Modelling Concurrency by tabular expressions

Yuwen Yang

Master of Science
Department of Computing and Software
McMaster University
Hamilton, Ontario
Canada, L8S 4K1

Abstract

Tabular expressions (Parnas et al) [26, 27, 29, 31, 32] are a software specification technique that becomes increasingly popular in software industry. The current state of the technique is restricted to sequential systems only. In this thesis we show how concurrency can be treated in some systematic way in the framework of tabular expressions.

A precise notion of composite global automata (compare [20, 36]) will be defined. The tabular expressions [24, 26, 28] used by Software Engineering Research Group will be slightly extended to deal with the transition function/relation of concurrent automata.

In the thesis, each sequential process is viewed as a Finitely Defined Automaton with Interpreted States [18], and all of the processes in the system are composed to a global finite state automata to model the concurrent system. The thesis starts with a common model of a nondeterministic finite automaton, extends the traditional automata, and associates two sets called synchronization set and competition set, respectively, to each action of the individual processes. Finally, whole processes in the system are composed and the actions dependence of individual process is eliminated for the global action. A simple example of Readers-Writers is given to illustrate this method.

Contents

Abstract **ii**

1 Introduction **1**
- 1.1 Background 1
- 1.2 Purpose 4
- 1.3 Outline 4

2 Tabular Expressions of Functional Documentation **5**
- 2.1 Background of Tabular Expressions 5
- 2.2 Definition of Tabular Expressions 6
  - 2.2.1 Raw Table Skeleton 7
  - 2.2.2 Cell Connection Graph and Medium Table Skeleton 8
  - 2.2.3 Raw and Medium Table Elements 9
  - 2.2.4 Well Done Table Skeleton 10
  - 2.2.5 Tabular Expressions 12
  - 2.2.6 Semantics of Tabular Expressions 12

3 Introduce to Finite State Automata **14**
- 3.1 Finite State Automata Definition 15
  - 3.1.1 Deterministic Finite Automaton (DFA) 15
  - 3.1.2 Nondeterminism Finite Automaton 17
- 3.2 Concurrent composition of automata 18
  - 3.2.1 Parallel Composition 18
  - 3.2.2 General Composition of Finite Automata 19
  - 3.2.3 Asynchronous Automata 20

4 Global Finite Automata **22**
- 4.1 An Approach to the Problem 22
- 4.2 Global State Machines 25
  - 4.2.1 Model for Each Individual Process 25

4.2.2 Model for Composite Process 29
4.3 Tabular Expression 38
4.3.1 Processes' Priority 40
4.3.2 The Equivalent Relation of Two Specification 40
4.3.3 The Properties of Tabular Form of Concurrent System Specification 41

5 An Example of Readers-Writers System 42
5.1 Readers-Writers Problems 42
5.1.1 Finite State Automata for Individual Process 43
5.1.2 Global Finite State Automaton for Readers-Writers System . . 45
5.2 The Completeness [23] of Tabular Expressions 46
5.3 The Invariant of the System 48
5.4 Deadlock Free Specification 50
5.5 Priority Analyses 51

6 Conclusion and Future Work 55
6.1 Contributions 55
6.2 Applications 56
6.3 Limitations 56
6.4 Future Work 57
6.5 Conclusions 57

Bibliography 58

Chapter 1

Introduction

This chapter provides a brief introduction to the background, purpose and the outline of this thesis.

1.1 Background

It is well known in the software community that a software product is of little use unless people can learn to use it and to maintain it. Documentation and specification plays an important role in this purpose. Hence, they are the key part of the total software product, and their development is an important topic in software engineering.

Documentation of a software product is basically generated for two purposes. One is to explain the features of the software and to describe how to use them. This is known as the user documentation since it is designed to be read by the user of the software products.

The other purpose of documentation is to describe the software itself so that the system can be implemented by the programmers and could be modified later in its life cycle. Documentation of this type is known as system documentation and is inherently more technical in nature than the user documentation. This thesis focuses on the documentation of the latter type, which is commonly refered to as software system specification. Software documentation and software specification is used interchangeably in this thesis.

A software specification expresses the functional and/or relational requirements of a piece of software. It states what a software product is intended to achieve (rather than how it should achieve it) and it states the attributes or features the software must possess.

A specification is used for validation and verification purposes; it is validated to assure that it represents the intended use of software and that the software as

specified will meet it objectives; then it forms a basis for developing a design that can be verified against the validated specification. The beauty of a specification lies in deriving a natural, abstract, implementation independent specification that is still precise and complete.

There are a few criteria to judge specifications. First and most important, a software specification is a contract between the specifier and the implementor defining the system to be constructed. It therefore must be clearly and unambiguously understood by both parties.

Secondly, it must be possible to ascertain whether an implementor has fulfilled such a contract; that is, to test whether a specification and an implementation is equivalent and consistent. Hence testability is another criterion for judging a specification.

However, traditional specifications usually include a narrative description of software structure using natural languages, which is inherently ambiguous. One way to avoid the ambiguity inherent in natural languages is to write the software requirement specification (SRS) in a particular SRS languages. The advantage of using SRS is that it is rather easy for both machines and human beings to detect inconsistencies, redundancies, incompleteness, and ambiguities.

It has been described in [12, 13, 24] that because of the vagueness and imprecision of natural languages, they are not suitable for writing precise documentation for software product. Rather, mathematical documentation can improve the consistency, precision and completeness of natural languages documentation.

Because of this reason, Parnas [24] has proposed the principle of functional/relational documentation. It defines the required content of documents in term of mathematical relations. In this scheme [29], each of the documents is associated with certain relations; the software document must contain a representation of these relations. If a document contains enough information to determine whether or not any pair is included in the specified relation, it is complete. No additional information should be included.

The advantages of this formal approach are to [5]:

- improve specifications and designs for which behavioral properties of systems can be precisely stated;
- support for reasoning about systems through formal analysis;
- provide a basis for (potentially) much more powerful support environments than exist today.

In [23, 26, 27, 29], tabular expression, a functional documentation model for documenting software products was proposed. Since then it has been successfully used

in military and civilian applications, such as in U.S Naval Research Laboratory for writing the A-7E document of software requirements, Darlington Nuclear Power for inspecting safety-critical programs, and Bell Labs for writing requirement documents. Its success in these practical projects has proved the power of tabular expressions on documenting software products.

The Software Engineering Research Group (SERG) at McMaster University has made significant contributions in using the tabular expressions to write a functional documentation for software system. However, most of the previous work was utilizing the functional method to reason sequentially about the whole model, without expressing the concurrent behavior of software system module explicitly.

However, in real life, very few of computational systems are sequential. On all levels, from dynamic web pages to bank interactive system, the computational system behaviors are truly concurrent, in the sense that they may be seen as spatially separated activities that accomplish a join task. Also, many such systems do not really mean to be terminated, and hence it makes little sense to discuss their behaviors in term of the traditional simple input-output model. Rather, one is interested in the behavior of often complex patterns of stimuli/response relationships over time.

Hence, during the study of concurrent system, one is forced to take a different view of behavior than the traditional input-output one. One needs a notion of behavior to express aspects of pattern of actions, for which a system is capable of performing. This is the main task to be accomplished in this thesis. The Finitely Defined Automata [18] with Interpreted States serves as a medium to specify the concurrent systems.

The uses of finite state automata significantly reduce the ambiguity of specification. Moreover, the concept of a "finite state automata" is now understood by any engineering/computer science graduate.

In this thesis, the nondeterministic automaton is used to model and reason about each process. The transition function of each process is given as a tabular expressions. Later, all processes involved in the concurrent system are combined into a *global* finite state automaton. The transition function of this global automaton is also described by the tabular form. In this model, the standard semantic of tabular expressions [15] is slightly extended to address concurrent issues.

In recent years, formal specification techniques have been used in both academics and industrial areas. It has been considered as one of the useful tools to specify and document concurrent systems. However, one disadvantage in using particular SRS languages is the length of time required to learn them. Even it is an excellent specification language available, very few people will use it if it requires a great effort to learn or it difficult to learn.

One interesting experiment [2] conducted in SERG has shown that functional documentation method in the tabular form can be learned and used by typical undergraduate engineering students within a short time.

1.2 Purpose

The purpose of this thesis is to develop a method to write a precise mathematical documentation of concurrent software system using tabular expressions. The goal for this specification technique is to lower its ambiguity level and to increase its understandability to computer-oriented people who design and test software.

The general nature of such a method is that a mathematical formalism is utilized to represent and reason the behavior of concurrent computational systems. The purpose of this method is to provide a conceptional understanding of systems and their behavior.

Throughout the thesis variations of automata are used to illustrate ideas, definitions and applications, and products of finite state automata are used to model concurrent systems.

Traditional automata notations as well as SERG's tabular expressions are extended to address the concurrent system properties.

1.3 Outline

Chapter 2 introduces tabular expressions as used by the McMaster Software Engineering Research group at McMaster University.

Chapter 3 introduces finite state automata, which includes deterministic and non-deterministic finite automata as well as composite automata.

In Chapter 4, a simple model for representing concurrent system is introduced, and its semantics is also defined. The choice of this model is then motivated.

Chapter 4 is the most important part of this thesis. First automata are used to model each process and a transition function/relation is specified with the traditional automaton technique. Secondly, the concurrent properties are modeled by combining each process involved. The global state change is described by the global transition function/relation. The semantics of transition is extended from the traditional automaton theory. The approach to document a concurrent system with tabular form is also described.

In Chapter 5 an example, reader and writer, is given to demonstrate and to analyze how to use this approach.

Chapter 6 discusses the contributions of this thesis. The conclusions are made based on this work, and future work in this area is proposed.

The author believes that this thesis is rather a beginning then a conclusion of a research approach. Chapters 1, 2, and 3 present the results already known, and Chapter 4, 5, and 6 are the contributions of this work.

Chapter 2

Tabular Expressions of Functional Documentation

2.1 Background of Tabular Expressions

It has been recognized that most software problems result from erroneous descriptions of the intended behavior. However, mathematical expressions can be used to provide precise, concise, and unambiguous description of software system behavior.

Since conventional expressions are lengthy or deeply nested, they increase the complexity of software document. It was pointed out in Parnas' work [23] that the traditional mathematical notation is not practical to provide a precise mathematical description of computer systems due to its special characteristics, e.g. the range and domain of functions are often tuples whose elements are of distinct types.

Parnas et al [23, 24, 26, 31] advocated the use of a relational model for documenting the intended behavior of programs. In this model, each of the documents is associated with certain relations. Relations are described by giving their characteristic predicate in terms of the values of concrete program variables. To properly specify and document these relations, tabular expressions are used. As a multi-dimensional expression, tabular notations are often easier to read and understand as compared to the equivalent traditional scalar expressions [1]. Furthermore, tabular expressions are more systematic, and can reduce a complex problem to a simple one.

Tabular expressions are means to represent complex relations that are used to specify and document software systems. The structure provided by tables makes it possible to separate discontinuous conditions, to isolate distinct domain and range elements, and to make it unnecessary to continually repeat common sub-expressions[1].

A formal syntax and semantics of tables was initially proposed by Parnas [23],

[1] A term or predicate expressions as defined in [25] will be called a scalar expression.

$f(x,y) =$

	$y = 10$	$y > 10$	$y < 10$	H_2
$x \geq 0$	0	y^2	$-y^2$	
H_1 $x < 0$	x	$x + y$	$x - y$	G

Table 2.1: An Example of Normal Function Table

which was based on the use of tabular expression in few practical applications [27, 29]. Later a more general treatment was given by Janicki [11, 13], and Janicki, Khedri[15]. The latter one is currently used as a standard by the SERG group. Both of their works paved the foundation of tabular expression used by SERG [2].

2.2 Definition of Tabular Expressions

In this chapter, the syntax and semantics of table based on the work of Janicki and Khedri [11, 13, 15] will be presented.

First, an example of normal table is presented that can be used to define a function, depicted as Table 2.1.

Informally, this table is to be read as follows. The predicate expression in header H_1 and H_2 partitions the domain, and is used to first select a row (based on the value of x), and then to choose a column (based on the value of y). The expression in the selected main grid G is the value of function.

The header H_1 at the left side of Table 2.1 is the predicate expression in terms of variable x, which partitions the domain of function with regard to variable x; the header H_2 at the top of Table 2.1 is the predicate expression in terms of variable y, which partitions the domain of function with regard to variable y; Together, header H_1 and H_2 partition the domain of the function.

Each cell in the main grid G of table defines the value of the function. For example, when the predicate expressions $x < 0$ and $y > 10$ are true, then the returning value of function is $x + y$. The function is defined as a union of each individual cell in the main grid.

