of books, one could have fixed time borrowing or have recall on demand. To have both of these in a library seems to create conflicts. Situations such as :

A person borrows a book on a fixed time basis. But a request for the book comes in the next day and recall on demand policy results in the book being recalled the same day. The borrower gets to keep the book for only one day.

might occur. But at the same time, a compromise can be made between these two policies. As is found in most libraries, the book is lent out for a fixed but a long period. In the mean time if a request comes for the same book, then

after allowing a minimum period (say a week) the book is called in. In this way the patrons are ensured of borrowing books for atleast a week, while the library is ensured of its power to force the book back both by recall and by fixed time lending. Compromises of this kind are discussed from a software engineering point of view in the next chapter.

- (ii) Negative dependencies : If  $Policy1 \rightarrow Policy4$  is a negative dependency arc it means that implementing Policy1 negatively affects Policy4. That is, Policy1 and Policy4 may be conflicting policies. For example, keeping books on reserve in a library (Policy4) has a negative affect on Policy1, borrowers thereby have a smaller selection of books to choose from. These arcs have been shown as dotted lines in figure 8.

Negative dependencies have been neglected due to time constraints. But they should be present in a comprehensive policy graph for any domain, as they could play an important role in the analysis of policy implementations.

In the DPG described above, the leaves of the graph are those nodes of the graph which do not have any positive dependencies budding out from them. One can look upon these policies as atomic. Along with these leaf nodes or atomic policies, we can store some information regarding the policy. The conditions which form the requirement for the leaf policy is stored with it. In our DPG the policy of exercizing a borrowing limit on the patrons is an atomic policy. This policy can be implemented in a library system by simply introducing a check on the number of books borrowed by the patrons before checking out a book to them. This information is stored with the policy using some predefined predicates and operators.

A typical policy unit looks like the following :

Policy BORROW LIMIT

- (a) *Conditions* : ((CHECKOUT (IS.PATRON.UNDER.BORROW.LIMIT)))

- (b) *Description* : There should be an upper limit on the number of books a person can borrow.
- (c) *Negative Examples* : (AGENDA1)
- (d) *Positive Examples* : ()

Here the conditions are described in a predefined syntax similar to an association list. *Is patron under borrow limit* is a predicate provided by the system for the user to make use of. The *description* is a text which could be used to inform the user about the policy. The *examples* are just a list of agendas. AGENDA1 is defined in the Appendix.

When trying to locate a policy in a specification defined by the user, the system tries to match this information provided by the policy unit with the user's specification. If a match is found we infer that the policy has been taken care of. In case of a failure in the matching process, it is assumed that the policy has not been implemented. This is when we adopt a different strategy and try to get help from some carefully selected example domain behaviors.

### Domain Behaviors

With each of the atomic policies we have some example cases. These are used as a representation of the domain behaviors under particular circumstances. The examples are to indicate the usefulness of the policy. There are two kinds of examples, namely, positive and negative. The negative examples show the need for the policy. They basically provide inputs for the specification to be tested on. Since these inputs are tailored to check for the existence of a particular policy, these examples are an attempt to show that an undesired behavior might result if the policy is not implemented. This would fall in the category of black box testing. The positive examples are intended to show the user that utilizing the policy might

result in a better system behavior. However, the system in its present state does not make use of the positive examples.

The examples are stored in the form of description of a subnet along with an initial marking for it. The subnet is indicative of the part of the specification which might implement the policy. In comparison to the work on legal reasoning systems [25], these subnets can be called *cases*. The policies are analogous to the *dimensions* used by Rissland. These policies act as an index to the cases stored in the knowledge base. The major difference between Rissland's *case* and the *case* here is that the our *case* can be an abstract one while the legal cases are real examples from the archives of legal proceedings.

These examples have their operational aspect described in the form of agendas used for running a ObjNPN. An agenda decides the behavior of the net; this helps us bring out domain behaviors from the specification model when we run an example on it. Negative examples bring out the undesired behaviors of the system described by the specification. For instance, the example displaying the resource underflow will have an agenda stating the following:

- (a) Set up some books initially in the library and set up some requests for the books.
- (b) Keep performing the Checkout transaction as long as a request remains.

Following this agenda the execution of the model will show all the books being borrowed (without any coming back). Finally, there would still remain a request which cannot be satisfied since no books are left in the library.

This system provides some primitive lisp constructs to help define the examples and agendas. The appendix provides a listing of an instance of an example and a description of an agenda.

