

KNOWLEDGE VERIFICATION OF INTELLIGENT AGENTS

A Thesis

Presented to

Department Computer Science

University Of Karachi

in Partial Fulfillment

of the Requirements for the Degree of

Bachelor of Science

in Computer Science

By

S.M.MASHHOOD ISHAQUE

JANUARY 2003

THESIS APPROVED.

Date

Dr. Nasir Touheed (Advisor)

ACKNOWLEDGEMENTS

I would like to thank God Almighty for granting me the abilities which I could not acquire otherwise, providing me with the best of the opportunities and surrounding me with such loving and caring circle of people. May He forgive His ungrateful creature!

I would like to thank my parents for extending me their unconditional love and support. I realize this even more as I progress in my life and I would never be able to return the favor.

I would like to thank my teachers for making me who I am. Especially I would like acknowledging Dr. Abbas K Zaidi for his guidance and being a source of inspiration for me. I would like to thank Dr. Nasir Touheed and Mr. Shahid Quereshi.

I would like to thank my friends and colleagues for believing in me even at times when I lost my own belief. They will always enjoy a special place in my heart.

ABSTRACT

The knowledge base of an intelligent agent represents the information that the agent uses in its decision making process. The performance of an agent depends critically on the quality of its knowledge base. Keeping in view that these intelligent agents are now being used in sensitive and time-critical applications ranging from Air-Traffic Control to Missile Control systems, it is imperative that the knowledge base of an intelligent system should be verified and validated.

This report discusses the implementation of a tool “RuleValidator” to perform the validation and verification of KBs, based on the methodology presented by Zaidi (1994). The methodology presented by Zaidi (1994) is based on transforming the rules in a KB to an equivalent Petri net representation and then applying the well-established analytical tools of the Petri net theory for the detection of errors.

CONTENTS

ACKNOWLEDGEMENTS	III
ABSTRACT.....	IV
CONTENTS	V
TABLE OF FIGURES	VII
TABLE OF ALGORITHMS	VIII
INTRODUCTION	1
1.1 Background and Motivation	1
1.2 Problem Statement.....	3
1.3 Problem Scope.....	4
1.4 Thesis in Outline.....	4
SOLUTION TO THE PROBLEM.....	5
2.1 User Interface	7
2.2 Control Unit.....	7
2.3 Rule Transformer.....	8
2.4 Rule Checker	8
2.5 RuleValidator Flow Chart	9
2.6 Closure.....	10
VALIDATION AND VERIFICATION OF DECISION MAKING RULES.....	11
3.1 Knowledge Base Structure Requirement.....	12
3.1.1 <i>Structure of Rules</i>	12
3.1.2 <i>Basic Inputs (U) / Main Concepts (ψ)</i>	12
3.1.3 <i>Mutually Exclusive Concepts (μ)</i>	12
3.1.4 <i>Absence of Direct Cycles</i>	13
3.2 Types of Problems	13
3.2.1 <i>Incompleteness</i>	13
3.2.1.1 Rules with ambiguous conditions.....	13
3.2.1.2 Rules with useless conclusions.....	14
3.2.1.3 Isolated Rules	14
3.2.2 <i>Circular Rules</i>	14
3.2.3 <i>Inconsistent (Conflicting) Rules</i>	15
3.2.3.1 Direct contradiction.....	15
3.2.3.2 Contradiction in input.....	15
3.2.3.3 Contradiction in conclusion.....	16
3.2.4 <i>Redundant Rules</i>	16
3.2.5 <i>Subsumed Rules</i>	16
3.3 Petri net Representation of Rules	17
3.4 Incidence Matrix representation of Petri nets. (Zaidi, 1994).....	18

3.5	Detection of Problematic Cases.....	19
3.5.1	<i>Detection of Incomplete Rules</i>	20
3.5.1.1	FindPath-to-Sources (FPSO) Algorithm	20
3.5.1.2	FindPath-to-Sinks (FPSO) Algorithm	20
3.5.1.3	Detection of Ambiguous Rules	21
3.5.1.4	Detection of Useless rules	22
3.5.1.5	Detection of Isolated Rules	22
3.5.2	<i>S-Invariant Analysis (Zaidi, 1997)</i>	23
3.5.2.1	Detection of Circular Rules	25
3.5.2.2	Detection of Inconsistent rules (Direct Contradictory rules)	26
3.5.2.3	Detection of rules having Contradiction in Input	27
3.5.3	<i>Dynamic Analysis (Zaidi, 1997)</i>	28
3.5.3.1	Detection of Redundant rules	30
3.5.3.2	Detection of Conflicting rules.	31
3.6	Closure.....	33
	CONCLUSION	34
	REFERENCES	35
	APPENDIX A	37

TABLE OF FIGURES

Figure 1.1: General architecture of the Disciple Learning Agent Shell (Tecuci, 1999)....	2
Figure 2.1: General Architecture of RuleValidator	6
Figure 2.2: Flow chart of RuleValidator.....	9
Figure 3.1: Petri net representation of rules.....	18
Figure A.1: Examples of graphs	38
Figure A.2: Examples of multigraphs.....	38
Figure A.3: Places, Transitions, and Arcs	39
Figure A.4: Petri Net PN1.....	41
Figure A.5: A Petri Net PN2 with a self-loop.....	42

TABLE OF ALGORITHMS

Algorithm 3.1: <i>Detection of Ambiguous rules</i>	22
Algorithm 3.2: <i>Detection of Useless rules</i>	22
Algorithm 3.3: <i>Detection of Isolated rules</i>	23
Algorithm 3.4: <i>Conversion of a Petri net to a Marked graph Petri net</i>	24
Algorithm 3.5: <i>Occurrence graph construction for redundant rules.</i>	31
Algorithm 3.6: <i>Occurrence graph construction for conflicting rules.</i>	31

CHAPTER 1

INTRODUCTION

This chapter discusses the motivation behind the project, along with the problem definition and scope of the problem.

1.1 Background and Motivation

The Intelligent Agent paradigm is the modern trend in building intelligent systems. The main component in any intelligent agent is its domain knowledge contained in its Knowledge Base (KB). The usual approach for KB development is to use knowledge acquisition (KA) techniques to elicit the knowledge from the expert and represent it into the knowledge base. This approach requires a knowledge engineer for the elicitation and representation of knowledge in the knowledge base (KB). The bottleneck in this approach is that it is very difficult for an expert to formalize his/her problem-solving strategies in a set of rules. Also, there are many rules and facts about the domain that the expert thinks of as the *Common Sense* and often fails to formalize them. This problem is generally called as '*The Common Sense Knowledge Problem*' in the Artificial Intelligence (AI) community.

The second approach for building the knowledge-based system is to use machine-learning algorithms to extract knowledge from the data and represent it in the knowledge base. The bottleneck in this approach is that it requires a lot of data.

An integrated Machine Learning and Knowledge Acquisition (ML & KA) approach to the problem of building the knowledge base of an intelligent agent can take advantage of their complementary natures; using machine learning techniques to automate the knowledge acquisition process, and knowledge acquisition techniques to enhance the power of the learning methods.

The methodology for building intelligent agents, proposed by Tecuci (1998a), named 'Disciple' makes use of the integrated ML & KA approach for the KB development. The current architecture of the Disciple shell (Tecuci, 1999), build upon the 'Disciple' methodology is presented in figure 1.1.