[2]The group SERG was recently renamed to SQC/REL

A definition of the same function with the classical predicate logic is given by:

$$f(x,y) = \begin{cases} 0 & \text{if } (x \geq 0) \wedge (y = 10) \\ y^2 & \text{if } (x \geq 0) \wedge (y > 10) \\ -y^2 & \text{if } (x \geq 0) \wedge (y < 10) \\ x & \text{if } (x < 0) \wedge (y = 10) \\ x+y & \text{if } (x < 0) \wedge (y > 10) \\ x-y & \text{if } (x < 0) \wedge (y < 10) \end{cases}$$

Comparing the tabular form and the conventional form of the same function illustrated above, it is obvious that tabular form is easier to understand, and easier to search for the relevant information. More detailed description can be found in Janicki's work [15].

The key assumptions behind the idea of tabular expressions are:

- The intended behavior of programs is modeled by a (usually complex) relation, say R.

- The relation R may itself be complex but it can be built from a collections of relation R_i, $i \in I$, where I is a set of indices, each R_i can be specified rather easily. In most cases R_i can be defined by a simple formula that can be held in few cells of a table. Some cells define the domain of R_i, the others R_i itself.

- The tabular expression that describes R is a structured collection of cells containing definition of R_i's. The structure of a tabular expression defines how the relation R can be composed for all the R_i's.

2.2.1 Raw Table Skeleton

Intuitively, a table is an *organized collection of sets of cells, each cell contains an appropriate expression.* Such an organized collection of *empty cells*, without expressions, will be called a (raw or medium) *table skeleton.* We assume that a cell is a primitive concept which does not need to be explained.

- A *header* H is an indexed set of cells, $H = \{h_i \mid i \in I\}$, where $I = \{1, 2, \ldots, k\}$ (for some k) is a set of indices.

- A *grid* G indexed by headers $H_1 \ldots H_n$, with $H_j = \{h_i^j \mid i \in I^j\}, j = 1, \ldots, n$, is an indexed set of cells G, where $G = \{g_\alpha \mid \alpha \in I\}$, and $I = I^1 \times \ldots \times I^n$. The set I is the index set of G.

$H_2 = \{h_j^2 \mid j = 1,2,3\}$

	h_1^2	h_2^2	h_3^2
h_1^1	g_{11}	g_{12}	g_{13}
h_2^1	g_{21}	g_{22}	g_{23}

$H_1 = \{h_i^1 \mid i = 1,2\}$ $\quad G = \{g_{i,j} \mid i = 1,2 \wedge j = 1,2,3\}$

Table 2.2: An Example of Raw Table Skeleton

- A *Raw Table Skeleton*, the first approximation of table skeleton, is a tuple $T = (H_1, \ldots, H_n, G)$, where $H_1, \ldots, H_n$ are headers, and G is the main grid indexed by headers $H_1, \ldots, H_n$. The elements of the set $\{H_1, H_2, \ldots, H_n, G\}$ are called *table components.*

Table 2.2 is an example of the Raw Table Skeleton.

2.2.2 Cell Connection Graph and Medium Table Skeleton

The first step in expressing the semantic difference between various types of tables is to define the *cell connection graph*, which characterizes information flow *("where do I start reading the table and where do I get the result?")* of a given table. Intuitively a Cell Connection Graph is a relation that could be interpreted as an *acyclic directed graph* with the grid and all headers as nodes, plus the decomposition of nodes into two distinct classes called *guard components* and *value components.* The only requirement for the relation is that each arc *must either starts from or ends at the grid G.*

Let $T = (H_1, \ldots, H_n, G)$ be a raw table skeleton, i.e. $Components(T) = \{H_1, \ldots, H_n, G\}$.

$$\longmapsto \ \subseteq Components(T) \times Components(T)$$

satisfying:

$$\forall A, B \in Components(T).A \longmapsto B \Rightarrow ((A = G \vee B = G) \wedge A \neq B), \tag{2.1}$$

plus a decomposition of *Components(T)* into *Guards(T)* and *Values(T).*

The relation $\longmapsto^*$ is a transitive and reflexive closure [3] of $\longmapsto$. A component $A \in Components(T)$ is maximum if $A \longmapsto^* B$ implies $B = A$ for every $B \in Components(T)$. Similarly, $A \in Components(T)$ is minimum if $B \longmapsto^* A$ implies $B = A$ for every $B \in Components(T)$. A component $A \in Components(T)$ is neutral if it is neither maximum nor minimum. The components built from the cell describing the domains are never maximum, while the components built from the

[3] $A \longmapsto^* B \Longleftrightarrow (A = B) \cup (A \longmapsto B) \cup (\exists A_1, \ldots, A_k.A \longmapsto A_1 \longmapsto A_2 \longmapsto \ldots \longmapsto A_k \longmapsto B)$

cells containing formulae for values are never minimal. For a Normal Table [23], each element is either maximum or minimum, there is only one maximum.

The partition of $Components(T)$ into $Guards(T)$ and $Values(T)$ must satisfy the following properties:

1. $Components(T) = Guards(T) \cup Values(T)$,
2. $Guards(T) \cap Values(T) = \emptyset$,
3. A is maximal $\Rightarrow A \in Values(T)$,
4. A is minimal $\Rightarrow A \in Guards(T)$,
5. $\forall A \in Guards(T), \forall B \in Values(T).\ A \longmapsto^{+}$ [4] B.

We may now define CCG, *Cell connection Graph*, as a triple

$$CCG = (Guards(T), Values(T), \longmapsto)$$

where $\longmapsto$ satisfies (2.1) and $Guards(T)$, $Values(T)$ satisfies above five properties.

By adding the Cell Connection Graph we obtain the next approximation of the table skeleton concept. By a *medium table skeleton* we mean a tuple $T = (CCG, H_1, \ldots, H_n, G)$, where $(H_1, \ldots, H_n, G)$ is a table skeleton and CCG is a cell connection graph for $H_1, \ldots, H_n, G$.

2.2.3 Raw and Medium Table Elements

Let $T^{med} = (CCG, H_1, \ldots, H_n, G)$ be a medium table skeleton with the index I, and let $T^{raw} = (H_1, \ldots, H_n, G)$ be the table skeleton. Considering the element $(h^1_{i_1}, \ldots, h^n_{i_n}, g_\alpha) \in H_1 \times \ldots \times H_n \times G$.

$$(h^1_{i_1}, \ldots, h^n_{i_n}, g_\alpha) \text{ is a raw element} \iff \alpha = (i_1, \ldots, i_n).$$

We will denote the raw element $(h^1_{i_1}, \ldots, h^n_{i_n}, g_\alpha)$ by $T^{raw} \mid_\alpha$, since it can be interpreted as a kind of projection (restriction) of T^{raw} onto the index α. The *set* $\{h^1_{i_1}, \ldots, h^n_{i_n}, g_\alpha\}$ will be denoted by $Components_\alpha(T^{raw})$.

Let $\longmapsto_\alpha \subseteq Components_\alpha(T^{raw}) \times Components_\alpha(T^{raw})$ be a relation defined as

$$c_1 \longmapsto_\alpha c_2 \iff \exists A_1, A_2 \in Components(T^{raw}).c_1 \in A_1 \wedge c_2 \in A_2 \wedge A_1 \longmapsto A_2.$$

We also define $Guards_\alpha(T^{raw}), Values_\alpha(T^{raw})$ as appropriate projections of $Guards(T^{raw})$ and $Values(T^{raw})$ onto $(h^1_{i_1}, \ldots, h^n_{i_n}, g_\alpha)$, formally

$$Guards_\alpha(T^{raw}) = \{c \mid c \in Componants_\alpha(T^{raw}) \wedge \exists A \in Guards_\alpha(T^{raw}).c \in A)\},$$

[4] $A \longmapsto^{+} B \iff (A \longmapsto B) \cup (\exists A_1, \ldots, A_k.A \longmapsto A_1 \longmapsto A_2 \longmapsto \ldots \longmapsto A_k \longmapsto B$

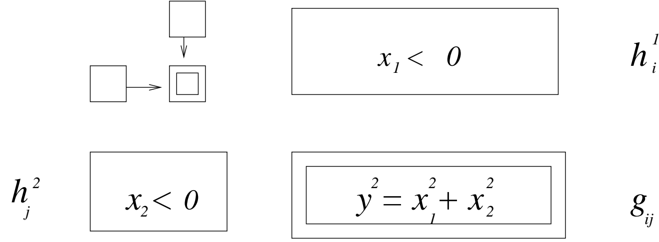

Figure 2.1: An Example of (partially) Interpreted Medium Element.

$$Values_\alpha(T^{raw}) = \{c \mid c \in Componants_\alpha(T^{raw}) \wedge \exists A \in Values_\alpha(T^{raw}).c \in A)\}.$$

The triple $CCG_\alpha = (Guards_\alpha, Values_\alpha, \longmapsto_\alpha)$ will be called the *cell connection graph* of $T^{raw} |_\alpha$. By a *medium element* of T^{med} we mean a tuple

$$T^{med} |_\alpha = (CCG_\alpha, h^1_{i_1}, \ldots, h^n_{i_n}, g_\alpha)$$

where $(h^1_{i_1}, \ldots, h^n_{i_n}, g_\alpha)$ is a raw element. Figure 2.1 illustrates a medium element.

2.2.4 Well Done Table Skeleton

Let R be a relation that will be specified by the tabular expression. Let $dom(R)$ and $range(R)$ denotes the *domain* and *range* of R, respectively. Both $dom(R)$ and $range(R)$ could be Catersian Product or subsets of Catersian Products, i.e. in general $dom(R) \subseteq X_1 \times \ldots \times X_k$, for some X_i, $range(R) \subseteq Y_1 \times \ldots \times Y_m$, for some Y_j.

The relation R can be composed of R_α's, $\alpha \in I$, where I is a finite set of indices, and the set $\mathcal{F} = \{R_\alpha \mid \alpha \in I\}$ will be called as a representation of R. In practice we frequently use a medium element $T |_\alpha$ to specify $R_\alpha, \alpha \in I$. The relation R is equal to $R = expr(\mathcal{F})$, and the table structure is supposed to make the understanding of $expr(\mathcal{F})$ natural and simple.

Let x be a (possible vector) variable over $dom(R)$, y be a (possible vector) variable over $range(R)$, and let $P(x)$ be a predicate defining the domain of R_α, i.e. $x \in dom(R_\alpha) \subseteq dom(R) \iff P(x) = true$.

Let $E_\alpha(x, y)$ be a relational expression that defines a superset E_α of the relation R_α, i.e. $R_\alpha \subseteq E_\alpha$ where $(x, y) \in E_\alpha \iff E_\alpha(x, y)$.

The relation R_α is a restriction of E_α to $dom(R_\alpha)$, i.e $R_\alpha = E_\alpha |_{dom(\alpha)}$, and is entirely described by the following predicate expression: *if* $P_\alpha(X)$ *then* $E_\alpha(x, y)$.

The idea we will be using is the following:

- The expressions defining the relational expression $E_\alpha(x, y)$ are held in *value cells* $(Values(T))$.

- The expressions defining the predicate expression $P_\alpha(x)$ are held in *guard cells* ($Guards(T)$)

To define precisely how the *medium element* can be used to specify the expression **if** $P_\alpha(x)$ **then** $E_\alpha(x,y)$, we need not only to divide cells into value and guard types, but also to decide how $P_\alpha(x)$ can be built from the expressions held in the guard cells and $E_\alpha(x,y)$ from the expressions held in the value cells.

Let $T = (CCG, H_1, \ldots, H_n, G)$ be a medium table skeleton. Assume that $Guards(T) = \{B_1, \ldots, B_r\}$, $Values(T) = \{A_1, \ldots, A_r\}$.

- A predicate expression $P_T(B_1, \ldots, B_r)$, where $B_1, \ldots, B_r$ are variables, is called a table predicate rule.

- A relation expression $r_T(A_1, \ldots, A_s)$, where $A_1, \ldots, A_s$ are variables, is called a table relation rule.

The predicate $P_\alpha(x)$ can now be derived from $P_T(B_1, \ldots, B_r)$ by replacing each variable B_i by the content of the cell that belongs to both the medium element Γ_α and the component B_i. Similarly, the relation expression $E_\alpha(x,y)$ can now be derived from $r_T(A_1, \ldots, A_s)$ by replacing each variable A_i by the content of the cell that belongs to both the medium element Γ_α and the component A_i.

A relational expression C_T in the form $R = Expr(\mathcal{F})$ is called a table *composition rule.* In general, $Expr(\mathcal{F})$ is a relational expression built from the expression defining R_α's, and various relational operators.

The final approximation of a table skeleton is the following.