The domain model has to be constructed before the critic can be used on any new domain. But we are optimistic that a network of interrelated domain

models can be developed where concepts of one domain can be shared by other domains. Having introduced the domain model we now move on to the working of our demonstration system.

### Overview of the System Operation

In this system, a person can develop a specification interactively, and then try analyzing it. To start with, it allows a primitive way of analysis that is, through symbolic execution of the specification. The users can execute their specification, which is in the form of a ObjNPN and see a visual execution of their specification. This symbolic evaluation of the specification will give the users a picture of their ultimate system's behavior.

The second way of analysis allows the users to find out if various policies concerning libraries have been implemented in their specification. The users can ask the system if a particular policy of the domain has been included in the specification. This feature can be used to help understand the specifications. The system goes through its policy knowledge base and tries to locate the policy requirements in the specification. If the policy asked for was not an atomic policy, then the DPG is traversed along the positive dependencies of the given policy. Existence of each positive dependency is recursively checked. Once we reach an atomic policy the requirements along with it are checked. If a part of the requirement is not found, then the user is notified by the system that the atomic policy was missing. Since we failed to detect the policy we assume that the policy has not been included. The user is shown an example depicting the need for the requirement which seemed to be missing. These examples are in the form of executions of the specification with particular initial markings for it. This execution brings out the need for the missing requirement, which in our case was the atomic policy. The system continues its search for any other missing positive dependency in the specification.

### An Example

In this section we look at an example of specification analysis in this system. Figure 9 shows a specification model of a library system. This is an incomplete specification which shows just the checkin and checkout transactions in the form of transitions BORROW and RETURN respectively. The word incomplete is used since the specification lacks other features which would make a complete library system. Some missing features include the concept of introducing new books into the library and staff handling. Our system, being of a limited domain model, allows only specifications involving these two operations.

The layout of the net is similar to the example explained in chapter II. Here the place BOOKS.ON.SHELF corresponds to the resource pool. The place BORROWED.BOOKS corresponds to the place RESOURCES.BORROWED. There is an additional place called RETURNABLE.BOOKS which can be thought of as a database which keeps track of books checked out of the library. The place READER is where readers place requests to borrow books.

At anytime during the construction of the specification, the user can make use of the critic. The user basically can ask the critic to ensure that the specification incorporates a given policy. Figure 10 shows the user expressing the request for the policy *large selection*. The policy is that the library should maintain a large collection of books for the use of patrons. The trace of the critic at work in figure 11 shows exactly how the policy is being checked. The critic looks in the policy tree for the various ways in which *large selection* can be enforced in a library. In this case it finds the following two possibilities :

- (a) *enforce checkin* - Forcing the patrons to return the books to the library could result in a large collection to remain at the library at all times. This in turn is generally accomplished by *recall on demand* policy, which means if there is a request for a book then call the book in. The critic thus goes on to other ways