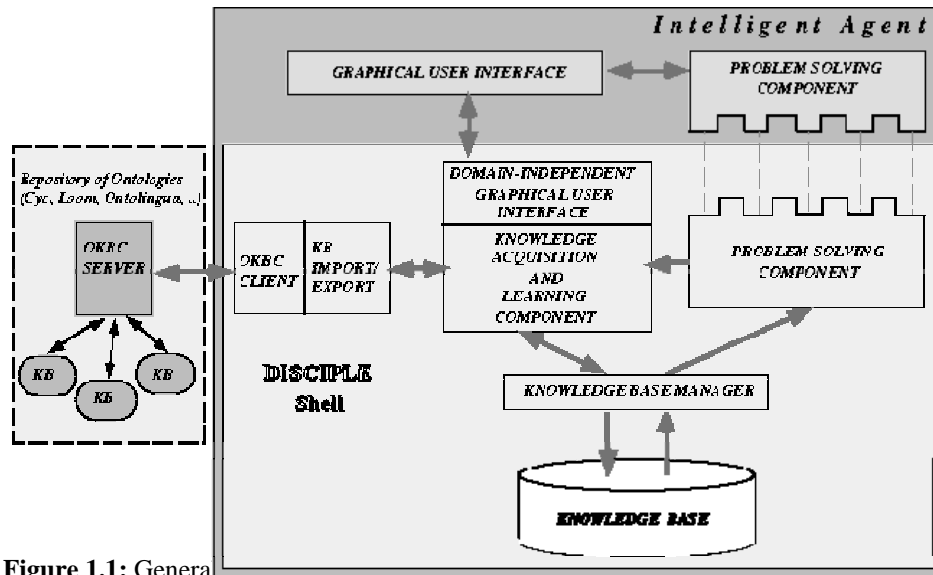


Figure 1.1: General

As is evident from Fig. 1.1, the Disciple shell uses not only the local KB but also the repositories of ontologies located on remote Open Knowledge Base Connectivity (OKBC) Servers (Chaudhri *et al.*, 1997), for the decision process. But there is no mechanism for checking both KBs for the presence of following erroneous rules in the KB.

1. Incomplete Rules
 - a. Ambiguous Rules
 - b. Useless Rules
 - c. Isolated Rules
2. Circular Rules
3. Inconsistent Rules
 - a. Direct Contradictory Rules
 - b. Rules having Contradiction in Input
 - c. Rules having Contradiction in Conclusion
4. Redundant Rules
5. Subsumed Rules

For a detailed description of these problematic cases see Chapter 3.

1.2 Problem Statement

The local KB as well as the remote ontologies must be validated and verified before their use.

1.3 Problem Scope

The current KB structure of Disciple (Tecuci, 1998b) is in a representation that is very much similar to the First Order Predicate Calculus and is a subset of the representation used by OKBC servers for the ontologies. Due to time limitation a step-down version of this problem was chosen that involves the validation and verification of KBs represented in Propositional Calculus.

1.4 Thesis in Outline

Chapter 2 presents a solution to the identified problem. A review of the methodology used, follows in chapter 3. A review of Petri net theory is presented in appendix A.

CHAPTER II

SOLUTION TO THE PROBLEM

The problem is handled by the implementation of a tool based on the methodology given by Zaidi (1994) for the validation and verification of decision-making rules. The methodology presented by Zaidi (1994) exploits the structural properties of Petri nets for the detection of errors in the KB. It is based on transforming the rules in a KB to an equivalent Petri net (PN) representation and then applying the well-established analytical tools of the Petri net theory for the detection of errors.

The tool named 'RuleValidator' is implemented in C++. The General architecture of RuleValidator is presented in figure 2.1. The four components of RuleValidator are shown by solid rectangular boxes in fig. 2.1. The whole execution cycle of RuleValidator can be divided into four overlapping phases, as shown in figure 2.1. An overview of the functionality of each component in these phases is described in the following sections. A flow chart of the whole execution cycle at an abstract level is presented in section 2.5, which gives an overall idea of RuleValidator.

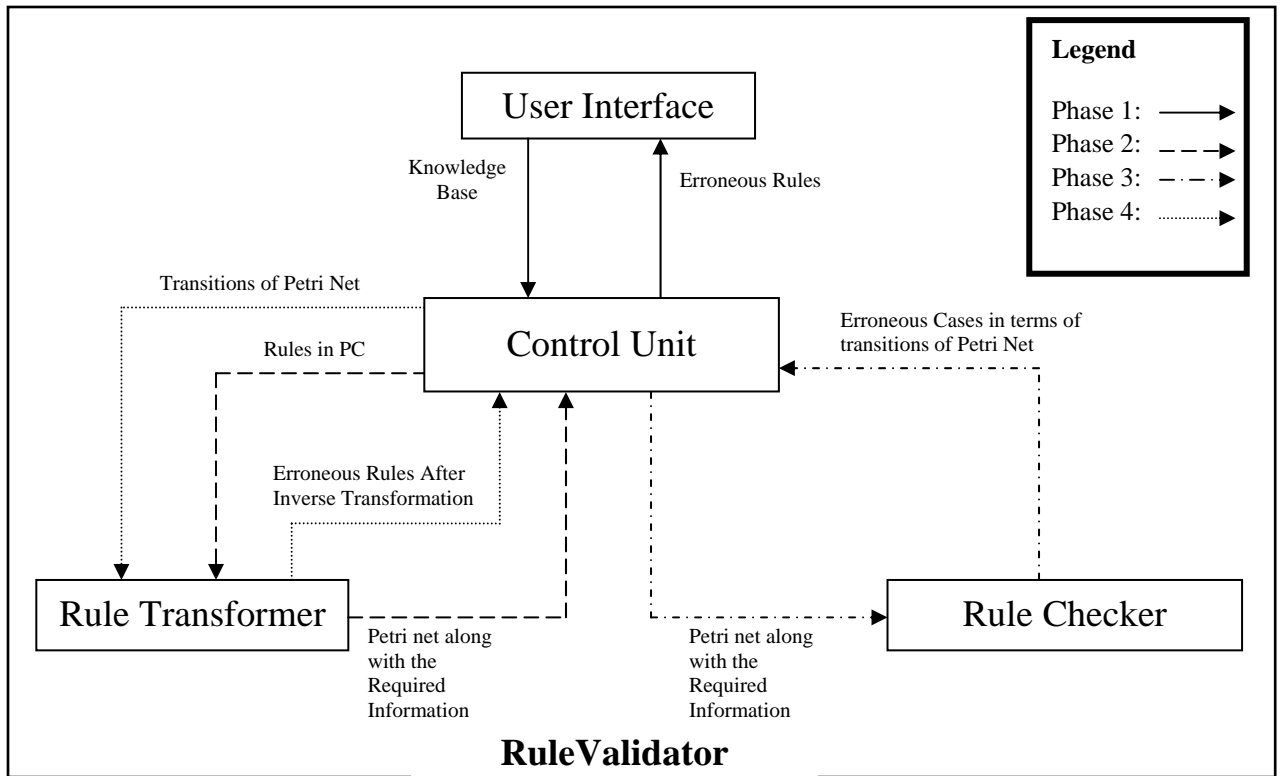


Figure 2.1: General Architecture of RuleValidator

2.1 User Interface

The User Interface is responsible for the following functions during phase 1 of the execution cycle of RuleValidator.

Phase 1:

1. Inputting the KB that the user wants to verify and validate, along with the required information.
2. Outputting the errors.
3. Transferring the KB to the control unit.
4. Receiving the results of the validation and verification of the KB from the control unit.

2.2 Control Unit

This component is responsible for the overall cooperation between the different components during phases 2,3, and 4.

Phase 2:

1. Transferring rules expressed in Propositional Calculus (PC) along with the required information to Rule Transformer.
2. Receiving the equivalent Petri Net representation along with the required information, mapped to the places or transitions of Petri net, from Rule Transformer

Phase 3:

1. Transferring the information received at the end of Phase 2 to Rule Checker.

2. Receiving the problematic rules in the form of transitions of Petri net from Rule Checker.

Phase 4:

1. Transferring transitions of Petri net to Rule Transformer.
2. Receiving rules in PC from Rule Transformer.

2.3 Rule Transformer

The Rule Transformer is responsible for the following functions during phase 2 and phase 4.

Phase 2:

1. Parsing rules in PC.
2. Transforming rules in PC to Petri net if they have no direct cycles.

Phase 4:

1. Transforming transitions of Petri net to rules in PC.

2.4 Rule Checker

The Rule Checker is responsible for the following functions during phase 3.

1. By performing static and dynamic analysis on the Petri net, it identifies the problematic cases discussed in chapter 1.
2. Once the erroneous rules are identified Rule Checker returns them to the control unit.

2.5 RuleValidator Flow Chart

Figure 2.2 presents the flow chart of RuleValidator's working at an abstract level of

execution.