- A *well done table skeleton* is a tuple $T = (P_T, r_T, C_T, CCG, H_1, \ldots, H_2, G)$, where $(CCG, H_1, \ldots, H_2, G)$ is a medium table skeleton, P_T is a table predicate rule, r_T is a table relation rule, and C_T is the table composition rule.

In Table 2.1, p_T is $H_1 \wedge H_2$, which means that the predicate expressions, are obtained by the conjunctions of contents of each cells in header H_1 and H_2; The r_T being G means that if guard cells are h_i^1 and h_j^2, then the corresponding value cell is in the cell $g_{i,j}$.

In principle, a well done table skeleton defines all the structure of a tabular expression except filling out all the cells with proper expressions that define all R_a.

In general, the table composition rule C_T should give a precise answer to the question, "how to build the whole, i.e R, from the parts, i.e. R_α" in term of algebra of relation. In [14], various operations are introduced. However, for our purpose, only *union* operation is used to build a whole relation R from all of R_α.

2.2.5 Tabular Expressions

We are now able to define formally the concept of tabular expression.

- A *tabular expression* is a tuple

$$T = (P_T, r_T, C_T, CCG, H_1, \ldots, H_n, G; \Psi, IN, OUT)$$

  where $(P_T, r_T, C_T, CCG, H_1, \ldots, H_n, G)$ is a well done table skeleton, and ψ is a mapping that assigns a predicate expression, or part of it, to each guard cell, and a relational expression, or part it, to each value cell. The predicate expression has variables over *IN*, the relational expression have variables over $IN \times OUT$, where *IN* is the set (usually heterogeneous product) of inputs, and *output* is the set (usually heterogeneous product) of output.

For every tabular expression T, we define the *signature* of T as:

$$Sign_T = (P_T, r_T, C_T, CCG)$$

The signature describes all the global and structural information about the table. We may say that a tabular expression is a triple: *signature, raw skeleton*–which describes the number of elements in headers and indexing of the grid, and the *mapping* ψ – which describes the contents of all cells.

Both P_T and r_T must satisfy the following consistency rule:

- For every $\alpha \in I$, P^T_α is a syntactically correct predicate expression.
- For every $\alpha \in I$, r^T_α is a syntactically correct relational expression.

2.2.6 Semantics of Tabular Expressions

Let $T = (P_T, r_T, C_T, CCG, H_1, \ldots, H_n, G; \Psi, IN, OUT)$ be a tabular expression, with the index I, and let $\alpha \in I$. By an *interpreted medium element*, we mean a tuple:

$$T|_\alpha = (P_T, r_T, CCG_\alpha; \psi \mid_{Components_\alpha T}).$$

Figure 2.1 plus $P_T = H_1 \wedge H_2$, represents an example of interpreted medium element.

For every $\alpha \in I$, we define A_α, E_α, as

$$x \in A_\alpha \Longleftrightarrow P_\alpha(x) = true, \;\; (x, y) \in E_\alpha \Longleftrightarrow E_\alpha(x, y).$$

Every interpreted medium element $T|_\alpha$ describes now the relation R_α, i.e.

$$(x, y) \in R_\alpha \Longleftrightarrow \textbf{if } P_\alpha(x) \textbf{ then } E_\alpha(x, y).$$

We may now define the semantics of tabular expressions in a formal way:

- The relation R_α describes the semantics of the *interpreted medium skeleton* $T|_\alpha$.
- The *semantics* of a *tabular expression* T is defined by

$$R_T = C_T(R_\alpha)$$

Further detailed treatments of syntax and semantics of tabular notation can be found in other works [13, 14, 23].

In the specification of concurrent systems, the most commonly used tables are the normal function table and normal relation table. Following is the formal definition of their p_T, r_T rules and C_T relation.

- $p_T = H_1 \wedge H_2$, header H_1 or H_2 partitions the current states into disjoint sets. Header H_2 or H_1 partitions the global events into disjoint sets
- $r_T = G$, the content in grid G is the set of next states
- $C_T = \cup_{\alpha \in I}$

Chapter 3

Introduce to Finite State Automata

In this chapter, deterministic and nondeterministic finite state automata as well as composite finite automata [7, 20, 35, 36] will be introduced. The application areas of these automata will also be discussed.

It has been recognized recently that *automata theory* is a successful medium in specifying and analyzing of the sequential and concurrent computer system, including both software and hardware systems. The reason for the success is mainly due to the simplicity of these techniques. Indeed, they are easy to implement and easy to use. Moreover, they have well defined semantics and are well understood by the computer society.

The finite state automaton, or finite state machine (abbreviated as FA), is a mathematical model of a dynamic system, with discrete inputs and outputs. The system can be in any one of a finite numbers of internal configurations or "states". The state of the system summarizes the information concerning previous inputs that are needed to determine the behavior of the system on subsequent inputs.

Applications for finite automata can be any devices in which there are a finite set of inputs and a finite set of parameters that must be "remembered" by the device.

A FA, particularly a non-deterministic automaton (NFA) is a useful tool for specifying a concurrent system. The beauty of the finite automaton is that it only needs to remember finite amount of information, namely the current state. This property makes the model easier to design and understand.

3.1 Finite State Automata Definition

First of all, we list some general concepts of automata theory. An *alphabet* Σ is a finite nonempty set. A word over Σ is a finite string $P = x_1 \dots x_n (x_i \in X, i = 1, \dots, n)$ of elements of Σ. For the *empty word*, i.e. for the word of length of 0, the notation ε is used.

Intuitively, a finite state automaton is a model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function. There are many variants, for instance, machines have actions (outputs) associated with transitions (Mealy machine) or states (Moore machine), multiple start states, transitions conditioned on no input symbol (a null) or more than one transition for a given symbol and state (nondeterministic finite state machine), one or more states designated as accepting states (recognizer), etc. In the thesis, non-deterministic finite automata will be used to model computing system.

3.1.1 Deterministic Finite Automaton (DFA)

A *DFA* consists of a finite set of states and a set of transitions from state to state that occurs on input symbols chosen from an alphabet Σ. For each input symbol there is exactly one transition output of each state (possibly back to the state itself). One state, usually denoted by q_0, is the initial state, from which the automaton starts. Some states are designated as final state. More formally, a *deterministic finite automaton* is 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*, or *input symbol*,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the *transition function*,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the *set of accept states* or *final states*.

Finite automata are frequently given by means of *transition tables*. If $M = (Q, \Sigma, \delta, q_0, F)$ is an automaton with m states and n input symbols, then the table of M has n rows and m columns. Each row is labeled by an input signal such that different rows have different labels. Furthermore, columns are also labeled by state in a one-to-one manner. If the label of ith row is x and the label of jth column is a, then the entry in the ith row and j^{th} column is $\delta(a, x)$:

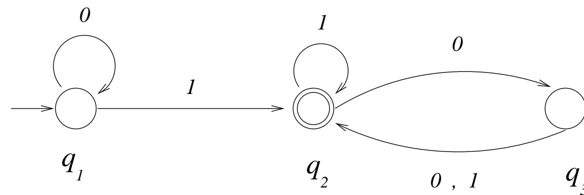

Figure 3.1: The Finite Automaton M_1

δ	$\cdots a \cdots$
$\vdots$	$\vdots$
x	$\cdots \delta(a,x) \cdots$
$\vdots$	$\vdots$

For example, let us consider the deterministic finite automaton M_1 depicted in Figure 3.1.

We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0, 1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state, and
5. $F = \{q_2\}$

It can be seen from the table that for every state on one input symbol 0 or 1, there is only one corresponding next state.

Extension of Notation: the function δ^* for DFA

If $M = (Q, \Sigma, \delta, Q_0, F)$, we now have a concise way of writing "The state to which the machine M responses if it is in the state q and receives an input symbol a," which is $\delta(q, a)$. We would like an equally concise way of writing "The state in which M endes up, if it begins with state q, and receives the string x of several input symbols. Let us write this as $\delta^*(q, x)$, and δ^* may be considered as an extension of the transition function δ, which is defined as $Q \times \Sigma$, to the large size $Q \times \Sigma^*$.

Definition 3.1
let $M = (Q, \Sigma, \delta, Q_0, F)$ *be a FA defining the function* $\delta^* : Q \times \Sigma^* \rightarrow Q$ *recursively as follows.*

1. *For any* $q \in Q, \delta^*(q, \varepsilon\ ) = q$
2. *For any* $y \in \Sigma^*, a \in \Sigma,$ *and* $q \in Q, \delta^*(q, ya) = \delta(\delta^*(q, y), a)$

The extended transition function can be used to decide whether certain string belongs to the language defined by the FA. Let x be the input string, q_0 be the initial state, and F be the set of final states. If $\delta^*(q_0, x) \subseteq F$, then x belongs to the language defined by the DFA.

3.1.2 Nondeterminism Finite Automaton

Nondeterminism is a useful concept to model the concurrent system. For *DFA*, the transition function δ specifies exactly one next state for each possible combination of a state and an input symbol. For *NFA*, several choices may exist for the next state at any given point. In other words, every state of a DFA always has one exit transition arrow for each symbol in the alphabet, but for a NFA a state may have zero, one, or many existing arrows for each alphabet symbol. More formally,
A *nondeterministic finite automaton* is 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*, or *input symbol*,
3. $\delta : Q \times \Sigma \longrightarrow 2^Q$ is the *transition function*,
4. $q_0 \in Q$ is the *start* state, and
5. $F \subseteq Q$ is the *set of accept states*

For example, let us consider the finite automaton M_2 depicted in Figure 3.2. The formal description of M_2 is M_2=$(Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0, 1\}$,
3. δ is described as

	0	1
q_1	$\{q_1, q_2\}$	$\{q_1\}$
q_2	$\{q_3\}$	$\emptyset$
q_3	$\emptyset$	$\{q_4\}$
q_4	$\{q_4\}$	$\{q_4\}$

4. q_1 is the start state, and
5. $F = \{q_4\}$

It can be seen from transition function δ that at state q_1 on input symbol 1, the next state is a set of states $\{q_1, q_2\}$.

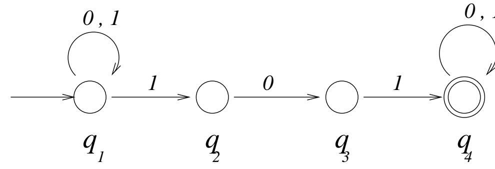

Figure 3.2: The Nondeterminism Finite Automaton M_2

Recursive definition of δ^* for a NFA

Let $M = (Q, \Sigma, \delta, Q_0, F)$ be a NFA, define the function $\delta^* : Q \times \Sigma^* \to 2^Q$ recursively as follows.

1. For any $q \in Q, \delta^*(q, \varepsilon) = q$
2. For any $y \in \Sigma^*, a \in \Sigma$, and $q \in Q, \delta^*(q, ya) = \bigcup_{p \in \delta^*(q,y)} \delta(p, a)$

If no confusion will be made in the context, δ is used instead of δ^* in this thesis.

3.2 Concurrent composition of automata

Traditionally, finite automata are considered to be a mathematical tool for modeling sequential systems, not concurrent systems. However, along with the development of formal method, recently there are significant numbers of researchers interested in using automata to model concurrent systems.

In this section, different approaches for combining independently finite automata are investigated in order to construct composite automata.

3.2.1 Parallel Composition

Shields [35] in his works defined a parallel composition of automata out of component automata.

Let M_A and M_B be the two individual finite state automata, two component machines that run synchronously in parallel. The new machine, also a finite state automaton, will take a pair of inputs, one from Σ_A and Σ_B. These inputs are processed by two component automata in the normal way, and the outputs are produced together as a pair. This parallel composition of two machines M_A and M_B is written as $M_A \mid\mid M_B$.

Formally, let $M_A = (Q_A, \Sigma_A, \Gamma, \delta_A, \lambda_A, q_{A0})$ and $M_B = (Q_B, \Sigma_B, \Gamma, \delta_B, \lambda_B, q_{B0})$ be finite automata, then their parallel composite, $M_A \mid\mid M_B$ is defined to be the machine M, where

1. $Q = Q_A \times Q_B$.
2. $\Sigma = \Sigma_A \times \Sigma_B$.
3. $\delta((s_1, s_2), (i_1, i_2)) = (\delta_1(s_1, i_1), \delta(s_2, i_2))$
4. $\lambda((s_1, s_2), (i_1, i_2)) = (\lambda_1(s_1, i_1), \lambda_2(s_2, i_2))$
5. $q_0 = (q_{A0}, q_{B0})$.

Obviously, Shields' method synchronizes the composition of independent processes and is suitable to construct composite automata from parallel running component automata that do not interfere with each other. Taking a close look at this method, we should observe that it does not provide a proper mechanism to specify asynchronous events of component automata as well as shared objects by different components.