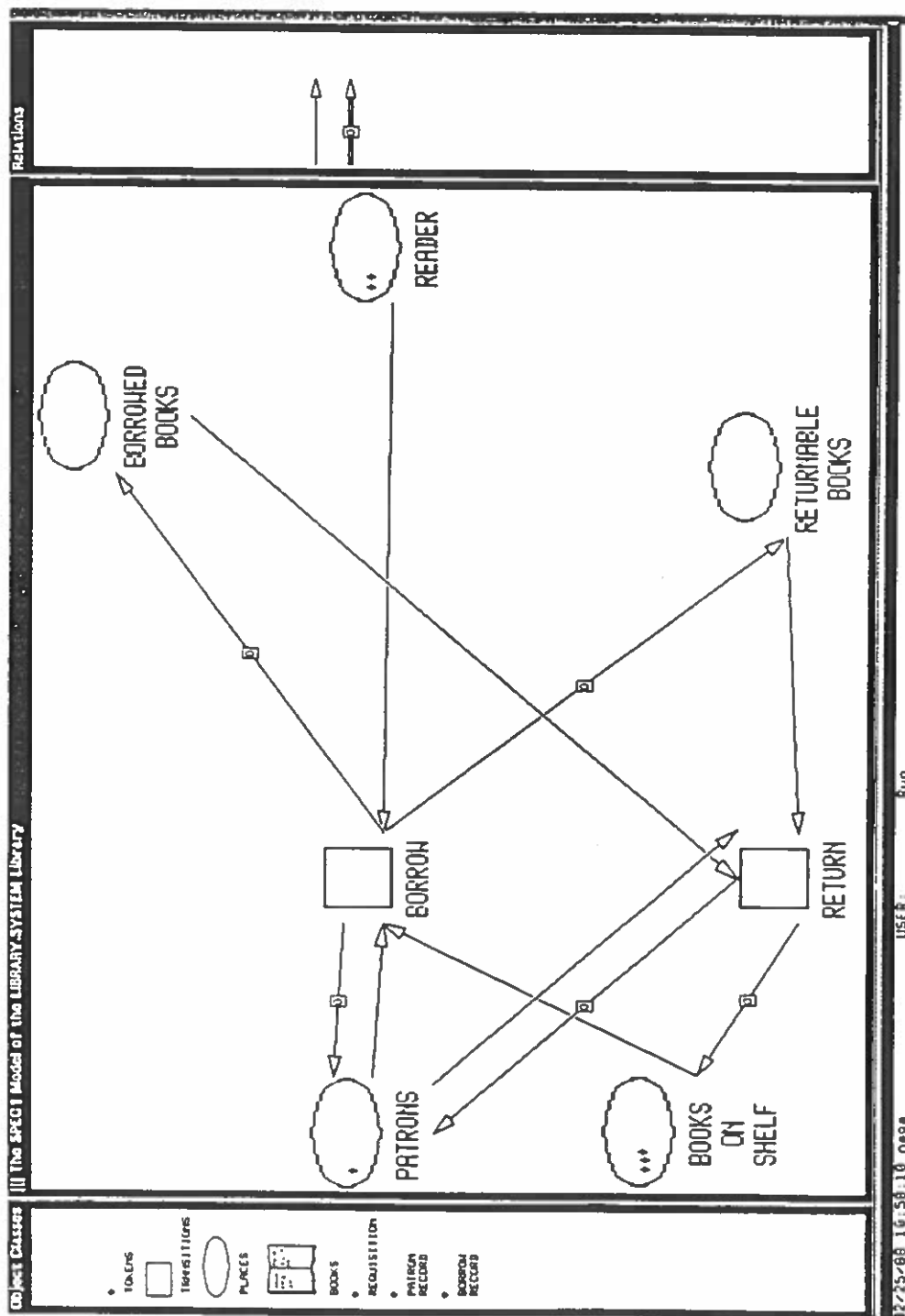


Figure 9: A Specification Model of a Library System

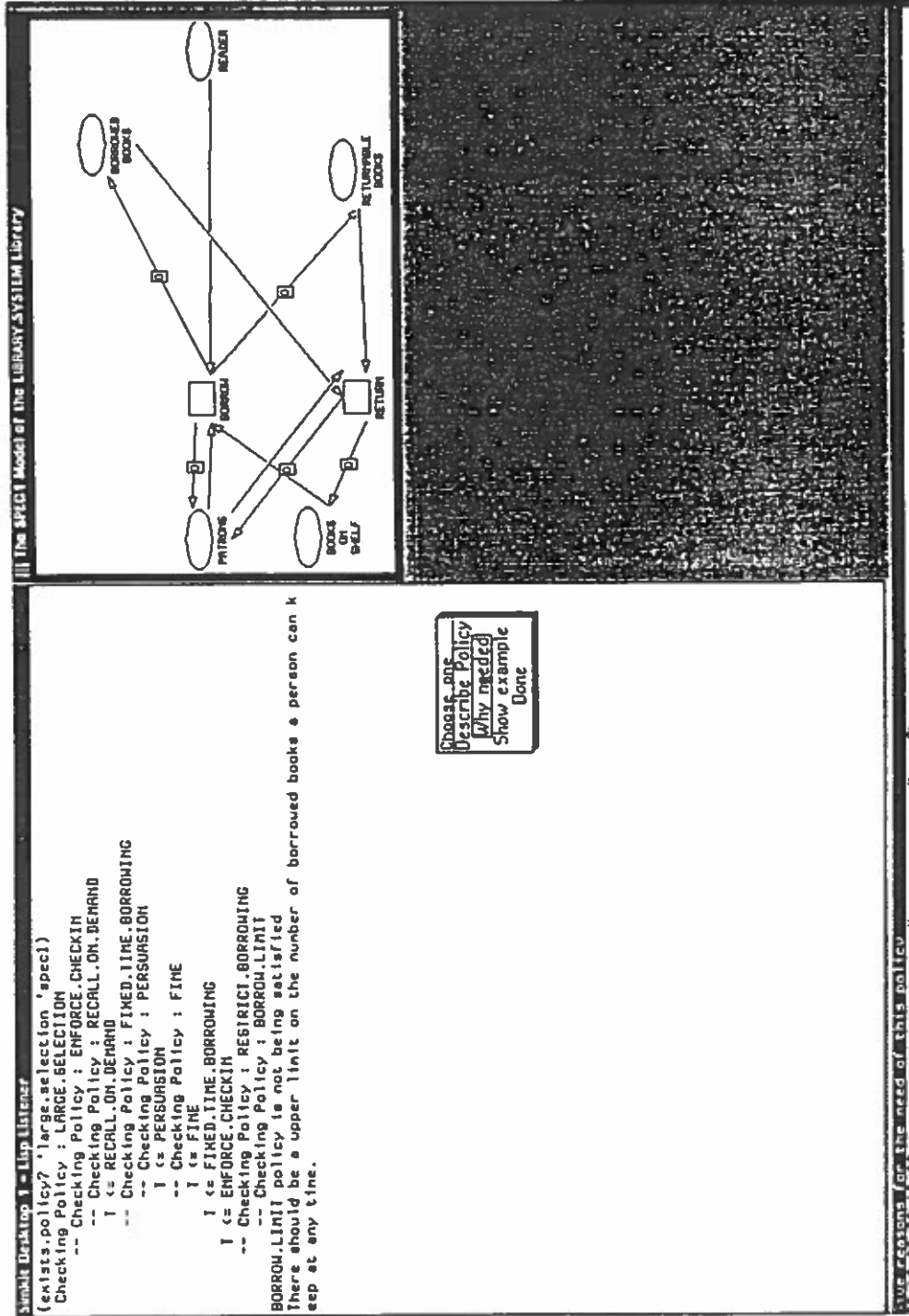


Figure 10: The Critic in use I

of enforcing the checkin of books. When it finds that all these policies have been incorporated in the specification it moves on to the other possibilities of having the *large selection*.

- (b) *restrict borrowing* - Another way for ensuring a large selection is by restricting the number of books to be borrowed out of the library. One way to do this is to set a *borrowing limit* on each patron of the library. The user in defining the specification must have introduced a transition condition which checks that each person borrowing a book does not exceed the borrowing limit. The critic looks for such a transition corresponding to CHECKIN in the user's specification. Since the transition corresponding to checkin is called BORROW in the user's specification the critic fails to identify a match. It then asks the user to specify this information by querying interactively.

The critic maintains a correspondence table which keeps track of the correspondences between the terminology adopted by the user in defining the specification and that used in the domain model. The critic now goes through the checks as prescribed in the policy *borrow limit*. There happens to be only one check in this case and that marks the existence of the transition condition using the predicate IS PATRON UNDER BORROW LIMIT. The critic searches for this and fails to find such a transition condition associated with BORROW. Hence, as figure 11 shows, the critic reports to the user that the policy *borrow limit* is not being satisfied. It then asks the user if an example to test the same is wanted. An interaction of this kind occurs between the user and the critic through menus; one such menu is shown in figure 10.

The critic now tries to apply the examples associated with the policy *borrow limit* on the user's specification. The running of the example involves being able to correctly place the initial marking for the net. To do this the critic again has to go through a matching process. Here again on failure it falls back on querying the user