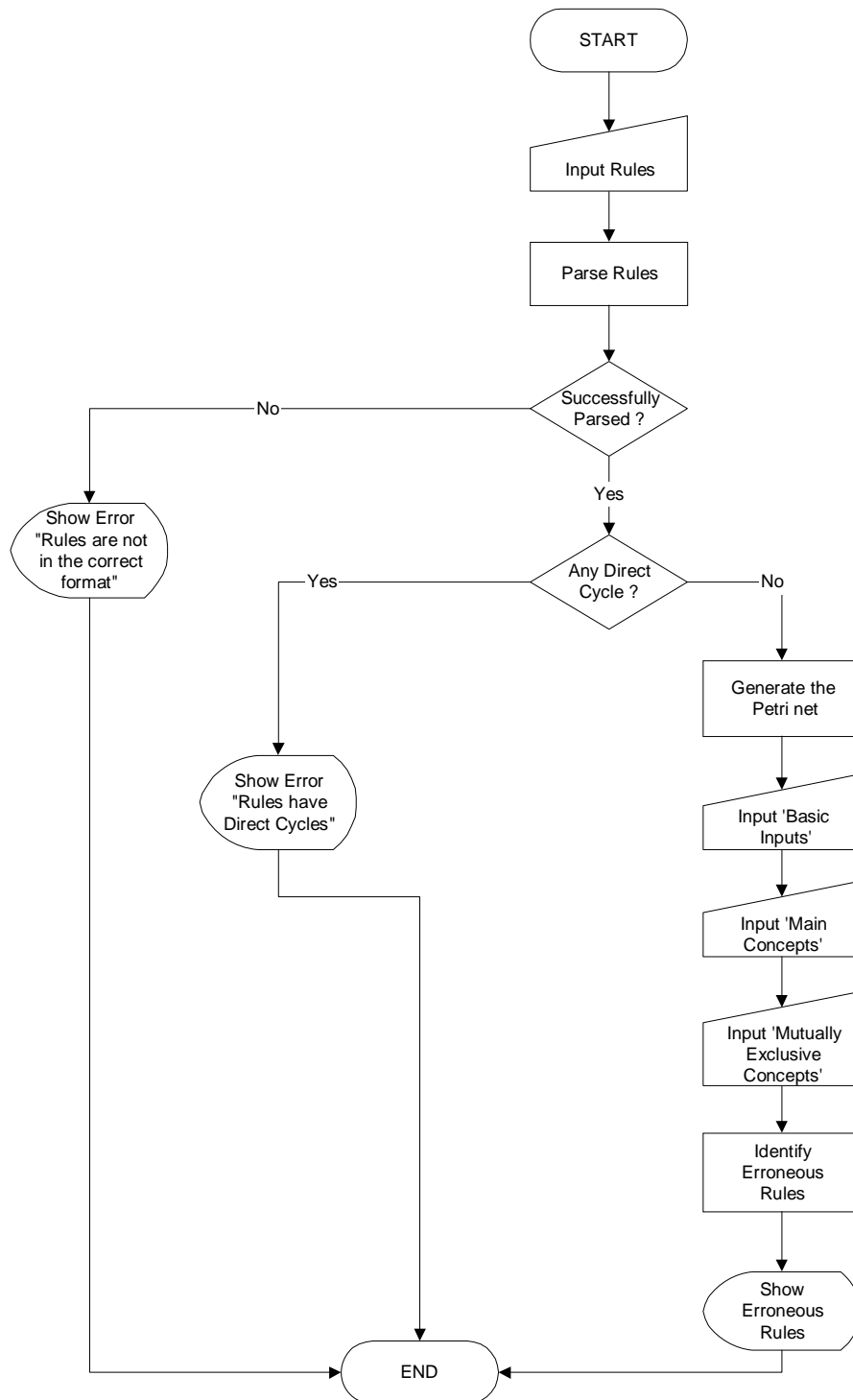


Figure 2.2: Flow chart of RuleValidator

2.6 Closure

A solution to the validation and verification of KBs of intelligent agents is presented. The solution has been successfully implemented for KBs represented in Propositional Calculus and in future will be extended to also address the validation and verification of KBs represented in First Order Predicate Calculus.

CHAPTER III

VALIDATION AND VERIFICATION OF DECISION MAKING RULES

This chapter presents an overview (for details, see Zaidi, 1994) of the methodology used, to implement RuleValidator. The methodology was presented by Zaidi (1994) for the validation and verification of decision-making rules in an organization. It exploits the structural and dynamic properties of Petri nets for the detection of errors in the KB. It is based on transforming the rules in a KB to an equivalent Petri net (PN) (See Appendix A for an introduction to Petri nets) representation and then applying the well-established analytical tools of the Petri net theory for the detection of errors.

Section 1, discusses the KB structure requirement for the application of the methodology. The types of problems that are addressed are presented in section 2, followed by the Petri net representation of rules and the algorithms for the detection of errors in the Petri net.

Through out the text, the symbols ‘ \wedge ’, ‘ \sim ’, and ‘ \rightarrow ’ are used for *Conjunction*, *negation*, and *Implication* respectively

3.1 Knowledge Base Structure Requirement

3.1.1 Structure of Rules

The methodology only considers the rules in a KB and does not consider the facts about the domain of discourse. All rules are represented as the statements of formal logic having the following structure.

$$p1 \wedge p2 \wedge p3 \wedge \dots \wedge pn \rightarrow q$$

Where, $p1$, $p2$, $p3$, pn , and q are literals (Propositions).

The above structure is also called as Horn Clause.

3.1.2 Basic Inputs (U) / Main Concepts (ψ)

For every KB there are some concepts that are characterized as the Basic Inputs (U) to the rule base. In our case these are the inputs that the user will provide to the Disciple Agent and on the basis of which Disciple Agent will produce some output concepts these will be the Main Concepts (ψ).

The application of the considered methodology requires the presence of Basic Inputs and Main Concepts of the rule base.

3.1.3 Mutually Exclusive Concepts (μ)

Concepts can be categorized as Mutually Exclusive Concepts (μ) i.e. the set of concepts that both cannot be true at the same time.

$$\mu = \{p1, \sim p1\} \quad \text{e.g.}:$$

In the above example the two concepts are syntactically mutually exclusive. However the methodology also provides the support for semantically defined mutually exclusive concepts.

3.1.4 Absence of Direct Cycles

The application of the methodology requires the absence of rules with direct cycles.

$$p1 \wedge p2 \rightarrow p1 \quad \text{e.g.}:$$

This type of rules must be identified before the transformation of rules.

3.2 Types of Problems

The methodology presented in zaidi (1994; 1997), addresses the following five types of problems in a rule base.

3.2.1 Incompleteness

Definition 3.1: Incompleteness (Levesque, 1984). “A knowledge base is *incomplete* when it does not have the information necessary to answer a question (*appropriately*) of interest to the system.”

Following three types of incompleteness are considered in zaidi (1994).

3.2.1.1 Rules with ambiguous conditions

Suppose there exists a rule ‘Rule 1’,

$$\text{Rule 1: } A \wedge p1 \rightarrow R$$

If, at least one of the concept in premise cannot be defined in terms of basic concepts and there exist no such rule, such that

$p2 \rightarrow A$ and

$p2 \in U$

Then, 'Rule 1' can be characterized as an ambiguous rule.

3.2.1.2 Rules with useless conclusions

Suppose there exist a rule 'Rule 2',

Rule 2: $p1 \wedge p2 \rightarrow R$

If the conclusion is not the main concept and there is no rule going from the conclusion to a main concept.

i.e., R is not the main concept and there is no such rule

$R \rightarrow \alpha$ and

$\alpha \in \psi$

Then, 'Rule 2' can be characterized as a useless rule.

3.2.1.3 Isolated Rules

A rule is an isolated rule if and only if all the propositions in its premise are complex concepts that cannot be explained in terms of the basic concepts, U ; and its conclusion R is not the main concept, nor does there exist a rule (or rules) taking R to an assertion α , where $\alpha \in \psi$.

3.2.2 Circular Rules

Following types of rules are considered Circular.

e.g., *Rule 3:* $p1 \rightarrow p2$

Rule 4: $p2 \rightarrow p3$

Rule 5: $p3 \rightarrow p1$

In the above case the set of rules will result in an infinite execution.

3.2.3 Inconsistent (Conflicting) Rules

Definition 3.2: Consistency (Zhang & Nquyen, 1989) “A rule base is defined to be consistent if and only if there is no way of reaching contradictory assertions from valid input data.”

The following three cases of inconsistent rules are considered in (Zaidi, 1994).