3.2.2 General Composition of Finite Automata

Gecseg [7] defined a general form of products of automata. In his method, all of the component states are fed back to each other, products of finite state automata are obtained by composition of certain individual finite state automata in such a way that the resulting system is also a finite state automaton. The most general form of such a composition of a finite system of automata can be derived by determining the actual input of each component automaton, depending on the actual states of these machines and on the actual input signal of the system; the output of the system also depends on the states of the component machines and on the input of system.

Definition 3.2 (General product)
Let $P_i = (Q_i, \Sigma_i, \Gamma_i, \delta_i, \lambda_i, q_{i0})(i = 1, \ldots, k; k > 0)$ be a system of finite automata, and assume two mappings.

- $\varphi : Q_1 \times \ldots \times Q_k \times \Sigma \rightarrow \Sigma_1 \times \ldots \times \Sigma_k$
- $\psi : Q_1 \times \ldots \times Q_k \times \Sigma \rightarrow \Gamma$

Let us define the composite finite automaton $P = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ where

1. $Q = Q_1 \times \cdots \times Q_k$ *is a finite set called the* states, *the element of* Q *is* $(q_1, \ldots, q_k)$ *where* $(q_1, \ldots, q_k) \in Q \Rightarrow q_1 \in Q_1 \wedge \cdots \wedge q_k \in Q_k$
2. $\Sigma = \Sigma_1 \times \cdots \times \Sigma_k$ *is finite set called the* alphabet, *or* input symbol
3. $\delta : Q \times \Sigma \rightarrow Q$ *is the* transition function, $\delta((q_1, \ldots, q_k), x) = (\delta_1(q_1, x_1), \ldots, \delta_k(q_k, x_k))$, *where* $(q_1, \ldots, q_k) \in Q$, $x \in X$ *and* $(x_1, \ldots, x_k) = \varphi(q_1, \ldots, q_k, x)$.
4. $\lambda((q_1, \ldots, q_k), x) = \phi(q_1, \ldots, q_k, x)$ *is the* output function, *and*

5. $q_0 = (q_{10}, \cdots, q_{j0}) \in Q$ *is the start state.*

From the above definition of an automata defined by general composition, it is clear that the actual transition of individual component automata relies on the actual state of each automaton and the input symbol of system, therefore this method can be easily utilized to synchronize the joint events of each individual automaton. However, this method does not explicitly provide a solution when the transition of individual components also rely on the value of shared objects.

3.2.3 Asynchronous Automata

Asynchronous automata, introduced by Zielonka, Klarlund, Mukund, Sohoni [20, 36], are natural generations of finite-state automata for concurrent systems. In its terminology, an asynchronous automata consists of a set of processes that periodically synchronize to process their inputs. Each letter a in the alphabet is associated with a subset $\theta(a)$ of processes which jointly decide on a move on reading a. A distributed alphabet of this type gives a rise to an *independence relation* I between letters: $(a, b) \in I$ if a and b are processed by disjoint sets of components.

Distributed alphabet. Let $\mathcal{P}$ be a finite set of processes. A *distributed alphabet* is a pair of (Σ, θ), where Σ is a finite set of *actions* and $\theta : \Sigma \to 2^{\mathcal{P}}$ assigns a non-empty set of processes to each $a \in \Sigma$

State space. With each process p, we associate a finite set of states denoted by V_p. Each state in V_p is called a *local state*. For $P \subseteq \mathcal{P}$, V_P is used to denote the product $\prod_{p \in P} V_p$. An element $\vec{v}$ of V_P is called a *P-state*. A $\mathcal{P}$ state is also called a *global state*. Given $\vec{v} \in V_P$ and $P' \subseteq P$, we use $\vec{v}_{P'}$ to denote the projection of $\vec{v}$ onto $V_{P'}$. Also, $\vec{v}_{\overline{P'}}$ abbreviates $\vec{v}_{P-P'}$. For singleton $p \in P$, we write $\vec{v}_p$ for $\vec{v}_{\{p\}}$. For $a \in \Sigma$, V_a stands for $V_{\theta(a)}$ and $V_{\overline{a}}$ stands for $V_{\overline{\theta(a)}}$. Similarly, if $\vec{v} \in V_P$ and $\theta(a) \subseteq P$, we use $\vec{v}_a$ for $\vec{v}_{\theta(a)}$ and $\vec{v}_{\overline{a}}$ for $\vec{v}_{\overline{\theta(a)}}$.

Asynchronous automaton. An asynchronous automaton $\mathcal{U}$ over (Σ, θ) is of the form $(\{V_P\}_{p \in \mathcal{P}}, \{\to_a\}_{a \in \Sigma}, \mathcal{V}_0, \mathcal{V}_F)$, where $\to_a \subseteq V_a \times V_a$ is the local transition relation for a, and $\mathcal{V}_0, \mathcal{V}_F \subseteq V_{\mathcal{P}}$ are set of *initial* and *final* global states. Each relation $\to_a$ specifies how the process $\theta(a)$ that meets on a may decide on a joint move. Other processes do not change their state. Thus, we define the *global transition relation* $\Rightarrow \subseteq V_{\mathcal{P}} \times \Sigma \times V_{\mathcal{P}}$ by $\vec{v} \stackrel{a}{\Longrightarrow} \vec{v'}$ if $\vec{v}_a \to_a \vec{v'}_a$ and $\vec{v}_{\bar{a}} = \vec{v'}_{\bar{a}}$

$\mathcal{U}$ is called deterministic if the global transition relation of $\mathcal{U}$ is a function from $V_{\mathcal{P}} \times \Sigma$ to $V_{\mathcal{P}}$ and if the set of initial state $\mathcal{V}_0$ is a singleton.

This model is a natural representation of the concurrent system, but it poses the same inconvenience as *general composite automata*, namely when the actions of individual component automata need to be synchronized with the involved objects that competes with the resources.

Zielonka's Asynchronous Automaton seems to be good for theoretical considerations, but their use for practical specifications seems to be rather limited (see [6]).A Major problem is that in reality the *independency relation* I is seldom completely static, it rather depends on some global state S, but dynamic $I(s)$ cause a lot of theoretical problem. See [16] for some analysis.

Chapter 4

Global Finite Automata

In this chapter, a simple, but effective approach for specifying concurrent systems with the global state machine is outlined first. Then, the formal semantics of global finite automata system are defined.

4.1 An Approach to the Problem

This section describes general ideas of our approach to write documentation and specification of a concurrent system.

First of all, the notion of event ordering is introduced. Some events may be allowed to happen at the same time or concurrently while others must happen in a predetermined order or sequentially. We call such situations the temporal ordering of events and we will call systems with these types of properties concurrent systems and sequential systems, respectively. For concurrent system, very often the ability to proceed with an activity will depend on some other activities being completed or having reached a suitable stage. For example, in a producer and consumer problem, the producer puts a product into the buffer and the consumer gets the product from the buffer. The consumer must wait until the buffer is not empty and the producer must not make products if the buffer is full.

In the specification of concurrent systems, we are particularly concerned with the patterns of synchronization between processes as well as objects in the system. As the system as a whole evolves so do the individual processes. It is important that we find some means of modeling the evolution of a system and, in particular concurrent properties in our context.

The particular techniques of interest are the specification of complex systems by the parallel composition of simpler ones, using restrictions where necessary to constrain the observable system behavior as a whole, and the system specification in

terms of their general state transitions.

Through investigations, it is found that finite state automata can best serve as this role. Finite state automaton is an abstract virtual machine with a very well defined behavior. As both a graphical and mathematical model, automata are convenient tool to model both sequential and concurrent systems. They can be used to model processes that run independently as well as concurrently. Such processes take particular states and perform state changes. The argument is that such states or changes that are coincidental may be combined into a global state change.

In this thesis, a concurrent system is considered to be a set of concurrent processes that may be cooperating (and need to be synchronized with them) or competing to acquire some resources. The goal of this thesis is to develop a tool to write a formal specification and documentation to synchronize different processes.

In this thesis, whole systems are viewed as normal non-deterministic automata at the "level" of "global states". Formally, the model advocated in [18] will be used. Basically, during the life cycle of each process, it changes its state from one to another or simply stays at the same state, depending on the actual input symbols and the current process state as well as the value of shared object. Moreover, the global state change is the combined result of individual process.

Certainly it is desired to record all of the possible states a system can evolve to through the actions of its various processes. At any instance there may be a number of processes, each of which is capable of a variety of next actions. One way to model the change of a system state is to model all of the changes that could occur. For a small system, it is possible to do so, but for a real practical system, it is not an easy task to model all of the changes of the global state as the results of state explode. Therefore, we do not attempt to use this approach. The divide and conquer strategy is utilized instead, namely the synchronization of different processes is considered at the global level and the actual actions of each individual process are delegated to their local level.

The goal is to develop specification methods for concurrent finite-state system that avoid the part of combinatorial state-space explosion problem, namely the explosion of the number of state due to the modeling of concurrency by interleaving. To this end, the tabular expressions are used to specify the constraints that must be satisfied by all of the processes that interact with each other. Further treatments will be illustrated in later sections.

To specify a concurrent program using tabular expressions, we propose the idea of constraints, in which every implementation of a concurrent program either satisfies the constraint imposed by the specification, or fails to implement the specification.

Constraints are requirements that are imposed on the solution by circumstance, force, or compulsion. Constraints limit absolutely the options open to a designer

of a solution by imposing immovable boundaries and limit. They can be policies for database integrity, resource limits etc.

For example, in the Readers-Writers system, our leading example, one of the constraints is whenever one of writers is in the writing state, other writers cannot enter the writing state and readers can not enter the reading state. Another constraints is whenever one of readers is in the reading state, writers cannot enter the writing state.

A software specification should identify constraints as a part of the final product. The designer, who implemented customer specifications, for any number of reasons, has design preferences. However, if his/her design did not meet customer's constraints, then the solution is not acceptable.

We consider executing of concurrent programs as states changing, from initial state to next state, etc in which they must satisfy the system constraints. The point is that the system can perform and only perform the transition our model allows, namely, satisfy the constraints that the system imposes.

For more than one processes accessing the same resource or critical section, concurrently, the system constraint will be considered when processes entering and leaving the critical section. This is the part we are interested in the most for concurrent issues. For example, when readers and writers both try to access the same database resource, it is considered that either readers or writers can enter the database to read or to write. In another situation, when they leave the critical section, they should satisfy the condition for the global state. For example, when all of reading readers and writing writers leave the critical section, the resource should enter the resource free state.

To clearly express the method, this approach is separated into few steps. First of all, the underlying concurrent system is analyzed and all the processes and objects involved in the system are identified, then nondeterministic finite state automata are used to model each individual process. For application concern in this stage, the state of the finite state automata can be combined if different values of one variable make no significant contribution to the definition of state. For example, if certain aspects of the state are irrelevant to the behavior being observed, these aspects can be ignored.

Secondly, these state machines are combined together to general state machines, which are called *global finite state automata*. They are used to model the concurrent system. The formal definition of global finite state automaton is given and the transition function is provided. The reason that the transitions function of the global state automata are not given as the traditional one is simply because it is neither convenient nor necessary. Since outside the critical section, the individual automata model can be used to check individual process. The transition function of global state automata will be given by the tabular form. The proof of completeness of our table

is also provided.

Thirdly, the system invariant is specified and the proof of the invariant held for the model specification is given. Finally, the deadlock free properties of the tabular specification are proved.

The formal tools associated with the methodology proposed here are elementary. Finite automata, particular the nondeterministic finite automata, are used to model each process involved as well as the entire system.

4.2 Global State Machines

It is assumed that concurrent systems are composed of different components, called processes, that can act in parallel and interact with each other. It is also assumed that processes have a finite state, and processes can be synchronized by executing together a joint transition and by performing an operation on shared objects. Furthermore, a possibility of simultaneous execution of a and b implies a possibility of execution in the order a followed by b, and in the order b followed by a (paradigm π_8 according to Janicki and Koutny classification [10]).

4.2.1 Model for Each Individual Process

First of all, each process in the concurrent system is modeled as an NFA. Let $I = \{1, \ldots, n\}$ [1] be an index set of processes. For all $i \in I$, $P_i = (Q_i, \Delta_i, \tilde{\delta}_i, q_i^0)$ [2], where Q_i is the set of local states of process P_i, Δ_i is the alphabet, $\tilde{\delta}_i$ is the transition function, and q_i^0 is the initial state.

It is assumed that $\Sigma_i \subseteq \Delta_i \subseteq \Sigma_i \cup \overline{\Sigma_i}$, where Σ_i is the set of all actions/events, and $\overline{\Sigma_i} = \{\overline{a} \mid a \in \Sigma_i\}$ is the set of co-actions/co-events. The meaning of the elements of set $\overline{\Sigma_i}$ will be explained shortly.