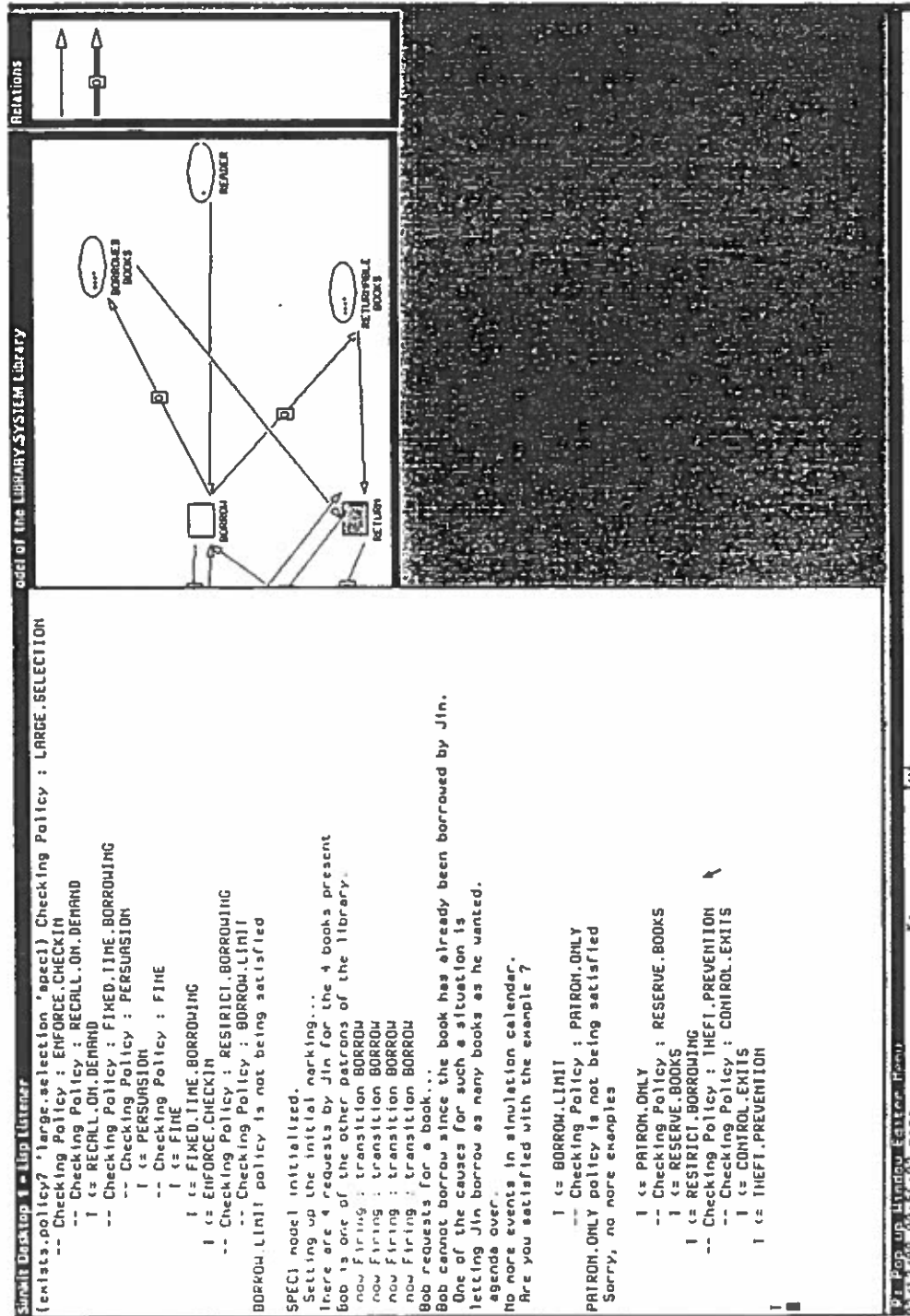


Figure 11: The Critic in use II

for matching the *places*. Once it finds the corresponding places the net is set up with a initial marking. The rest is now just executing the net. The critic does not work any further as far as the result of execution is concerned. The user is expected to infer from the resultant behavior of the specification during the execution that the policy *borrow limit* is necessary to prevent the books to be emptied from the library.

After the execution the critic continues with the remaining possibilities still uncovered. Thus the whole policy tree is looked into. Figure 11 presents a commentary of the execution of the example.

This chapter now brings us to the conclusion of this work. We will look at some of the shortcomings in this research in the concluding chapter. More importantly, the concluding chapter will examine future prospects and potential improvements to our approach.

## CHAPTER 4

### CONCLUSION

This chapter looks at the positive and negative implications of the methodology we have used for analysis of specifications. It suggests possible extensions and variations from our approach for future work.

#### A Discussion

In the methodology presented for analysis of specifications the matching process poses the major hurdle. Detection of components related to the policies can be successfully accomplished only if we are able to match them with the knowledge stored in the domain model about the policy. What does it mean if our critic fails to detect the existence of some policy's components (because it could not match the requisite elements from its knowledge base to the user's specification)? Our system does not ensure that the user did not adopt alternate ways for expressing the policies. The only way to accomplish this imposes more restrictions on the user. This can be attempted by expanding our domain model to include more predefined components (e.g., transitions corresponding to the transactions/events) in the domain. The user is then restricted to use only these predefined components. For example, in our library system to ensure that the users define transitions corresponding to checkin and checkout transactions, we provide these transitions to them like we do for the resource tokens. To ensure that the users will define all the checkout related policies as its transition/input conditions, we have introduced the domain dependent predicates and operators in our system. These urge the user to define the transition conditions only using the operators/predicates known to the analyst in terms of the domain constraints.

While detection of presence(or absence) of policies in a specification threatens us with the concept matching problems, there is yet another phase of analysis which poses the same threat. This phase, as we saw in the previous chapter, is the presentation of the domain behaviors to the users in order to express the shortcomings in the specifications.