3.2.3.1 Direct contradiction

Consider the following set of rules,

Rule 6: $q1 \wedge q2 \rightarrow p1$

Rule 7: $q3 \wedge q4 \rightarrow p2$

Rule 8: $p1 \wedge p2 \rightarrow \sim q1$

The above rules are inconsistent because the presence of $q1$ implies the presence of $\sim q1$ and we know that both cannot be true at the same time because both are mutually exclusive.

3.2.3.2 Contradiction in input

Consider the following set of rules,

Rule 9: $p1 \wedge p2 \rightarrow p3$

Rule 10: $p3 \wedge \sim p1 \rightarrow A$

The above rules are inconsistent because, in order for A to be true, $p1$ and $\sim p1$ must be true, hence requiring the presence of two mutually exclusive concepts.

3.2.3.3 *Contradiction in conclusion*

Consider the following set of rules,

$$\text{Rule 11: } p1 \wedge p2 \rightarrow p3$$

$$\text{Rule 12: } p1 \wedge p2 \rightarrow \sim p3$$

The above rules are inconsistent because their antecedents are the same but their conclusions are mutually exclusive concepts.

3.2.4 **Redundant Rules**

Redundancy in a rule base refers to the presence of multiple copies of the same rule or the presence of sets of rules that have the same effect (output) when initiated.

Following types of rules are considered Redundant.

e.g., $\text{Rule 13: } p1 \wedge p2 \wedge p3 \wedge p4 \rightarrow A$

$$\text{Rule 14: } p1 \wedge p2 \rightarrow q1$$

$$\text{Rule 15: } p3 \wedge p4 \rightarrow q2$$

$$\text{Rule 16: } q1 \wedge q2 \rightarrow A$$

3.2.5 **Subsumed Rules**

A rule base may have two rules with identical conclusions where the antecedent conditions of one rule are a subset of the antecedent conditions of another. The first rule is said to subsume the second.

e.g., *Rule 17: $p_1 \rightarrow A$*

Rule 18: $p_1 \wedge p_2 \rightarrow A$

Here the first rule subsumes the second.

3.3 Petri net Representation of Rules

An individual rule of the form ' $P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow Q$ ' is transformed to a Petri net (PN), with a single transition having n input places, each representing a single input proposition P_i , and an output place representing the assertion Q (Zaidi, 1997). The labels of the places and transitions correspond respectively to the propositions and the rules they represent. In this way m rules will be transformed into m Petri nets.

The next step is to unify these individual PNs by combining all the places having common label, and introducing two virtual places, P_{in} and P_{out} , and two virtual transitions T_{in} and T_{out} . Such that,

$$TPreset(T_{in}) = P_{in},$$

$$TPostset(T_{in}) = \{x \mid x \in U\}$$

And

$$TPreset(T_{out}) = \{y \mid y \in \psi\}$$

$$TPostset(T_{out}) = P_{out}.$$

Where, $TPreset$ and $TPostset$ are functions that return the input places and output places of a transition, respectively.

Example

Consider the following set of rules,

Rule 1: $P1 \wedge P2 \rightarrow P3$

Rule 2: $P2 \wedge P4 \rightarrow P5$

Rule 3: $P5 \wedge P3 \rightarrow P6$

The above rules can be converted into the Petri net shown in fig. 3.1.

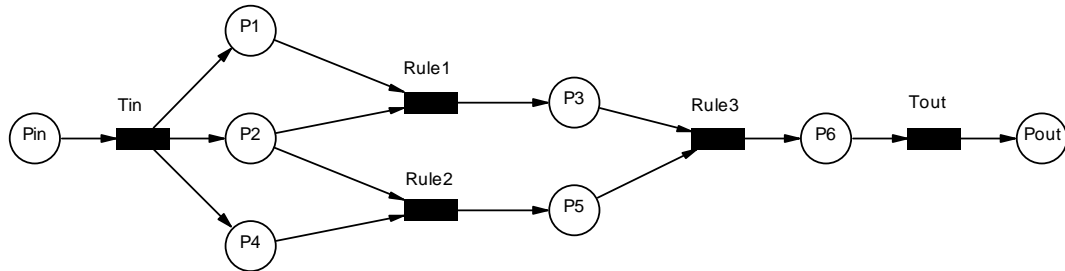


Figure 3.1: Petri net representation of rules

3.4 Incidence Matrix representation of Petri nets. (Zaidi, 1994)

“A Petri net with n places and m transitions can be represented by an $n \times m$ matrix C , the Incidence Matrix. The rows correspond to places, the columns correspond to transitions, and,

- $C_{ij} = 1$, if there is a directed arc from the j -th transition to the i -th place. “1” indicates that the firing of the j -th transition adds one token to the i -th place.
- $C_{ij} = -1$, if there is a directed arc from the i -th place to the j -th transition. “1” indicates that the firing of the j -th transition removes one token from the i -th place.
- $C_{ij} = 0$, if there is no arc from the j -th transition to the i -th place, or from i -th place to the j -th transition”

This representation is only feasible for pure Petri nets, i.e. Petri nets having no self-loop, as the corresponding element cannot be both +1 and -1 at the same time.

3.5 Detection of Problematic Cases

This section presents the algorithms for the detection of problematic cases in the PN representation of rules. It first discusses the detection of Incomplete rules along with the two supporting algorithms. The S-Invariant analysis for the detection of Circular and Inconsistent rules follows. Then we discuss the Occurrence graph analysis for the detection of problematic cases left undiscovered by structural analysis.

Some notations and functions that are used are:

- $x \Rightarrow y$ means, that y is reachable from x .
- $\cdot p$: When applied to the place p , returns a set of transitions PPR.
Where $PPR = \{x \mid x \text{ is a transition, and } x \text{ has a directed arc to } p\}$
- $p \cdot$: When applied to the place p , returns a set of transitions PPO.
Where $PPO = \{x \mid x \text{ is a transition, and } p \text{ has a directed arc to } x\}$
- $\cdot t$: When applied to the transition t , returns a set of places TPR.
Where $TPR = \{y \mid y \text{ is a place, and } y \text{ has a directed arc to } t\}$
- $t \cdot$: When applied to the transition t , returns a set of places TPO.
Where $TPO = \{y \mid y \text{ is a place, and } t \text{ has a directed arc to } y\}$
- **T(S)**: When applied to the set S , consisting of nodes (places and transitions) of a PN, returns the transitions that are in the set S .

- **P (S):** When applied to the set S, consisting of nodes (places and transitions) of a PN, returns the places that are in the set S.

3.5.1 Detection of Incomplete Rules

In order to support the detection of Incomplete rules, following two algorithms are used.

(The original net is denoted by $PN = (P, T, I, O)$)

3.5.1.1 FindPath-to-Sources (FPSO) Algorithm

The FPSO algorithm (Zaidi, 1994), when applied to a node p , collects all nodes (places or transitions) from where there is a path to node p . Formally it can be defined as follows.

For a $PN = (P, T, I, O)$

$$p \in V (= P \cup T)$$

$$FPSO(p) = (S, PN')$$

Where,

$$S = \{x \mid x \Rightarrow p\}$$

$$PN' = (P', T', I', O')$$

$$PN \supseteq PN'$$

$$P \supseteq P'$$

$$T \supseteq T'$$

$$I \supseteq I'$$

$$O \supseteq O', \text{ and}$$

$$P' \cup T' = S$$

3.5.1.2 FindPath-to-Sinks (FPSI) Algorithm

The FPSI algorithm (Zaidi, 1994), when applied to a node p , collects all nodes (places or transitions) that are reachable from node p . Formally it can be defined as follows.

For a $PN = (P, T, I, O)$

$$p \in V (= P \cup T)$$

$$FPSI(p) = (S, PN')$$

Where,

$$S = \{x \mid p \Rightarrow x\}$$

$$PN' = (P', T', I', O')$$

$$PN \supseteq PN'$$

$$P \supseteq P'$$

$$T \supseteq T'$$

$$I \supseteq I'$$

$$O \supseteq O', \text{ and}$$

$$P' \cup T' = S$$

3.5.1.3 Detection of Ambiguous Rules

In a Petri net representation of a rule base, an ambiguous rule can be identified by the presence of following structure.

Proposition 3.1 Ambiguous Rules (Zaidi, 1994)

‘A decision rule represented by a transition R is called *Ambiguous* if and only if

$\exists p, p \in \bullet R, p \notin U$, and there does not exist a node (place) q , such that $q \in U$ and $q \Rightarrow p$.’