Without any loss of generalization, we may assume that $Q_i \cap Q_j = \emptyset \Leftrightarrow i \neq j$, $\Sigma_i \cap \Sigma_j = \emptyset \Leftrightarrow i \neq j$. Even if two processes P_i and P_j have virtually identical action a, the a from P_i can be renamed to a_i, and the a from P_j can be renamed to a_j.

Let O be a finite set of objects shared by the processes, and for every $o \in O$, let v_o denote the value of an object o, and V_o denote the set of values that can be assigned to the object o. Let also V denote the Cartesian product $\prod_{o \in O} V_o$. We may also assume that $V_o \cap V_{o'} = \emptyset \Leftrightarrow o \neq o'$. We use $V_{O'}$ to denote the Cartesian product $\prod_{o \in O'} V_o$. It is assumed that $O' \subseteq O$. Let $\vec{v}_O = (v_1, \ldots, v_k) \in V$, we use $\vec{v}_{O'}$ to denote the projection of $\vec{v}_O$ onto $V_{O'}$.

[1] n is the number of processes in the concurrent system.

[2] Interactive systems and concurrent systems are not intended to stop, so it is meaningless to model these system with a final state, therefore the final state is not applicable in our model.

The set Q_i and the state q_i^0 are defined as in standard finite state automaton theory. However, the definition of events is extended, namely to associate two sets to each event a of process P_i. One set, called the *synchronization set*, consists of events of process $P_j, (j \neq i) \wedge (j \in \{1, \ldots, n\})$ to be synchronized with event a by executing the joint transition; another set, called the *competition set*, consists of events of process $P_j, (j \neq i) \wedge (j \in \{1, \ldots, n\})$ that competes for access to the shared objects.

Formally, let $\Sigma = \bigcup_{i=1}^n \Sigma_i$, $\overline{\Sigma} = \bigcup_{i=1}^n \overline{\Sigma_i}$, $\Delta = \bigcup_{i=1}^n \Delta_i$, $\Sigma^j = \Sigma - \Sigma_j, j \in I$. Let $S : \Sigma \rightarrow 2^\Sigma$, and $C : \Sigma \rightarrow 2^\Sigma$ be two mappings, such that $\forall a \in \Sigma . a \in \Sigma_i \Rightarrow S(a) \subseteq \Sigma^i \wedge C(a) \subseteq \Sigma^i$. The set $S(a)$ is called the synchronization set of a, and the set $C(a)$ is called the competition set of a. It is assumed that $S(a) \cap C(a) = \emptyset$.

Let $\nu : \Sigma \rightarrow \prod_{o \in O'} 2^{V_o}$ be a mapping interpreted as follows. If $O' = \{o_1, \ldots, o_j\}$, $\nu(a) = V'_{o_1} \times \ldots \times V'_{o_j}$ means that each $V'_{o_i}, i \in \{1, \ldots, j\}$ is the set of values of the shared object o_i for which the set of events $C(a) \cup \{a\}$ has equal opportunity to compete. For example, in the Readers-Writers problem, the database is modeled as a shared object. When the database is in free status (no reader and writer is accessing the database), readers and writers have equal opportunity to compete to access it. For most of the applications, $\mid O' \mid = \mid \{o\} \mid = 1$, and then $\nu : \Sigma \rightarrow 2^{V_o}$. It is assumed that $S(a) = C(a) = \emptyset$ implies $\nu(a) = \emptyset$.

The set $S(a)$ is $\emptyset$ whenever event a does not perform any joint transition with other events. The set $C(a)$ is $\emptyset$ whenever event a does not compete for any shared objects. An event $a \in \Sigma$ is *independent* if $S(a) = C(a) = \emptyset$, otherwise it is a *dependent* event. The set $\Delta_i, i \in I$, must satisfy: $\forall \overline{a} \in \overline{\Sigma_i}.\ a$ is independent $\Leftrightarrow \overline{a} \notin \Delta_i$

There are two status for a given event "a", namely *enable* and *disable*. Whenever an event "a" satisfies all the restrictions (if there are any) that the concurrent system imposes on it, it is at *enable* status, written as a. Otherwise it is in a *disable* status, written as $\overline{a}$.

Let $\sigma : \Delta \rightarrow \{enable, disable\}$, and

$$\sigma(a) = \begin{cases} enable & \text{if } a \in \Sigma \\ disable & \text{if } a \in \overline{\Sigma} \cap \Delta \end{cases}$$

The function σ is useful form the application viewpoint even though it is completely defined by the distribution of Δ into Σ and $\overline{\Sigma}$.

For an event a of process P_i, if a is an independent event, it is always evaluated to *enable* status. In this case, process P_i transits to the next state according to the transition function; If set $S(a)$ and/or $C(a)$ of an event a is not empty, it is evaluated to *disable* from the function σ, and this event leads the process P_i staying at the same state. Through the composition of other proper processes, it is possible to eliminate all the elements of set $S(a)$ and $C(a)$, then the event a is *enable* and can be executed to lead the process P_i to next state, possibly to the same state, according to the

local transition function. Further details of elimination will be discussed in the next section.

The transition function $\tilde{\delta_i}$ is slightly extended to the traditional finite state automata. The alphabet of traditional automata consists of events/actions, or possibly ε for NFA. The alphabet of P_i consists of events/actions, part of co-events/co-actions, and ε. For independent action a, only one transition needs to be defined for a given state q, namely $\tilde{\delta_1}(q, a)$; For non-independent action b, two transitions need to be defined for a given state q, one for $\sigma(b) = enable$, another for $\sigma(b) = disable$, namely $\tilde{\delta_1}(q, b)$ and $\tilde{\delta_1}(q, \overline{b})$ respectively. It is assumed that the ε action upon any state leads to that state itself. Formally $\tilde{\delta_i}(q_i, \varepsilon) = q_i$, where $\tilde{\delta_i}$ is the transition function of process i, $q_i \in Q_i$

Let $q_j, q_k \in Q_i$, $\hat{a} \in \Delta_i$. Function $\tilde{\delta_i}$ is defined as follows:

$$\tilde{\delta_i}(q_j, a) = \begin{cases} \{q_j\} & \text{if } \hat{a} = \varepsilon, \\ \{q_k\} & \text{if } (\hat{a} \in \Sigma_i) \wedge (q_k \text{ is the next state of the transition}), \\ \{q_j\} & \text{if } \hat{a} \in \overline{\Sigma_i} \wedge (b \in \Sigma_i \wedge \overline{b} = \hat{a} \wedge \tilde{\delta_i}(q_j, b) \text{ is defined}) \end{cases}$$

Each process is defined as so called *deterministic nondeterministic automaton.* Deterministic means that the transition function $|\delta_i(q_i, a)| \leq 1$, where $q_i \in Q_i$ and $a \in \Sigma$; Nondeterministic means that not all event and state combination is defined. In other words, an automaton is deterministic since no nondeterministic choice is made, but it is nondeterministic according to the standard automaton theory.

It is assumed that all undefined transitions are considered as error and this error propagates to the next level, namely global automata level. Therefore, no matter what level the error happens, it should been considered as an error of the whole system.

Let us consider a producer and consumer example. In this model, producer produces one product once a time and puts it into a buffer; The buffer has a capacity of two, which means that it can contain at most two products; The consumer consumes the product that fetched from the buffer once a time. The producer and consumer compete to access the buffer when $v_{buf} = 1$, where a producer can produce a product or a consumer can consume one product. The producer cannot produce if buffer is full, and consumer cannot consume if buffer is empty.

The buffer "buf" has three values, namely $0, 1, 2$, which represent no product in the buffer, only one product in the buffer, and two products in the buffer, respectively. Hence $V_{buf} = \{0, 1, 2\}$. The producer is modeled as $P_1 = (Q_1, \Delta_1, \tilde{\delta_1}, q_1^0)$, depicted as left of Figure 4.1, where

1. $Q_1 = \{s_1, s_2\}$ is a finite set of states, s_1 means that producer is ready to produce, and s_2 means that producer finishes making product.
2. $\Sigma_1 = \{t_1, t_2\}$ is a finite set of *events*, and $\overline{\Sigma_1} = \{\overline{t_1}, \overline{t_2}\}$ is the finite

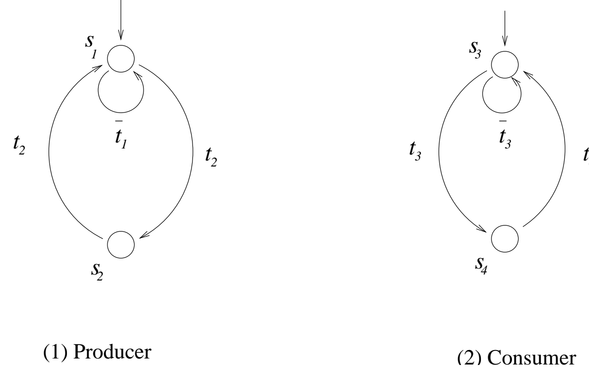

Figure 4.1: The Finite State Automata of Producer and Consumer

$\tilde{\delta}_1$	t_1	$\overline{t}_1$	t_2	ε
s_1	$\{s_2\}$	$\{s_1\}$		$\{s_1\}$
s_2			$\{s_1\}$	$\{s_2\}$

Table 4.1: Transition Function of Producer

set of *co-events*. There are two events for producer. One of them t_1 is not a joint event since set $S(t_1)$ is empty, and it competes to access object *buf* with event t_3 of process P_2 when $v_{buf} = 1$. $C(t_1) = \{t_3\}$ and $\nu(t_1) = \{1\}$. Another event t_2 is an independent event and it is free to execute the transition without considering other events. $\Delta_i = \{t_1, t_2, \overline{t_1}\}$ is the *alphabet*, where $\Sigma_1 \subseteq \Delta_1 \subseteq \Sigma_1 \cup \overline{\Sigma_1}$
3. $\tilde{\delta}_1 : Q_1 \times \Delta_1 \rightarrow 2^{Q_1}$ is the *transition function* defined as table 4.1, and
4. $s_1 \in Q_1$ is the initial state of producer.

The consumer is modeled similarly as the producer. It is modeled as $P_2 = (Q_2, \Delta_2, \tilde{\delta}_2, q_2^0)$, depicted as right of Figure 4.1, where

1. $Q_2 = \{s_3, s_4\}$ is a finite set of states, s_3 means that consumer is ready to consume the produce, and s_4 means consumer finish consume product.
2. $\Sigma_2 = \{t_3, t_4\}$ is a finite set of *events*. $\overline{\Sigma_2} = \{\overline{t_3}, \overline{t_4}\}$ is a finite set of *co-events*. There are two events for consumer. One of them t_3 is not a joint event since set $S(t_3)$ is empty, and it competes to access object *buf* with event t_1 of process p_1. Another event t_4 is an independent event and it is free to execute the transition without considering of other events. $\Delta_2 = \{t_3, t_4, \overline{t_3}\}$

$\tilde{\delta}_2$	t_3	$\bar{t}_3$	t_4	ε
s_3	$\{s_4\}$	$\{s_3\}$		$\{s_3\}$
s_4			$\{s_3\}$	$\{s_4\}$

Table 4.2: Transition Function of Consumer

3. $\tilde{\delta}_2 : Q_2 \times \Delta_2 \to 2^{Q_2}$ is the *transition function* defined as Table 4.2, and
4. $s_3 \in Q_2$ is the initial state of consumer.

From the model of producer and consumer, it is easy to understand that if an event of a process is not an independent event, it cannot be evaluated solely by the information of itself. It needs other processes' activities to joint decide a move on the restrictions that concurrent system imposes on them. Hence, we need compose individual process to a general finite state automata to eliminate the elements of set $S(a)$ and $C(a)$ for event a.

4.2.2 Model for Composite Process

In the process based specification techniques, a concurrent system is modeled as a set of interconnected processes. Each process can perform some transitions and interact with other processes and shared objects. At any given time, there may be several transitions (from different processes) concurrently running in the system. The complete dynamic behavior of a system can be specified by the dynamic behavior of each process and shared objects.

Having specified the individual process as a NFA, the concurrent system is defined as Global Finite State Automata(GFSA). It is assumed that there are n processes in the system and the processes can access a finite set of k shared objects.