There are two questions which need to be answered here. First, one can ask if the example presentation is essential. Second, if it is essential then can we adopt a different mechanism to do so. The answer to the first question has been discussed in the introductory chapter. It is not absolutely necessary to make use of example presentation. But on the other hand, expert analysts find it the best means to get across to their clients. The legal reasoning process again can be used as a comparison here. The lawyers who advocate the legal cases have accepted examples as the most appropriate means for presenting their cases. Our critic in a software engineering sense is an advocate too. It tries to present its case that the user's specification might involve faults.

The answer to the latter question is again debatable. To begin with in our demonstration system, the examples are just a predefined set of domain behaviors. To present these cases then could be very simple if we were to make use of some *fixed* English texts. The texts could be presented to the user. But conceivably, this would be a bottleneck if we wanted to expand our system to incorporate the feature of generating examples on its own. We envision a complete critiquing system of this nature to have the capacity of generating new example domain behaviors from the few initially handcoded ones. Using Rissland's [24] approach of stretching cases along specified dimensions, we should be able to generate new cases. To be able to show these newly generated cases we need either a natural language generator or some other equally powerful mechanism. We believe that a visual symbolic execution of the user's specification would be another worthy mechanism. So we are convinced that the approach we have prescribed is on the right track.

Other problems anticipated with this approach can include the following. The critic may fail to match the components despite seeking help from the user. Besides, when trying to demonstrate the example, the example might fail to get the desired effect. The critic as of now does not do a post execution analysis of the example. Therefore it does not determine if the case presented produced the correct behavior from the specification. A post execution analysis on the state of the specification model might help in discovering if the example served its purpose, and if not, whether the critic failed to match the components correctly.

### Reaching Compromises

Our belief has been that any system design is a result of compromises. The designer often faces the problem of making an optimal choice between two or more mutually conflicting policies in a resource management system. For example, as we saw in the previous chapter, fixed time borrowing and recall on demand, are two policies that could be incorporated by reaching the compromise that the borrower will be allowed to keep the book for a minimum of one week. How one reaches such compromises is a very interesting facet of the specification design process.

If we have an example that covers the fixed time borrowing and another that covers the recall on demand, the system in its current form can express the need for both of these policies individually using the corresponding examples. But if the user then introduces both of them in the specification it would basically conflict with the general goal of the library to serve the patrons, as mentioned in the previous chapter (the patrons get to keep the book for only one day). What we need then are negative links to policies not just from individual policies as we have now but also from groups of policies. In this case policies recall on demand and fixed time borrowing individually do not conflict with our policy of serving patrons but taken together they do. Hence our policy graph is insufficient. We somehow need to identify such a group of conflicting policies.



One avenue to look into is the example generation through modifications suggested in the last section. Capability to generate example cases by stretching them along various dimensions might help us deduce the possible problems and hence might lead us to the right decisions. The dimension here refers to some higher level property of the examples. Since the examples are retrieved by indexing through the policies in our context, the policies can be looked upon as dimensions.

The examples, like Rissland's *cases*, should have focal slots which will be associated with specific dimensions of the examples. This would imply that the same skeleton of the example can be used by more than one policy. But each policy would be interested in a different focal slot of the example. Consider, for instance, the policy of maintaining the security of the library database. This has the atomic policy of *password clearance* as a positive dependency. The example associated with this policy would show that a user can possibly break the password by accident. One of the focal slots of this example is the length of the password. This focal slot will be associated with the *password clearance* policy. If the password is not *long enough* it can be broken. Of course, the user decides the optimal length for a password that can be broken accidentally. Another focal slot of the same example is the number of trials a person is allowed to give the password. This focal slot is associated with the policy that allows only a fixed number of trials. Once it is pointed to the user through an example that a person can get the correct password by trying a large number of times, the user can then determine the number of trials that should be allowed.

### More Automation ?

From our model of specification analysis, it can be observed that to be able to perform any useful analysis of specifications extensive knowledge of the domain is required. Moreover, we had to lay down some restrictions on the specification developers (for example, making them define the components of their specifications

using primitives that are known to the analyst). One may wonder whether, given all this knowledge about the domain, can we possibly automate the specification development process (that is, do *synthesis* as well as analysis). The following model gives a picture of such an automated system.

Consider the analyst for library systems. It knows the basic requirements for building a specification for a library system. Hence, it can set up these initial components for the user (for example: provide transitions like CHECKIN and CHECKOUT; places showing book storage, library catalogue database; and relations between these in the form of input/output arcs). The designers now basically have to decide on what kinds of policies they want in their system. This step can be accomplished by letting the designers browse through the policy graph and make selections on the policies they are interested in. The analyst can then fill in the details of these policies into the specification. The analyst can still critique the specification for missing or conflicting policies.