An algorithm that is based on the above proposition, for identifying the ambiguous rules, is presented in algo. 3.1.

Algorithm 3.1: *Detection of Ambiguous rules*

- Apply *FPSI* to *Pin*; $FPSI(Pin) = (S1, PNI)$,
- Calculate $AP = P - \mathbf{P}(S1)$,
- $\forall p, p \in AP$ if $p \notin \psi$, then $p \cdot (\neq \Phi)$ identifies the Ambiguous rules.

3.5.1.4 *Detection of Useless rules*

In a Petri net representation of a rule base, a *Useless* rule can be identified by the presence of following structure.

Proposition 3.2 Useless Rules (Zaidi, 1994)

‘A decision rule represented by a transition R is called *Useless* if and only if

$\exists p, p \in R \cdot, p \notin \psi$, and there does not exist a node (place) q , such that $q \in \psi$ and $p \Rightarrow q$.’

An algorithm that is based on the above proposition, for identifying the useless rules, is presented in algo. 3.2.

Algorithm 3.2: *Detection of Useless rules*

- Apply *FPSO* to *Pout*; $FPSO(Pout) = (S2, PN2)$,
- Calculate $UP = P - \mathbf{P}(S2)$,
- $\forall p, p \in UP$ if $p \notin U$ then $\cdot p (\neq \Phi)$ identifies the Useless rules.

3.5.1.5 *Detection of Isolated Rules*

In a Petri net representation of a rule base, an *Isolated* rule can be identified by the presence of following structure.

Proposition 3.2 Isolated Rules (Zaidi, 1994)

‘A decision rule represented by a transition R is called *Isolated* if and only if

$\forall p, p \in \cdot R, p \notin U$, and there does not exist a node (place) q , such that $q \in U$ and $q \Rightarrow p$.’

and

$\forall p, p \in R\cdot, p \notin \psi$, and there does not exist a node (place) q , such that $q \in \psi$ and $p \Rightarrow q$.’

An algorithm that is based on the above proposition for identifying the isolated rules is presented in algorithm 3.3.

Algorithm 3.3: *Detection of Isolated rules*

- Calculate $AR = T - \mathbf{T}(S1)$, and $UR = T - \mathbf{T}(S2)$,
- $AR \cap UR$ identifies the Isolated rules.

3.5.2 S-Invariant Analysis (Zaidi, 1997)

The S-invariant analysis looks at all the directed paths in the PN and searches them for problematic cases. The analysis is shown to reveal certain patterns of PNs that correspond to circular and inconsistent rules. Before a detailed description of the analysis is presented, the following definitions are in order.

Definition 3.3: S-invariant. Given an incidence matrix C of a PN, an S-invariant is a $n \times 1$ non-negative integer vector X of the kernel of C^T , i.e.

$$C^T X = 0$$

Definition 3.4: Support of S-invariant. If X is an S-invariant, the set of places whose corresponding components in X are strictly positive is the *support* of the invariant, denoted by $\langle X \rangle$.

The support of an S-invariant is said to be *minimal* if and only if it does not contain the support of another S-invariant but itself and the empty set.

Definition 3.5: S-component. The *S-component* associated with an S-invariant X of a Petri Net is the subnet whose places are the places of $\langle X \rangle$ and whose transitions are the input and output transitions of the places of $\langle X \rangle$.

By extension, a *minimal S-component* is the S-component of a minimal support S-invariant.

Definition 3.6: Marked Graph. A *marked graph* is a connected Petri net in which each place has exactly one input and one output transition.

The minimal S-invariants of the PN are calculated after all dangling nodes found during the detection of incompleteness are removed and the PN is transformed to an equivalent marked graph representation. The algorithm to convert a PN into a marked graph Petri net (MPN) is given in algo. 3.4.

Algorithm 3.4: *Conversion of a Petri net to a Marked graph Petri net*

- Merge the virtual input and output places, P_{in} and P_{out} , into a single external place P_e .
- $\forall p \bullet$ so that $|\bullet p| = m (> 1)$ and $|p \bullet| = n (> 1)$, create $m \times n$ copies of p , denoted by p^i , where $i = 1, 2, \dots, m \times n$. Create n links from each of m transitions, $t_i \in \bullet p$, to all of

the n transitions, $t_j \in p^\bullet$, through these copies of the place p . The process will create $m \times n$ links between the input and output transitions of place p .

A place p is said to constitute a link from a transition t_i to another transition t_j if $p \in \bullet t_i$ and $p \in \bullet t_j$. The process of converting a PN to a MPN preserves the connectivity of the original PN in the sense that if two nodes are directly connected in PN then they will remain connected in the MPN.

Once the PN is converted to an MPN and all minimal supports S-invariants are calculated the next step is to analyze these S-invariants for the presence of specific patterns that can lead to circularity and inconsistency in the rule base. The following subsections discuss the detection of circular and inconsistent rules.

Some notations that are used to represent copies of places that might be generated during the conversion of PN to MPN:

- $p^{(i)}$, i -th copy of place p
- $p^{(\cdot)}$, any copy of place p .

3.5.2.1 *Detection of Circular Rules*

The minimal S-components of the calculated S-invariants are all the directed elementary circuits present in the MPN. The following proposition characterizes the two different types of circuits in MPN.

Proposition 3.3

Let $\langle Xi \rangle$ be the minimal support of the calculated S-invariant Xi ;

- If $\langle Xi \rangle$ contains the external place P_e , then the S-component associated with $\langle Xi \rangle$ represents a directed path from a basic input to one of the main concepts.
- If $\langle Xi \rangle$ does not contain the external place P_e , then the S-component associated with $\langle Xi \rangle$ represents a loop (directed elementary circuit) inside the Petri net structure (rule base).

Using the above proposition all circular rules can be identified in the rule base.

3.5.2.2 Detection of Inconsistent rules (Direct Contradictory rules)

The following proposition characterizes the Direct Contradictory rules.

Proposition 3.4

If, in a PN representation of a rule base, there exists a directed path, $p \Rightarrow q$ between places p and q , where $p, q \in \mu$, the set of mutually exclusive concepts, then the set of transitions IR1, with the following definition, identifies the inconsistent rules;

$$IR1 = \{ t \mid t \in T, p \Rightarrow t \text{ and } t \Rightarrow q \}$$

Using the above proposition the Direct Contradictory rules can be identified by the following propositions.

Proposition 3.5

If there exists a $\langle X_i \rangle$ which contains places $p^{(i)}$ and $q^{(i)}$, where $p, q \in \mu$, then $\langle X_i \rangle$ indicates the presence of direct contradictory rules in the rule base. The transitions in the directed path $p \Rightarrow q$ of the S-component associated with $\langle X_i \rangle$ identify the set of direct contradictory rules IR1.

3.5.2.3 Detection of rules having Contradiction in Input

The following proposition characterizes the rules that require contradictory inputs to be present in order to be fired.

Proposition 3.6

If, in a PN representation of a rule base, there exist two directed paths, $p \Rightarrow t$ and $q \Rightarrow t$, where $p, q \in \mu$, the set of mutually exclusive concepts, $t \in T$, and the two directed paths do not share a place node, then the rule represented by t is an inconsistent rule that requires contradictory assertions to be true in the input.

Using the above proposition the Inconsistent rules can be identified by the following propositions.

Proposition 3.7

If there exist $\langle X_i \rangle$ and $\langle X_j \rangle$, where $p^{(i)} \in \langle X_i \rangle$ and $q^{(j)} \in \langle X_j \rangle$, $p, q \in \mu$, and $\langle X_i \rangle \cap \langle X_j \rangle \neq \Phi$, then $\langle X_i \rangle$ and $\langle X_j \rangle$ indicate the presence of possible inconsistent rules in

the rule base. The S-components associated with $\langle X_i \rangle$ and $\langle X_j \rangle$ identify the set of inconsistent rules (Proposition 3.8)

Proposition 3.8

In the S-components associated with $\langle X_i \rangle$ and $\langle X_j \rangle$, if $\exists t, t \in T$, such that $p^{(i)} \Rightarrow t, q^{(j)} \Rightarrow t$ and there does not exist a place r such that $p^{(i)} \Rightarrow r^{(i)}, q^{(j)} \Rightarrow r^{(j)}$, and $r^{(i)} \Rightarrow t, r^{(j)} \Rightarrow t$, then the transitions in the directed paths $p^{(i)} \Rightarrow t$ and $q^{(j)} \Rightarrow t$ identify the inconsistent rules.