Definition of Global Finite State Automaton

Let Con $= (P_1, \ldots, P_n, O, \{V_o\}_{o\in O}, S, C, \delta, \nu)$ be a concurrent system. We construct a global automaton $P = (Q, \tilde{\Sigma}, \delta, q_0)$ that model concurrent system as follows:

- $Q = \prod_{i\in I} Q_i \times \prod_{o\in O} V_o$, is a finite set called the *global states*, or simply *states*. An element of Q is $(q_1, \ldots, q_n, v_1, \ldots, v_k)$, where $(q_1, \ldots, q_n, v_1, \ldots, v_k) \in Q \Rightarrow q_1 \in Q_1 \wedge \cdots \wedge q_n \in Q_n \wedge v_1 \in V_1 \wedge \ldots \wedge v_k \in V_k$,

- $\tilde{\Sigma} = \prod_{i\in I} \Sigma_i^{\varepsilon}$, [3] is a finite set called the *alphabet*, or *input symbol*,

[3] Σ_i^{ε} represent $\Sigma_i \cup \varepsilon$, $i = 1, \cdots n$

- $q_0 = \{q_1^0, \ldots, q_n^0, v_1^0, \ldots, v_k^0)$, is the initial state, where $q_0^i \in Q_i (i \in \{1 \ldots n\})$ is the initial state of process p_i, and v_i^0 is the initial value of an objects o_i,

- $\delta : Q \times \tilde{\Sigma} \rightarrow 2^Q$ is the transition function.

  The mapping of $\delta : Q \times \tilde{\Sigma} \rightarrow 2^Q$ is quite complicated formally, although intuitively rather obvious. It must take into account the rule for "Element elimination of the sets $S(a)$ and $C(a)$". However, The transition function δ must satisfy following property:

  $\delta((q_1, \ldots, q_n, v_1, \ldots, v_k), (a_1, \ldots, a_n)) = \{(p_1, \ldots, p_n, v'_1, \ldots, v'_k)\}$ [4] $\Rightarrow \tilde{\delta}_i(q_i, \tilde{a}_i) = \{p_i\} \wedge \tilde{a}_i \in \{\varepsilon, a_i, \overline{a}_i\}$

In this scheme, all of the component states and shared objects values are fed back to each other and the transition relation of the global automaton is derived by determining the actual input of each component automaton, depending on the actual states of these machines and objects values. The event dependence of each component event is eliminated through the combination of component event and shared object.

The Elements Elimination of the Sets S, C

Let an element of the set of global events be $A = (a_1, \ldots, a_n)$, where $a_i \in \Sigma_i^\varepsilon (1 \leq i \leq n)$, $S(a_i)$ is the synchronization set of a_i and $C(a_i)$ is the competition set of a_i. Let also $\delta((q_1, \ldots, q_n, v_1, \ldots, v_k), (a_1, \ldots, a_n)) = \{(p_1, \ldots, p_n, v'_1, \ldots, v'_k)\} \Rightarrow \delta_i(q_i, \tilde{a}_i) = \{p_i\} \wedge \tilde{a}_i \in \{\varepsilon, a_i, \overline{a}_i\}$. Let $\vec{v_O} = (v_1, \ldots, v_k) \in V$, we use $\vec{v}_{O'}$ to denote the projection of $\vec{v_O}$ onto $V_{O'}$.

Let $\lambda : \tilde{\Sigma} \rightarrow 2^\Sigma$ be a mapping interpreted as follows. If $A = (a_1, \ldots, a_n) \in \tilde{\Sigma}$, then $\lambda(A) = \{a_1, \ldots, a_n\}$ with the duplicate ε actions removed.

Five cases need to be considered for each individual process's action a_i.

1. If $a_i = \varepsilon$, no element in the sets $S(a_i), C(a_i)$ needs to be eliminated, since they are already empty. The process i executes transition $\tilde{\delta}_i(q_i, \tilde{a}_i) = \tilde{\delta}_i(q_i, \varepsilon) = \{p_i\} = \{q_i\}$.

2. If a_i is an *independent* action, then sets $S(a_i), C(a_i)$ are also empty. The action a_i is evaluated to *enable* status. The process i executes transition $\tilde{\delta}_i(q_i, \tilde{a}_i) = \tilde{\delta}_i(q_i, a_i) = \{p_i\}$.

3. If $S(a_i) \neq \emptyset \wedge C(a_i) = \emptyset$, all elements of set $S(a_i)$ need to be eliminated to execute a joint transition.

[4] v'_i is the value of object i after execute the transition

Two cases need to be considered. First, if $S(a_i) \subseteq \lambda(A)$, which means that the synchronization actions of a_i are exactly the sub-actions of the global action A with respect to individual processes, then all the elements of set $S(a_i)$ are eliminated. The action a_i is evaluated to *enable* status. The process i executes transition $\tilde{\delta}_i(q_i, \tilde{a}_i) = \tilde{\delta}_i(q_i, a_i) = \{p_i\}$.

Secondly, if $S(a_i) \nsubseteq \lambda(A)$, which means that the synchronization actions of a_i are not all appeared as the sub-actions of the global action A with respect to its corresponding processes, then some elements of set $S(a_i)$ cannot be eliminated. The action a_i is evaluated to *disable* status. Therefore, the process i executes transition $\tilde{\delta}_i(q_i, \tilde{a}_i) = \tilde{\delta}_i(q_i, \overline{a_i}) = \{p_i\}$.

4. If $S(a_i) = \emptyset \wedge C(a_i) \neq \emptyset$, then the elements of set C_i need to be eliminated to possibly execute only one action of set $C \cup \{a_i\}$ depending on the value of the shared objects. Supposed that a_i competes with other processes to access shared objects O' on set of value $\nu(a_i)$. If $\vec{v}_{O'} \notin \nu(a_i)$, then the elements elimination is irrelevant since the shared objects values do not satisfy the competing condition. The transition of process i follows the definition of δ function.

   However, if $\vec{v}_{O'} \in \nu(a_i)$, then two cases need to be considered for elements elimination of set $C(a_i)$

   First, if $C(a_i) \wedge \lambda(A) = \emptyset$, which means that in the global action A, the compete actions of action a_i do not appear as sub-actions of A with respect to the corresponding processes. All the elements in the competition set of a_i are eliminated. The action a_i is evaluated to *enable*. Hence the process i executes transition $\tilde{\delta}_i(q_i, \tilde{a}_i) = \tilde{\delta}_i(q_i, a_i) = \{p_i\}$

   Secondly, if $C(a_i) \wedge \lambda(A) \neq \emptyset$ which means that action a_i competes with other actions to access the shared objects O'. Only one of competing actions is evaluated to *enable*, and other actions in the set $C(a_i) \cup \{a_i\}$ are evaluated to *disable*. The action i is either evaluated to *enable* and process i executes transition $\delta_i(q_i, \tilde{a}_i) = \delta_i(q_i, a_i) = \{p_i\}$, or the action i is evaluated to *disable* and process i executes transition $\tilde{\delta}_i(q_i, \tilde{a}_i) = \tilde{\delta}_i(q_i, \overline{a_i}) = \{p_i\}$. It is arbitrary that the action i is evaluated to *enable* or *disable* status if we do not take into account the priority of competing actions.

5. If $S(a_i) \neq \emptyset \wedge C(a_i) \neq \emptyset$, careful consideration is needed. Three cases need to be considered for such a situation.

   First, if $S(a_i) \nsubseteq \lambda(A)$, which means that the synchronization actions of a_i are not all appeared as the sub-actions of the global action A with respect to the corresponding processes, then some elements of set $S(a_i)$ cannot be eliminated.

The action a_i is evaluated to *disable* status. Therefore, the process i executes transition $\tilde{\delta}_i(q_i, \tilde{a}_i) = \tilde{\delta}_i(q_i, \overline{a_i}) = \{p_i\}$.

Secondly, if $S(a_i) \subseteq \lambda(A)$, $\vec{v}_{O'} \notin \nu(a_i)$, then for the reason explained above, this case is irrelevant for elements elimination of sets $S(a_i)$ and $C(a_i)$.

Thirdly, if $S(a_i) \subseteq \lambda(A)$, $\vec{v}_{O'} \in \nu(a_i)$, which means that the synchronization actions of a_i are exactly the sub-actions of the global action A with respect to individual processes, then all the elements of set $S(a_i)$ are eliminated. It also means that action a_i compete with other actions to access the shared objects O'. Only one of competing actions is evaluated to *enable*, and other actions in the set $C(a_i) \cup \{a_i\}$ are evaluated to *disable*. The action i is either evaluated to *enable* and process i executes transition $\tilde{\delta}_i(q_i, \tilde{a}_i) = \tilde{\delta}_i(q_i, a_i) = \{p_i\}$, or The action i is evaluated to *disable* and process i executes transition $\tilde{\delta}_i(q_i, \tilde{a}_i) = \tilde{\delta}_i(q_i, \overline{a_i}) = \{p_i\}$.

Put it together, the transition function δ should satisfy the following property: $\delta((q_1, \ldots, q_n, v_1, \ldots, v_k), (a_1, \ldots, a_n)) = \{(p_1, \ldots, p_n, v'_1, \ldots, v'_k)\} \Rightarrow \tilde{\delta}_i(q_i, \tilde{a}_i) = \{p_i\} \wedge \tilde{a}_i \in \{\varepsilon, a_i, \overline{a}_i\}$ and

$$\tilde{\delta}_i(q_i, \tilde{a}_i) = \begin{cases} \tilde{\delta}_i(q_i, \varepsilon) & \text{if } a = \varepsilon, \\ \tilde{\delta}_i(q_i, a_i) & \text{if } (S(a_i) = \emptyset) \wedge (C(a_i) = \emptyset), \\ \tilde{\delta}_i(q_i, a_i) & \text{if } (S(a_i) \neq \emptyset) \wedge (C(a_i) = \emptyset) \wedge (S(a_i) \subseteq \lambda(A)), \\ \tilde{\delta}_i(q_i, \overline{a}_i) & \text{if } (S(a_i) \neq \emptyset) \wedge (C(a_i) = \emptyset) \wedge (S(a_i) \nsubseteq \lambda(A)), \\ \tilde{\delta}_i(q_i, a_i) & \text{if } (S(a_i) = \emptyset) \wedge (C(a_i) \neq \emptyset) \wedge (\vec{v}_{O'} \in \nu(a_i)) \wedge (C(a_i) \wedge \lambda(A) = \emptyset), \\ \tilde{\delta}_i(q_i, \overline{a}_i) & \text{if } (S(a_i) = \emptyset) \wedge (C(a_i) \neq \emptyset) \wedge (\vec{v}_{O'} \in \nu(a_i)) \wedge (C(a_i) \wedge \lambda(A) \neq \emptyset), \\ \cup \tilde{\delta}_i(q_i, a_i) & \\ \tilde{\delta}_i(q_i, \overline{a}_i) & \text{if } (S(a_i) \neq \emptyset) \wedge (C(a_i) \neq \emptyset) \wedge (S(a_i) \nsubseteq \lambda(A)), \\ \tilde{\delta}_i(q_i, \overline{a}_i) & \text{if } (S(a_i) \neq \emptyset) \wedge (C(a_i) \neq \emptyset) \wedge (S(a_i) \subseteq \lambda(A)) \wedge (\vec{v}_{O'} \in \nu(a_i)), \\ \cup \tilde{\delta}_i(q_i, a_i) & \end{cases}$$

From the definition of individual process i, one may find that action $\overline{a_i}$ upon certain state leads the process to the state itself, and action ε upon any local states leads the process to the state itself too. One may immediately asked "Why don't you just use ε action to substitute all the $\overline{a}$, and simplify the transition function table". Careful readers may find that it is not necessary the case. For example, for a global action A upon certain global state, if the action a_i is evaluated to *disable* status, then according to our scheme, upon certain state, it leads the process to the originals state itself; Upon other states, it may leads the processes to error if a_i is not defined in these states. If all the co-action of $\overline{a}$ is substituted by ε action in the transition function, then upon any local state, it leads the process to the originals state itself, which is wrong. It is our intention to capture the illegal global events through local transition function, Therefore $\overline{a}$ action and ε action is not equivalent, it cannot be substituted by ε action.

Base on the above scheme, the global finite state automaton of Producer-Consumer concurrent system is defined as NFA $P = (Q, \tilde{\Sigma}, \delta, q_0)$, where

- $Q = Q_1 \times Q_2 \times V_{buf}$ is a finite set called the *global states*, or simply *states*, and $Q = \{(s_1, s_3, 0), (s_1, s_3, 1), (s_1, s_3, 2), (s_1, s_4, 0), (s_1, s_4, 1), (s_1, s_4, 2), (s_2, s_3, 0), (s_2, s_3, 1), (s_2, s_3, 2), (s_2, s_4, 0), (s_2, s_4, 1), (s_2, s_4, 2)\}$

- $\tilde{\Sigma} = \Sigma_1^{\varepsilon} \times \Sigma_2^{\varepsilon}$ is a finite set called the *alphabet*, or *input symbol*, and $\tilde{\Sigma} = \{(t_1, t_3), (t_1, t_4), (t_1, \varepsilon), (t_2, t_3), (t_2, t_4), (t_2, \varepsilon), (\varepsilon, t_3), (\varepsilon, t_4), (\varepsilon, \varepsilon)\}$.