The problem we face is whether such a domain oriented analyst can be extended across domains. We could generalize some of the domain knowledge to address issues in related domains. For example the library system knowledge can be generalized to a subset of resource management systems.

### Contribution of this Thesis

This research document focusses its attention on salient issues summarized below :

- (a) *Goals of Analysis*: The need for the analysis of software specifications is seen in the form of its goals.
- (b) *A Specification language*: ObjNPN, which can be thought of as an implementation of GIST, is presented. Its features are discussed, showing its worth as a specification language.

- (c) *A Critic*: The design of a critic is presented whose goal is to validate the intentions in a software specification. This critic uses simulation and a case-based approach to accomplish this task.

There has been very little research into analysis of specifications [23, 27]. This research is an attempt to provide some impetus to future work. Even though this work does not involve the development of a complete system, it discusses most issues involved in the design of such a system. It also provides some new ideas for extensions to the system we have designed.

## APPENDIX A

## DESCRIBING INITIAL MARKING AND AGENDA

Here we present a listing of how the marking of a NPN for running an example is encoded followed by an example of an Agenda for running the example.

An example data in BNF-like form looks as follows :

```

<example> ::= (<entry>+)
<entry>   ::= (<token.type>
              <number>
              <place>
              (<slot><slot.value>)*

```

<token.type> : this is an atom denoting a token class(e.g., BOOKS)

<number> : this represents the number of token instances to be created.

<place> : this is the atom denoting the place where the token instances are to be placed.

<slot> : this atom denotes a slot of the token class given by token.type.

<slot.value> : this is a lisp s-expr which gets evaluated while creating the token instances. This allows reference to tokens which would have been created before, something like create a request token for a book token in the place shelves.

The following construct is helpful in setting up the token description in terms of other tokens present else where in the net. This returns a list of slot and slotvalues, so it can be used in defining the example :

```

(exists.tokens.in.place <token.specs>

```

```
(assignments))
```

where,

```
(token.specs) ::= ((token.spec)*)
```

```
(token.spec) ::= ((place)⟨tokens⟩)
```

```
(tokens) ::= ((ivar)⟨token.class⟩)
```

```
(assignments) ::= (slot)⟨slot.value⟩⟨assignments⟩|()
```

Example usage :

```
(exists.tokens.in.place
  (('shelves (t1 'books)(t2 'texts)
    (reference.room (t3 'journals)))
  'name '(get.value ,t1 'name)
  'title '(get.value ,t1 'title)
  type 'borrow)
```

An actual example listing :

The following is an example of setting up a marking to show the need for exercising borrowing limit on patrons.

```
(defparameter example1
  '((patron.record 2 patrons
    ((patrons.id "jim"
      fines 0
      borrow.limit 10)
     (patrons.id "bob"
      fines 0
      borrow.limit 10)))
  (requisition 4 reader
    ((borrowers.id "jim"))))
  (books 4 books.on.shelves
    (exists.tokens.in.place
      ('reader
        (rlst 'requisition)))
    (let (lst)
      (do ((r rlst (cdr r))(cnt 1 (+ 1 cnt)))
          ((>cnt 4) lst)
        (setq lst (cons '(author (get.value
```

```

',(first r)
'author)
title (get.value
',(first r)
'title)
type 'borrow)
lst)))))))))
(defparameter example1.2
'((requisition 1 reader
(exists.tokens.in.place
('borrowed.books
(booklst 'books)))
'((author (get.value ',(first booklst) 'author)
title (get.value ',(first booklst) 'title)
type 'borrow)))))))))

```

### Agenda :

An Agenda is a list of expressions. An expression can be either a list of transition names or a list of two elements with the first element as the atom `*eval*` and the next element as an lisp s-exp to be evaluated. The following is an example of an agenda definition. The s-exp to be evaluated uses some predefined functions like "flush.all.places" and "set.up.example".

```

(setq agenda1 '(*eval* (flush.all.places 'spec1))
(*eval* (format t "~& Setting up the initial marking...~%" ))
(*eval* (set.up.example example1 'spec1))
(*eval* (progl
(format t "~&There are 4 requests by
Jim for the 4 books present~%" )
(format t "~&Bob is one of the other
patrons of the library.~%" )))
(borrow)
(borrow)
(borrow)
(borrow)
(*eval* (progl
(format t "~&Bob requests for a book...~%" )
(set.up.example example1.2 'spec1)
(format t "~&Bob cannot borrow since the book