3.5.3 Dynamic Analysis (Zaidi, 1997)

The dynamic analysis of the PN is performed to explore the behavioral properties of the graph. The specific technique presented here is the occurrence graph analysis. The analysis considers all the possible states of the net that can be reached from an initial state (marking) after a finite number of firings of transitions in the net.

Definition 3.7: Marking (state). A marking (state) of a Petri net is a mapping $M: P \rightarrow \{0, 1, 2, \dots\}$ that assigns a non-negative integer (the number of tokens) to each place.

Definition 3.8: Occurrence graph. An occurrence graph associated with a Petri net and an initial marking M^0 represents all possible reachable markings from M^0 . Each node of occurrence of the occurrence graph represents a reachable marking, and the arcs between nodes represent transitions from one marking to another (with arc annotations indicating the transition name).

For a given input, the analysis can generate an occurrence graph for the PN, and the occurrence graph can be searched for states that can be reached from more than one distinct paths in the OG – *redundant cases* – and for conflicting states reachable from single input – *conflicting cases*. But, with no knowledge of the feasible input domain, one is left with 2^n possible combinations of input vectors ($n = |U|$). The construction of these many OGs is simply too large a problem. An approach is presented to overcome this problem.

The redundancies present in a rule base appear in the PN representation as places with multiple inputs representing the redundancy in arriving at a conclusion, and with places with multiple outputs representing several paths originating from the same input. This heuristic is used to determine whether or not the PN representing the rule base should be analyzed by the OG analysis for the redundant cases.

The heuristic can also be used to narrow down the search for redundancies to only those parts of the PN with such places. The following steps, performed on the PN, extracted the problematic parts of the original PN.

- For the PN construct a set **A**: $\mathbf{A} = \{p \mid p \in \text{set of places in PN and } |\cdot p| > 1\}$.
- From **A** select a place p_i (starting from the elements of the set of main concepts ψ), and apply $FPSO(p_i)$. The result is denoted by \mathbf{PN}_i .
- If \mathbf{PN}_i contains places with multiple output arcs, put \mathbf{PN}_i in a set **RN**. Otherwise, declare \mathbf{PN}_i free of redundant cases.

- Remove from \mathbf{A} all the places that are present in \mathbf{PN}_i . Repeat the process as long as there are elements in \mathbf{A} .

The only thing remaining is the determination of initial markings before any OGs can be constructed and searched for states that correspond to problematic cases. The solution to this problem reverses the direction of all the arcs in each \mathbf{PN}_i . The idea is to scan the feasible input space from a known and much smaller set of outputs. The resulting net is denoted by \mathbf{PN}_i' and defined as $\mathbf{PN}_i' = (\mathbf{P}, \mathbf{T}, \mathbf{O}, \mathbf{I})$, where \mathbf{PN}_i is given by the tuple $(\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O})$. The modified net is then initialized by putting a single token in the place p_i (note that p_i is the place on which the **FPSO** algorithm was applied that resulted in the determination of \mathbf{PN}_i). Since all the nets in the **RN** are **Y**-nets, every transition in the \mathbf{PN}_i' will have only one input; a single token in place p_i enough to ensure that every place in the \mathbf{PN}_i' will appear as marked at least once in the occurrence graph constructed for \mathbf{PN}_i' .

Definition 3.9: Y-nets. A Petri net $(\mathbf{P}, \mathbf{T}, \mathbf{I}, \mathbf{O})$ is a *Y-net* if $\forall t \in \mathbf{T}, |t \bullet| = 1$.

3.5.3.1 Detection of Redundant rules.

Proposition 3.9

Let $\mathbf{PN}_i \in \mathbf{RN}$. Distinct firing vectors corresponding to multiple paths between two nodes of \mathbf{OG}_i , or paths from one root node to two or more nodes having the same set (or subset)

of places as marked, identify redundant rules. The transitions whose corresponding components in the firing vectors are positive represent the rules that are redundant.

Algorithm 3.5: *Occurrence graph construction for redundant rules.*

The algorithm proceeds by reversing the arrows of the nets \mathbf{PN}_i in the set \mathbf{RN} and initializing the modified nets, termed \mathbf{PN}_i' , by putting a token in source places. The occurrence graphs \mathbf{OG}_i so obtained are searched for redundancies.

3.5.3.2 *Detection of Conflicting rules.*

The case in which conflicting assertions can be made through a single input can be detected by a methodology similar to the one used for the redundancies.

Proposition 3.10

Let $\mathbf{PN}_i \in \mathbf{RN}$ and let there not exist any redundant case inside the decision rules represented by \mathbf{PN}_i . Then application of the techniques used for redundancy and results from proposition 3.9 on Occurrence graph \mathbf{OG}_i identifies conflicting rules.

Algorithm 3.6: *Occurrence graph construction for conflicting rules.*

- For a set of mutually exclusive concepts μ_i present in the rule base, merge all the places in μ_i into a single virtual place p_i .
- Apply $FPSO(p_i)$. The result is denoted by \mathbf{PN}_i .

- If \mathbf{PN}_i contains places with multiple output arcs, put \mathbf{PN}_i in a set \mathbf{CN} . Otherwise, declare \mathbf{PN}_i free of conflicting rules.
- Repeat for all sets of mutually exclusive concepts defined for the system in hand, except for those sets whose elements have already been identified in the conflicting rules reported by the static analyses.

3.6 Closure

A review of the validation and verification methodology that is implemented in this project is presented.

CHAPTER IV

CONCLUSION

Nowadays when intelligent agents are heavily used in sensitive areas such as war, manufacturing industries, etc., any error in the KB of the agent could lead to heavy losses. Hence a need for the validation and verification of KBs of intelligent agents naturally arises, but unfortunately often ignored, that results in anomalies during run-time.

An approach for the integration of an existing V & V technique with intelligent agents is presented that result in the development of error-free intelligent agents. The tool RuleValidator based on a V & V technique, presented by Zaidi (1994) has been successfully implemented.

REFERENCES

- Aho A. V., Sethi R. and Ullman J. D. (1988), "Compilers, Principles, Techniques, and Tools," Addison-Wesley Publishing Company, Reading, MA.
- Chaudhri, V. K., Farquhar, A., Fikes, R., Karp, P., and Rice, J. (1997), "Open Knowledge Base Connectivity 2.0." Technical Report, Artificial Intelligence Center of SRI International and Knowledge Systems Laboratory of Stanford University.
- Galton A. (1990), "Logic for Information Technology," John-Wiley and Sons Ltd., Chichester, England.
- Levesque, H. J. (1984), "The logic of incomplete knowledge Bases", On Conceptual Modeling: Perspective from Artificial Intelligence, Databases and Programming Languages, pp. 165-189. Springer-Verlag, New York.
- Levis A. H. (1998), "Introduction to Petri nets", <http://viking.gmu.edu/>
- Lipschutz S. (1986), "Theory and Problems of Data Structures," McGraw-Hill Book Company, New York.
- Martin J. C. (1996), "Introduction to Languages and the Theory of Computation", The McGraw-hill Companies, Inc., New York.
- Murata T. (1989), "Petri Nets: Properties, Analysis and Application," Proceedings IEEE vol. 77, no. 4, pp. 541-579.
- Tecuci G. (1998a), "Building Intelligent Agents: An Apprenticeship Multistrategy Learning Theory, Methodology, Tool and Case Studies," Academic Press, London.
- Tecuci G. (1998b), "Disciple-98 Knowledge Model, Data Structure and Research Issues", Technical Report, Learning Agents Laboratory, George Mason University, VA

Tecuci G. 2. Tecuci G., Wright K., Lee S. W., Boicu, Bowman M., (1999), "A Learning Agent Shell and Methodology for Developing Intelligent Agents."

Zaidi A. K. (1994) "Validation and Verification of Decision Making Rules". Technical Report GMU/C31-155-TH, C3I Center, George Mason University, Fairfax, VA.

Zaidi A. K. and Levis A. H. (1995), "Validation and Verification of Decision Making Rules", Automatica, Vol. 33, No. 2, pp 155-169. Elsevier Science Ltd., Great Britain

Zhang, Du and Nguyen D. (1992), "A Tool for Knowledge Base Verification," in Development of Knowledge-Based Shells, World Scientific Publishers, Singapore.