- $\delta : Q \times \tilde{\Sigma} \to 2^Q$ is the *transition function*, $\delta((q_1, q_2, v_{buf}), (a_1, a_2)) = (\tilde{\delta_1}(q_1, \tilde{a}_1), \tilde{\delta_2}(q_2, \tilde{a}_2), v'_{buf})$ where $(q_1, q_2, v_{buf}) \in Q$, and $\tilde{a}_i \in \{\varepsilon, a_i, \overline{a_i}\}$ where $a_i \in \Sigma_i$. The transition function/relation is defined as Table 4.3 in traditional way.

- $q_0 = (s_1, s_3, 0)$ is the initial state, where $s_1 \in Q_1$ is the initial state of producer, $s_3 \in Q_2$ is the initial state of consumer, 0 is the initial value of shared object *buf*.

The transition function of Producer-Consumer is defined as follows. The initial state of Producer-Consumer system is $(s_1, s_3, 0)$, which means that producer is in "ready to produce product" state, consumer is in "ready to consume product" state and the value of buffer is 0 (no product in the buffer). At the initial state, if the global transition event is (t_1, t_3), since consumer cannot consume any product when the value of buffer is 0, then t_1 is evaluated to *enable* status, t_3 is evaluated to *disable* status, and $\delta((s_1, s_3, 0), (t_1, t_3) = \{(\tilde{\delta}_1(s_1, t_1), \tilde{\delta}_2(s_3, \overline{t}_3), 1)\}$. Similarly, $\tilde{\delta}((s_1, s_3, 0), (t_1, \varepsilon)) = \{(\tilde{\delta}_1(s_1, t_1), \tilde{\delta}_2(s_3, \varepsilon), 1)\}$, and $\delta((s_1, s_3, 0), (\varepsilon, t_3)) = \{(\tilde{\delta}_1(s_1, \varepsilon), \tilde{\delta}_2(s_3, \overline{t}_3), 0)\}$.

At state $(s_1, s_3, 1)$, which represents that producer is in "ready to produce product" state and consumer is in "ready to consume" state, and the value of buffer is 1. If the global transition event is (t_1, t_3), since the value of buffer is $1 \in \nu(t_1)$ and $1 \in \nu(t_3)$, the producer and consumer have equal right to access the share object, but only one process can get the change to access it. Therefore, the set of next states is defined as $\delta((s_1, s_3, 1), (t_1, t_3)) = \{(\tilde{\delta}_1(s_1, t_1), \tilde{\delta}_2(s_3, \overline{t}_3), 2), (\tilde{\delta}_1(s_1, \overline{t}_1), \tilde{\delta}_2(s_3, t_3), 0)\}$. The rest terms are defined similarly as above. The final result of transition function after being evaluated from process's local transition function is shown in Table 4.4.

There are two processes and one share object involved in the Producer-Consumer system, each process has two states and the object has three values, hence transition table needs $2 \times 2 = 12$ rows. Each process has two actions and a ε action, hence the table needs 9 columns to specify the global actions. It has been shown that the table size increases exponentially.

Let us consider another system, called weird system. A weird software system consists of two processes P_1, P_2 and one shared object o. P_1 has three events, namely

δ	(t_1,t_3)	(t_1,t_4)	(t_1,ε)	(t_2,t_3)	(t_2,t_4)	(t_2,ε)	(ε,t_3)	(ε,t_4)	$(\varepsilon,\varepsilon)$
$(s_1,s_3,0)$	$\{(\tilde{\delta}_1(s_1,t_1),\tilde{\delta}_2(s_3,\bar{t}_3),1)\}$		$\{(\tilde{\delta}_1(s_1,t_1),s_3,1)\}$				$\{(s_1,\tilde{\delta}_2(s_3,\bar{t}_3),0)\}$		$\{(s_1,s_3,0)\}$
$(s_1,s_3,1)$	$\{(\tilde{\delta}_1(s_1,t_1),\tilde{\delta}_2(s_3,\bar{t}_3),2),$		$\{(\tilde{\delta}_1(s_1,t_1),s_3,2)\}$				$\{(s_1,\tilde{\delta}_2(s_3,t_3),1)\}$		$\{(s_1,s_3,1)\}$
	$(\tilde{\delta}_1(s_1,\bar{t}_1),\tilde{\delta}_2(s_3,t_3),0)\}$								
$(s_1,s_3,2)$	$\{(\tilde{\delta}_1(s_1,\bar{t}_1),\tilde{\delta}_2(s_3,t_3),1)\}$		$\{(\tilde{\delta}_1(s_1,\bar{t}_1),s_3,2)\}$				$\{(s_1,\tilde{\delta}_2(s_3,t_3),1)\}$		$\{(s_1,s_3,2)\}$
$(s_1,s_4,0)$		$\{(\tilde{\delta}_1(s_1,t_1),$						$\{(s_1,\tilde{\delta}_2(s_4,t_4),0)\}$	$\{(s_1,s_4,0)\}$
$(s_1,s_4,0)$		$\tilde{\delta}_2(s_4,t_4),1)\}$							$\{(s_1,s_4,0)\}$
$(s_1,s_4,1)$		$\{(\tilde{\delta}_1(s_1,t_1),$						$\{(s_1,\tilde{\delta}_2(s_4,t_4),1)\}$	$\{(s_1,s_4,1)\}$
$(s_1,s_4,1)$		$\tilde{\delta}_2(s_4,t_4),2)\}$							$\{(s_1,s_4,1)\}$
$(s_1,s_4,2)$		$\{(\tilde{\delta}_1(s_1,\bar{t}_1),$						$\{(s_1,\tilde{\delta}_2(s_4,t_4),2)\}$	$\{(s_1,s_4,2)\}$
$(s_1,s_4,2)$		$\tilde{\delta}_2(s_4,t_4),2)\}$							$\{(s_1,s_4,2)\}$
$(s_2,s_3,0)$				$\{(\tilde{\delta}_1(s_2,t_2),$		$\{(\tilde{\delta}_1(s_2,t_2),s_3,0)\}$	$\{(s_2,\tilde{\delta}_2(s_3,\bar{t}_3),0)\}$		$\{(s_2,s_3,0)\}$
$(s_2,s_3,1)$				$\tilde{\delta}_2(s_4,\bar{t}_4),0)\}$		$\{(\tilde{\delta}_1(s_2,t_2),s_3,1)\}$	$\{(s_2,\tilde{\delta}_2(s_3,t_3),0)\}$		$\{(s_2,s_3,1)\}$
$(s_2,s_3,2)$				$\{(\tilde{\delta}_1(s_2,t_2),$		$\{(\tilde{\delta}_1(s_2,t_2),s_3,2)\}$	$\{(s_2,\tilde{\delta}_2(s_3,t_3),1)\}$		$\{(s_2,s_3,2)\}$
				$\tilde{\delta}_2(s_4,t_4),0)\}$					
				$\{(\tilde{\delta}_1(s_2,t_2),$					
				$\tilde{\delta}_2(s_4,t_4),1)\}$					
$(s_2,s_4,0)$					$\{(\tilde{\delta}_1(s_2,t_2),$	$\{(\tilde{\delta}_1(s_2,t_2),s_4,0)\}$		$\{(s_2,\tilde{\delta}_2(s_4,t_4),0)\}$	$\{(s_2,s_4,0)\}$
$(s_2,s_4,1)$					$\tilde{\delta}_2(s_4,t_4),0)\}$	$\{(\tilde{\delta}_1(s_2,t_2),s_4,1)\}$		$\{(s_2,\tilde{\delta}_2(s_4,t_4),1)\}$	$\{(s_2,s_4,1)\}$
$(s_2,s_4,2)$					$\{(\tilde{\delta}_1(s_2,t_2),$	$\{(\tilde{\delta}_1(s_2,t_2),s_4,2)\}$		$\{(s_2,\tilde{\delta}_2(s_4,t_4),2)\}$	$\{(s_2,s_4,2)\}$
					$\tilde{\delta}_2(s_4,t_4),1)\}$				
					$\{(\tilde{\delta}_1(s_2,t_2),$				
					$\tilde{\delta}_2(s_4,t_4),2)\}$				

Table 4.3: The Global transition Function of Producer-Consumer System

δ	(t_1, t_3)	(t_1, t_4)	(t_1, ε)	(t_2, t_3)	(t_2, t_4)	(t_2, ε)	(ε, t_3)	(ε, t_4)	$(\varepsilon, \varepsilon)$
$(s_1, s_3, 0)$	$\{(s_2, s_3, 1)\}$		$\{(s_2, s_3, 1)\}$				$\{(s_1, s_3, 0)\}$		$\{(s_1, s_3, 0)\}$
$(s_1, s_3, 1)$	$\{(s_2, s_3, 2), (s_1, s_4, 1)\}$		$\{(s_2, s_3, 2)\}$				$\{(s_1, s_4, 0)\}$		$\{(s_1, s_3, 1)\}$
$(s_1, s_3, 2)$	$\{(s_1, s_4, 1)\}$		$\{(s_1, s_3, 2)\}$				$\{(s_1, s_4, 1)\}$		$\{(s_1, s_3, 2)\}$
$(s_1, s_4, 0)$		$\{(s_2, s_3, 1)\}$						$\{(s_1, s_3, 0)\}$	$\{(s_1, s_4, 0)\}$
$(s_1, s_4, 1)$		$\{(s_2, s_3, 2)\}$						$\{(s_1, s_3, 1)\}$	$\{(s_1, s_4, 1)$
$(s_1, s_4, 2)$		$\{(s_1, s_3, 2)\}$						$\{(s_1, s_3, 2)\}$	$\{(s_1, s_4, 2)\}$
$(s_2, s_3, 0)$				$\{(s_1, s_3, 0)\}$		$\{(s_1, s_3, 0)\}$	$\{(s_2, s_3, 0)\}$		$\{(s_2, s_3, 0)\}$
$(s_2, s_3, 1)$				$\{(s_1, s_4, 0)\}$		$\{(s_1, s_3, 1)\}$	$\{(s_2, s_4, 0)\}$		$\{(s_2, s_3, 1)\}$
$(s_2, s_3, 2)$				$\{(s_1, s_4, 1)\}$		$\{(s_1, s_3, 2)\}$	$\{(s_2, s_4, 1)\}$		$\{(s_2, s_3, 2)\}$
$(s_2, s_4, 0)$					$\{(s_1, s_3, 0)\}$	$\{(s_1, s_4, 0)\}$		$\{(s_2, s_3, 0)\}$	$\{(s_2, s_4, 0)\}$
$(s_2, s_4, 1)$					$\{(s_1, s_3, 1)\}$	$\{(s_1, s_4, 1)\}$		$\{(s_2, s_3, 1)\}$	$\{(s_2, s_4, 1)\}$
$(s_2, s_4, 2)$					$\{(s_1, s_3, 2)\}$	$\{(s_1, s_4, 2)\}$		$\{(s_2, s_3, 2)\}$	$\{(s_2, s_4, 2)\}$

Table 4.4: The Final Result of Producer-Consumer System's transition Function

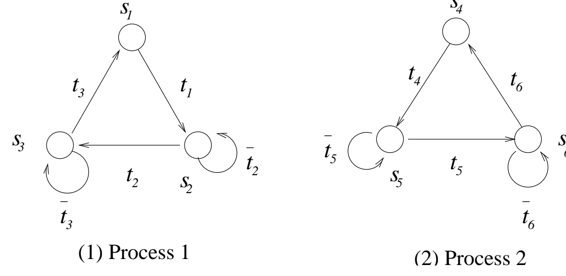

Figure 4.2: The Finite State Automata of Process 1 and Process 2

$\tilde{\delta}_1$	t_1	t_2	$\overline{t}_2$	t_3	$\overline{t}_3$	ε
s_1	$\{s_2\}$					$\{s_1\}$
s_2		$\{s_3\}$	$\{s_2\}$			$\{s_2\}$
s_3				$\{s_1\}$	$\{s_3\}$	$\{s_3\}$

Table 4.5: Transition Function of Process 1

t_1, t_2, t_3, and three states, s_1, s_2, s_3 ; P_1 has three events, namely t_4, t_5, t_6, and three states, s_4, s_5, s_6 ; Object o is shared by the two processes and $(0 \leq v_o \leq k) \wedge (v_o \in int)$ (for some integer k). The initial value of object o is 0.