```

```
has already been borrowed by Jim.~%"")
(format t "~& One of the causes for
such a situation is")
(format t "~&letting Jim borrow as many
books as he wanted.")))
```

```
)
```

## BIBLIOGRAPHY

- [1] Balzer, R. & Goldman, N.,(1979). Principles of Good Software Specification and Their Implications for specification Languages, Proceedings of IEEE Conference on Specifications of Reliable Software, 36-46.
- [2] Barstow, D.,(1986). Artificial Intelligence and Software Engineering, Schlumberger-Doll Research, Ridge-field, Connecticut.
- [3] Boehm B.W.,(1984). Software Engineering Economics, IEEE Transactions on Software Engineering, SE-10(1).
- [4] Carbonell, J.G., & Minton, S., Metaphor and Commonsense Reasoning.
- [5] Cohen, D.,(1983). Symbolic execution of the Gist specification language, Proceedings of the 8th International Joint Conference on AI.
- [6] Feldman, Y.A. & Rich, C.,(1986). Reasoning with Simplifying Assumptions: A Methodology and Example, AAAI86.
- [7] Fickas, S.,(1987). Supporting the Programmer of a Rule Based Language, International Journal of Expert Systems, 4(2).
- [8] Fickas, S., (1987). Automating the Software Specification Process, Technical report 87-05, Computer Science Department, University of Oregon, Eugene, OR 97403.
- [9] Fickas, S., (1987). Automating Analysis: An Example, Fourth International Workshop on Software Specification and Design, Monterey.
- [10] Fickas, S. & Nagarajan, P., (1988). Critiquing a Software Specification, Technical Report 88-01, Computer Science Department, University of Oregon, Eugene, OR 97403.
- [11] Fickas, S., Collins, S., & Olivier, S., (1987). Problem Acquisition in Software Analysis: A Preliminary Study, Technical Report 87-04, Computer Science Department, University of Oregon, Eugene, OR 97403.
- [12] Gardner, A.L., (1987). An Artificial Intelligence Approach to Legal Reasoning, A Bradford Book, The MIT Press, Cambridge, Massachusetts.



- [13] Genrich, H.J. & Lautenbach K., (1981). System Modeling with High-Level Petri Nets, *Theoretical Computer Science*, 13, 109-136.
- [14] Greenspan, S., (1984). Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition, Ph.D. Thesis, Computer Science dept., Toronto.
- [15] KEE Reference Manual, Intellicorp, Palo Alto, California.
- [16] Leiden, S.H., O'Donnell, J.T., Peterson, E. & Shink, T., Building of Expert systems using Petri nets as a representation of knowledge flow, GTE Government Systems Corporation, Communication Systems division, 77 "A" Street, Needham Heights, MA 02194.
- [17] Liskov, B.H. & Berzins, V., An Appraisal of Program Specifications, *Research Directions in Software Technology*, edited by P. Wagner, 276-301, MIT Press.
- [18] Marr, D., (1976). Artificial Intelligence - a personal view, AIM 355, MIT AI Laboratory.
- [19] Meyer, Bertrand (1985). On Formalism in Specifications, *IEEE Software*.
- [20] Peterson J.L., (1981). Petri Net theory and the Modeling of Systems, Prentice Hall, Englewood Cliffs, N.J.,.
- [21] Reisig, W., Petri Nets - An Introduction, EATCS, Monographs on Theoretical Computer Science, 4, Published by Springer-Verlag.
- [22] Reisig, W., (1986). Place/Transition Systems, Advanced Course on Petri Nets, Bad Honnef.
- [23] Rich, C., Waters, R.C. & Reubenstein, H.B., Towards a Requirements Apprentice, Fourth International Workshop on Software Specification and Design, Monterey.
- [24] Rissland, E., Constrained Example Generation, Technical Report, COINS, University of Massachusetts.
- [25] Rissland, E. & Ashley, K.D., A Case-Based System for Trade Secrets Law, Technical Report, COINS, University of Massachusetts.
- [26] SIMKIT Reference Manual, Intellicorp, Palo Alto, California.
- [27] Swartout, W., (1983). The Gist behavior explainer, Proceedings of the National Conference on AI.
- [28] Swartout, W. & Balzer, R., (1982). On the Inevitable Intertwining of Specification and Implementation, *CACM*, 25(7), 438-440.

- [29] Wheeler G.R., (1985). Numerical Petri Nets- A Definition, RLR 7780, Telecom Australia Research Laboratories.
- [30] Wilbur-Ham M.C., (1985). Numerical Petri Nets - A Guide, RLR 7791, Telecom Australia Research Laboratories.