APPENDIX A

AN INTRODUCTION TO PETRI NETS (LEVIS, 1998)

A.1 INTRODUCTION

Petri Nets (PN) are a special type of graph. A graph consists of two types of elements, nodes or vertices and edges, and the manner in which these elements are interconnected.

Formally,

Definition 1: Graph, A graph $G = (V, E, \Phi)$ consists of a nonempty set V called the set of *nodes* of the graph, a set E called the set of *edges* of the graph, and a mapping Φ from the set of edges E to a set of pairs of elements of V .

If the pair of nodes connected by an edge is ordered, then the edge is *directed* and an arrow is placed on the edge indicating the direction. If all the edges of the graph are directed, then the graph itself is called a *directed graph* or *digraph*. Three examples of graphs are shown in Figure A.1. Note that the first one, consisting of only two unconnected nodes, can be considered both as a directed and an undirected graph.

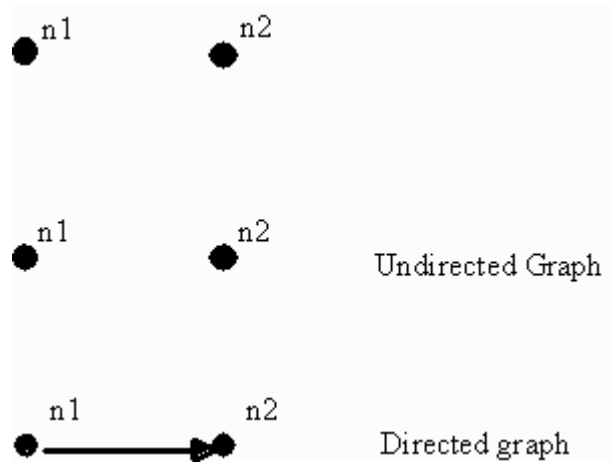


Figure A.1: Examples of graphs

Two nodes that are connected by an edge in a graph are called *adjacent* nodes. Nodes need not be represented by dots only; they can be represented by circles, bars, boxes, or any other convenient symbol for the particular application.

When a graph contains parallel edges, i.e., edges that connect the same pair of nodes and, if directed, have the same direction, then it is called a *multigraph*. Examples of multigraphs are shown in Figure A.2.

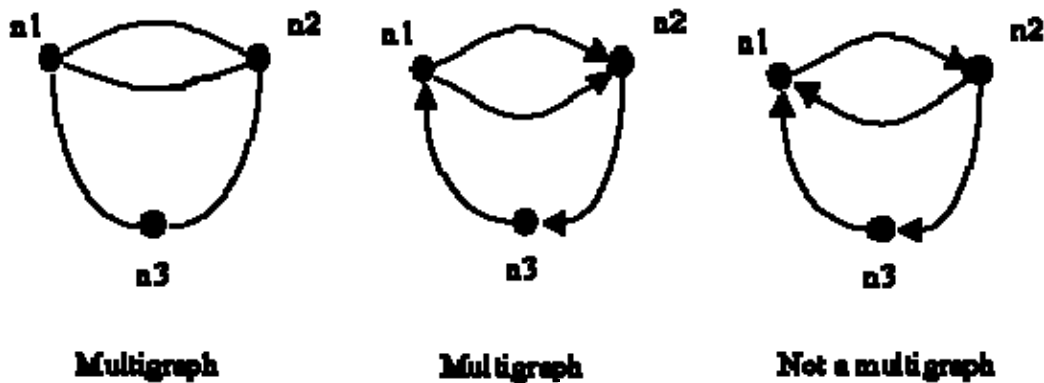


Figure A.2: Examples of multigraphs

While Petri Nets are multigraphs, in this appendix we will consider ordinary Petri Nets only. In many applications, parallel edges are very useful and the multigraph properties of Petri Nets can be used to advantage. However, they introduce notational and other complexities that are best addressed when extensions to Ordinary Petri Nets are considered (e.g., Colored Petri Nets)

A second characteristic of Petri Nets as graphs is that they are *bipartite* graphs. This means that they have *two* types of nodes. Different symbols are used to distinguish the two types of nodes. By convention, the first type of node is called a *place* and is denoted by a circle or ellipse. The second type is called a *transition* and is denoted by a solid bar, or a rectangle. The edges of a Petri Net are called *arcs* and are always *directed*. The symbols are shown in Figure A.3.

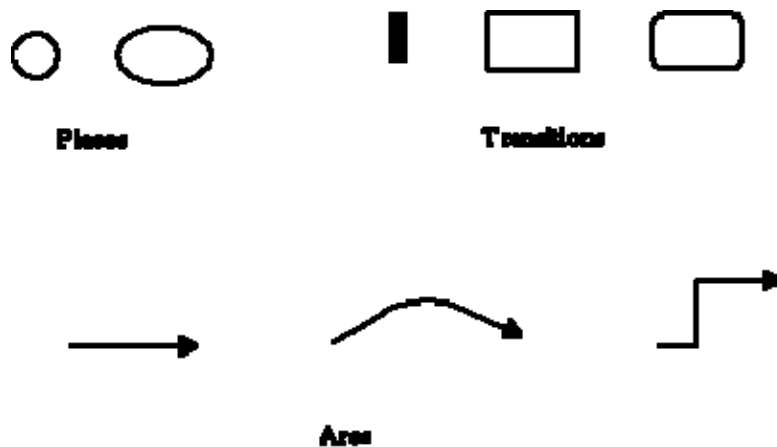


Figure A.3: Places, Transitions, and Arcs

A bipartite graph has a special property: an edge can connect only two nodes that belong to different types. Therefore, there can be an arc from a place to a transition, from a

transition to a place, but *not* from a place to a place or a transition to a transition. Note that this rule is easy to implement in any Petri Net editing software.

Now that Petri Nets have been placed in the context of graph theory, we are ready for some formal definitions.

A.2 ORDINARY PETRI NETS

Definition 2 A **Petri Net** is a *bipartite directed graph* represented by a quadruple

$PN = (P, T, I, O)$ where:

- $P = \{p_1, \dots, p_n\}$ is a finite set of places.
- $T = \{t_1, \dots, t_m\}$ is a finite set of transitions.
- $I(p, t)$ is a mapping $P \times T \rightarrow \{0, 1\}$ corresponding to the set of directed arcs from places to transitions.
- $O(t, p)$ is a mapping $T \times P \rightarrow \{0, 1\}$ corresponding to the set of directed arcs from transitions to places.

The nets, where I and O take the values of 0 or 1, are called ordinary Petri Nets. An example of a Petri Net is shown in Figure A.4; let it be denoted PN_1 . Places are represented by circles and transitions by bars. This is the convention that will be adopted for Ordinary Petri Nets.

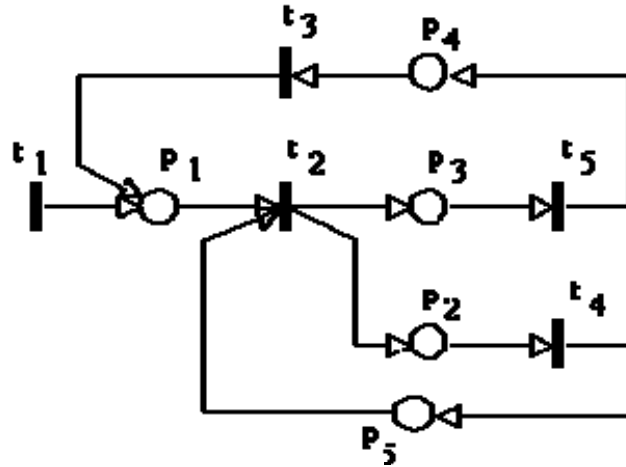


Figure A.4: Petri Net PN1.

The *structure* of this Petri Net, i.e., the quadruple that defines it, can be represented as follows:

$$P = \{p_1, p_2, p_3, p_4, p_5\}$$

$$T = \{t_1, t_2, t_3, t_4, t_5\}$$

$I(p_1, t_1) = 0$	$I(p_2, t_1) = 0$	$I(p_3, t_1) = 0$	$I(p_4, t_1) = 0$	$I(p_5, t_1) = 0$
$I(p_1, t_2) = 1$	$I(p_2, t_2) = 0$	$I(p_3, t_2) = 0$	$I(p_4, t_2) = 0$	$I(p_5, t_2) = 1$
$I(p_1, t_3) = 0$	$I(p_2, t_3) = 0$	$I(p_3, t_3) = 0$	$I(p_4, t_3) = 1$	$I(p_5, t_3) = 0$
$I(p_1, t_4) = 0$	$I(p_2, t_4) = 1$	$I(p_3, t_4) = 0$	$I(p_4, t_4) = 0$	$I(p_5, t_4) = 0$
$I(p_1, t_5) = 0$	$I(p_2, t_5) = 0$	$I(p_3, t_5) = 1$	$I(p_4, t_5) = 0$	$I(p_5, t_5) = 0$
$O(t_1, p_1) = 1$	$O(t_2, p_1) = 0$	$O(t_3, p_1) = 1$	$O(t_4, p_1) = 0$	$O(t_5, p_1) = 0$
$O(t_1, p_2) = 0$	$O(t_2, p_2) = 1$	$O(t_3, p_2) = 0$	$O(t_4, p_2) = 0$	$O(t_5, p_2) = 0$
$O(t_1, p_3) = 0$	$O(t_2, p_3) = 1$	$O(t_3, p_3) = 0$	$O(t_4, p_3) = 0$	$O(t_5, p_3) = 0$
$O(t_1, p_4) = 0$	$O(t_2, p_4) = 0$	$O(t_3, p_4) = 0$	$O(t_4, p_4) = 0$	$O(t_5, p_4) = 1$
$O(t_1, p_5) = 0$	$O(t_2, p_5) = 0$	$O(t_3, p_5) = 0$	$O(t_4, p_5) = 1$	$O(t_5, p_5) = 0$

We denote by $\bullet t$ the set of all input places of transition t and call it the *preset* of t and by $t\bullet$ the set of all output places of transition t and call it the *postset* of t . The same notation applies for places as well: the preset $\bullet p$ denotes the set of all input transitions of

place p , while the postset p^\bullet denotes the set of all output transitions of place p . For the Petri Net PN_1 of Figure 4, the following relations hold:

$$\bullet p_1 = \{t_1, t_3\},$$

$$t_2^\bullet = \{p_2, p_3\}$$

for the preset of p_1 and the postset of t_2 . Note that by defining the set of places, the set of transitions, and the presets and postsets either of all places or of all transitions we obtain an equivalent representation of the structure of a Petri Net. The concepts of preset and postset become very useful in describing algorithms for the analysis of Petri Nets.

Self-loops and Pure Petri Nets

A place p and a transition t are on a *self-loop*, if p is both an input and an output place of t . A Petri Net will be *pure*, if it does not contain self loops.

The Petri Net PN_1 on Fig. A.4 is pure, while the Petri Net PN_2 in Fig. A.5 is not: it contains the self-loop (t_2, p_3) .

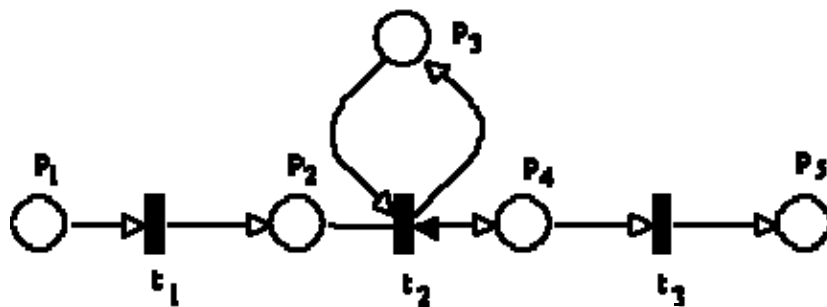


Figure A.5: A Petri Net PN_2 with a self-loop (t_2, p_3)

The following notions are taken directly from graph theory and interpreted in the context of Petri Nets.

Subnet of a Petri Net

A subnet of a Petri Net $PN = (P, T, I, O)$

is a Petri Net $PNs = (Ps, Ts, Is, Os)$ such that:

$$Ps \subseteq P ; Ts \subseteq T$$

and Is and Os are the restrictions of I and O to $Ps \times Ts$ and $Ts \times Ps$, respectively.

The self loop of PN_2 in Figure 5 is one possible subnet :

$$PN_{21} = (P_1, T_1, O_1, I_1)$$

where

$$P_1 = \{p_3\}; T_1 = \{t_2\}$$

$$I(p_3, t_2) = 1 \text{ and } O(t_2, p_3) = 1$$

Paths

A *path* is a set of k nodes and $k-1$ arcs, for some integer k , such that the i -th arc either connects the i -th node to the $i+1$ -th node, or the $i+1$ -th node to the i -th node. The path is directed, if for all $i = 1, 2, \dots, k$, the i -th arc connects the i -th node to the $i+1$ -th node.

A path in which no arc is traversed more than once is called a *simple path*.

A path in which no node is traversed more than once is an *elementary path*.

Clearly, all elementary paths are simple paths, but the converse is not true. In Figure 5 the path $(p_1, t_1, p_2, t_2, p_4, t_3, p_5)$ is an elementary path and a simple path. The path $(p_1, t_1, p_2, t_2, p_3, t_2, p_4, t_3, p_5)$ is a simple path, but not an elementary one, since the node t_2 is traversed twice.

Connectivity

A Petri Net is *connected* if and only if there exists a path - not necessarily directed - from any node to any other node.

Strong Connectivity

A Petri Net is *strongly connected* if and only if there exists a directed path from any node to any other node.

The net of Fig. A.4 , PN₁, is connected but not strongly connected: there is no directed path from p₁ to t₁, for instance. A net with the same nodes and links as PN₁ with the exception of t₁ and its link to p₁ would be strongly connected. The net PN₂ in Fig. A.5 is also connected and, very clearly, not strongly connected. Indeed, both nets have some common properties. In Fig. A.4, transition t₁ has no preset, there are no incoming arcs. In Fig. A.5, place p₁ has no preset, no incoming arcs. When a connected node has only outgoing arcs, we will call it a *source*; when it has only incoming arcs, we will call it a *sink*. (Sometimes these are referred to as start and end nodes.) If a net has even one sink or source, it cannot possibly be strongly connected.

Directed Circuit

A *directed circuit* is a directed path from one node back to itself.

Directed Elementary Circuit

A *directed elementary circuit* is a directed circuit in which no node appears more than once.

The net PN₁ in Fig. A.4 has two directed elementary circuits:

- p₁ - t₂ - p₃ - t₅ - p₄ - t₃ - p₁
- t₂ - p₂ - t₄ - p₅ - t₂

Directed elementary circuits play a key role in the performance analysis of DES models described by Petri Nets. An algebraic characterization of those circuits will be given in section 7 for a specific class of Petri Nets.

A.3 LINEAR ALGEBRAIC APPROACH

In section A.2, Petri Nets were introduced using a graph theoretic approach. Petri Nets can also be described in terms of integer arithmetic. This ability of representing models both graphically and in terms of integer arithmetic makes them particularly attractive as tools for computer aided design, where the user can be interacting graphically with the nets on a monitor (editing) while at the same time he can use a number of algorithms to support the analysis.

Incidence Matrix

The topological structure of a pure Petri Net can be represented by an integer matrix C , called an *incidence* or flow matrix. C is an $n \times m$ matrix whose m columns correspond to the transitions and whose n rows correspond to the places of the net. C is defined as follows:

$$C_{ij} = O(t_j, p_i) - I(p_i, t_j) \text{ for } 1 \leq i \leq n, 1 \leq j \leq m.$$

Note that the definition is restricted to *pure* Petri Nets. There is actually a problem with non-pure Petri Nets in the sense that self-loops cannot be represented in the incidence matrix: a 1 and a -1 cancel each other to yield a zero in the matrix, thus losing track of the existence of the self-loop.

The mappings O and I can be reconstructed from the matrix C in the following trivial way:

$$O(t_j, p_i) = \max \{C_{ij}, 0\}, I(p_i, t_j) = \min \{C_{ij}, 0\}.$$

The incidence matrix of the Petri Net PN₁ is given below.

$$C_1 = \begin{bmatrix} 1 & -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & -1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & -1 & 0 & 1 & 0 \end{bmatrix}$$

For a detailed overview of Petri nets, See Murata (1989)