Event t_1 of process P_1 and event t_4 of process P_2 are both *independent* events; Event t_2 of process P_1 and event t_5 of process P_2 competes for accessing object o. These two events have equal priority to access the object o when $0 < v_o < k$, namely $C(t_2) = \{t_5\}$, $C(t_5) = \{t_2\}$ and $\nu(a_2) = \nu(a_2) = \{v_o | 0 < v_o < k\}$. Event t_2 adds 1 to o if $0 \leq v_o < k$. Event t_5 subtract 1 from o if $0 < v_o \leq k$; Event t_3 and event t_6 are joint events, namely $S(t_3) = \{t_6\}$ and $S(t_6) = \{t_3\}$ and they synchronize to transact simultaneously. Therefore, P_1 and P_2 are defined as follows.

P_1 is modeled as $P_1 = (Q_1, \Sigma_1, \tilde{\delta}_1, s_1)$, depicted as left of Figure 4.2, where

1. $Q_1 = \{s_1, s_2, s_3\}$ is a finite set of states,
2. $\Sigma_1 = \{t_1, t_2, t_3\}$ is finite set of *alphabet*, or *events*.
3. $\tilde{\delta}_1 : Q_1 \times \Sigma_1 \rightarrow 2^Q_1$ is the *transition function* defined as table 4.5, and
4. $s_1 \in Q_1$ is the initial state.

P_2 is modeled as $P_2 = (Q_2, \Sigma_2, \tilde{\delta}_2, s_4)$, depicted as right of figure 4.2, where

1. $Q_2 = \{s_4, s_5, s_6\}$ is a finite set of states,
2. $\Sigma_2 = \{t_4, t_5, t_6\}$ is finite set of *alphabet*, or *events*.
3. $\tilde{\delta}_2 : Q_2 \times \Sigma_2 \rightarrow 2^Q_2$ is the *transition function* defined as table 4.6, and
4. $s_4 \in Q_2$ is the initial state.

$\tilde{\delta}_2$	t_4	t_5	$\overline{t}_5$	t_6	$\overline{t}_6$	ε
s_4	$\{s_5\}$					$\{s_4\}$
s_5		$\{s_6\}$	$\{s_5\}$			$\{s_5\}$
s_6				$\{s_4\}$	$\{s_6\}$	$\{s_6\}$

Table 4.6: Transition Function of Process 2

The global finite state automata of weird system is defined as $P = (Q, \tilde{\Sigma}, \delta, q_0)$, where

1. $Q = Q_1 \times Q_2 \times V_o$ is a finite set called the *global states*, or simply *states*.
2. $\tilde{\Sigma} = \Sigma_1^\varepsilon \times \Sigma_2^\varepsilon$ is a finite set called the *alphabet*, or *input symbol*,
3. $\delta : Q \times \tilde{\Sigma} \to 2^Q$ is the *transition function*, $\delta((q_1, q_2, v_o), (a_1, a_2)) = (\tilde{\delta_1}(q_1, \tilde{a}_1), \tilde{\delta_2}(q_2, \tilde{a}_2), v'_{buf})$ where $(q_1, q_2, v_o) \in Q$, and $\tilde{a}_i \in \{\varepsilon, a_i, \overline{a_i}\}$ where $a_i \in \Sigma_i$. 4. $q_0 = (s_1, s_4, 0) \in Q$ is the initial state.

Let an element of a set of global events be $A = (a_1, a_2)$, where $a_1 \in \Sigma_1$ is an event of process 1, $a_2 \in \Sigma_2$ is an event of process 2. An element of a set of global state be $(q_1, q_2, v_o) \in Q$, where $q_1 \in Q_1 \wedge q_2 \in Q_2$ and $v_o \in V_O$.

A few interested global events need careful consideration in this system.

- The global event is $A = (t_2, t_5)$, which means the event of process 1 and the event of process 2 compete to access the object o. When $0 < v_o < k$, since t_2, t_5 have equal priority to access the object o, then either $q_1' = \tilde{\delta}_1('q_1, t_2) \wedge q_2' = \tilde{\delta}_2('q_2, \overline{t}_5) \wedge v_o' = 'v_o + 1)$ [5] or $q_1' = \tilde{\delta}_1('q_1, \overline{t}_2) \wedge q_2' = \tilde{\delta}_2('q_2, t_5) \wedge v_o' = 'v_o - 1)$ will be true, but not both. When $v_o = 0$, process 2 cannot execute action t_5, then $q_1' = \tilde{\delta}_1('q_1, t_2) \wedge q_2' = \tilde{\delta}_2('q_2, \overline{t}_5) \wedge v_o' = 1)$ is true; On the other hand, when $v_o = k$, process 1 cannot execute action t_2, then $q_1' = \tilde{\delta}_1('q_1, \overline{t}_2) \wedge q_2' = \tilde{\delta}_2('q_2, t_5) \wedge v_o' = k)$ is true,

- The global event $A = (t_3, t_6)$, which means that the event of process 1 synchronizes with the event of process 2 to jointly execute a transition. Since the global action satisfies the system constraints, the transition $\tilde{\delta}_1('q_1, a_1) \wedge \tilde{\delta}_2('q_2, a_2)$ is executed. If global event is $A = (a_1, a_2)$ and $a_1 = t_3 \wedge a_2 \neq t_6$, $\tilde{\delta}_1('q_1, \overline{a}_1)$ will be executed with respect to process 1 ; Similarly if global event is $A = (a_1, a_2)$, and $a_1 \neq t_3 \wedge a_2 = t_6$, $\tilde{\delta}_2('q_2, \overline{a}_2)$ will be executed with respect to process 2.

[5] The same convention as traditional tabular expressions is used, namely if a represents the value of a variable (or state of a process), then $'a$ represents the value of variable (or state of a process) before the transition, and a' is used to represents the value of variable (or state of process) after the transition.

There are two processes and one share object involved in the weird software system, each process has 3 states and the object has k values, hence transition table needs $3 \times 3 \times k = 6k$ rows. Each process has 3 actions and a ε action, hence the table needs $(3+1) * (3+1) = 16$ columns to specify the global actions. If the value of k is relatively big, then the table will need many rows to specify the transition. Therefore, it is not practical to use traditional transition table to specify it. The transition function will be specified using tabular expression in next section.

4.3 Tabular Expression

Finite automata are frequently given by means of *transition tables*. If $M = (Q, \Sigma, \delta, q_0)$ is an automaton with m states and n events symbols, then table M has m rows and n columns. For practical software system, the value of m and n could be large, especially when combining different processes together, the events and states will be increased exponentially, therefore it is not practical to use the conventional transition table to specify global automata. However, the formal specification notation, tabular expressions can be used to resolve this problem.

Here, the meanings of tabular expressions used by SERG is slightly extended. Functional method uses tabular expressions to specify Input/Output relation of sequential program execution. For the concurrent program, the situation is a little more delicate. One of the reasons for this lies in the fact that such programs do not have inputs and outputs defined as easily as those for sequential programs. In this work, the concurrent system is viewed as a global finite state automaton, and the tabular expression is used to specify the transition function of the global automaton.

The two-dimensional table [11, 12, 23] will be used to write the transition function of concurrent systems. The predicate in the first column (called header H_1) /row of each table partitions the global state into mutually exclusive set. i.e. for a given state, only one row in each table can be evaluated as "true". The predicate in the first row (called header H_2) /column of table partitions the global events into mutually exclusive set. i.e. for a given set of input event, only one column in each table can be evaluated as "true". The cell in the main *grid* is the expressions of set of next states in term of each process local transition function and the object next value.

The Tabular Expressions of Producer-Consumer system

The transition function of producer and consumer system is specified as follows. Let an element of a set of global events be $A = (a_1, a_2)$, where $a_1 \in \Sigma_1$ is an event of producer, $a_2 \in \Sigma_2$ is an event of consumer; An element of a set of global state be $(q_1, q_2, v_{buf}) \in Q$, where $q_1 \in Q_1 \wedge q_2 \in Q_2$. The transition function of Producer and Consumer system is defined as tabular form depicted as Table 4.7.

δ	$(a_1 = t_1, a_2 = t_3)$	$(a_1 = t_1, a_2 \neq t_3)$	$(a_1 \neq t_1, a_2 = t_3)$	$(A_1 \neq t_1, A_2 \neq t_3)$
$'v_{buf} = 0$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), 1)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 1)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), 0)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 0)\}$
$'v_{buf} = 1$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), 2),$ $(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), 0)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 2)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 0)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 1)\}$
$'v_{buf} = 2$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), 1)\}$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), 2)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_3), 1)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 2)\}$

Table 4.7: The Tabular Expressions of Producer-Consumer System's Transition Function

δ	$'v_o = 0$	$0 < 'v_o < k$	$'v_o = k$
$(a_1 = t_1, a_2 = t_4)\cup$ $(a_1 = \varepsilon, a_2 = t_4)\cup$ $(a_1 = t_1, a_2 = \varepsilon)\cup$ $(a_1 = t_3, a_2 = t_6)$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 0)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), v_o^{'} = 'v_o\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), k)\}$
$(a_1 = t_1, a_2 = t_5)\cup$ $(a_1 = \varepsilon, a_2 = t_5)$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), 0)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), v_o^{'} = 'v_o - 1)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), k - 1)\}$
$(a_1 = t_1, a_2 = t_6)\cup$ $(a_1 = \varepsilon, a_2 = t_6)$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), 0)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), v_o^{'} = 'v_o)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), k)\}$
$(a_1 = t_2, a_2 = t_4)\cup$ $(a_1 = t_2, a_2 = \varepsilon)$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 1)\}\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), v_o^{'} = 'v_o + 1))\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), k)\}$
$(a_1 = t_2, a_2 = t_5$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), 1)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), v_o^{'} = 'v_o + 1),$ $(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), v_o^{'} = 'v_o - 1)\}$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), k - 1)\}$
$(a_1 = t_2, A_2 = t_6)$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, \overline{a}_2), 0)\}$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, \overline{a}_2), v_o^{'} = 'v_o)\}$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, \overline{a}_2), k)\}$
$(a_1 = t_3, A_2 = t_4)\cup$ $(a_1 = t_3, \varepsilon)$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), 0)\}$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), v_o^{'} = 'v_o)\}$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), k)\}$
$(a_1 = t_3, A_2 = t_5)$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, \overline{a}_2), 0)\}$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), v_o^{'} = 'v_o - 1)\}$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), k - 1)\}$

Table 4.8: The Tabular Expressions of Weird System's Transition Function

The Tabular Expressions of Weird system

Let an element of a set of global events be $A = (a_1, a_2)$, where $a_1 \in \Sigma_1$ is an event of process 1, $a_2 \in \Sigma_2$ is an event of process 2; An element of a set of global state be $(q_1, q_2, v_o) \in Q$, where $q_1 \in Q_1 \wedge q_2 \in Q_2$.

The transition function of the Weird system is defined as tabular form depicted as Table 4.8. Header H_1 of the table is the predicate of global actions which partitions the global events into mutually exclusive set; Header H_2 is the predicate of global state which partitions the global states into mutually exclusive sets.

It can be observed from the Producer-Consumer system and the weird system that the tabular expressions can significantly reduce the number of rows and columns needed to specify the transition function. It is especially true for the loosely coupled system, since the global state of this kind of system can be partitioned easily to few sections.

δ	$(a_1 = t_1, a_2 = t_3)$	$(a_1 = t_1, a_2 \neq t_3)$	$(a_1 \neq t_1, a_2 = t_3)$	$(a_1 \neq t_1, a_2 \neq t_3)$
$'v_{buf} = 0$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), 1)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 1)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, \overline{a}_2), 0)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 0)\}$
$'v_{buf} = 1$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), 0),$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 2)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 0)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 1)\}$
$'v_{buf} = 2$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), 1)\}$	$\{(\delta_1('q_1, \overline{a}_1), \delta_2('q_2, a_2), 2)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_3), 1)\}$	$\{(\delta_1('q_1, a_1), \delta_2('q_2, a_2), 2)\}$

Table 4.9: The Tabular Expressions of Producer-Consumer System's Transition Function with Priority

4.3.1 Processes' Priority

Priority is considered as the most important factor and it should be dealt with before anything else. In the specification of concurrent computing system, priority is an important property needed to be addressed. In a practical computing system, such needs do exist when some processes' activities have high priority. For example, in the Producer-Consumer system, when the value of buffer is $v_{buf} = 1$, if the global event is (t_1, t_3), then the producer and consumer have equal right to access the buffer.

However, for whatever reasons, if it is desirable to give some events a high priority, it is nice to have that kind of mechanism to do so. This could be done easily by adding additional symbol to process's events, namely adding *+,* - symbols to the events of competition set of a competing event.

Let t_1 be an event of process 1, and t_1 competes with an event t_2 of process 2 and an event t_3 of process 3 to access an shared object o on a set of values V'_o. t_2 has a higher priority than t_1 while t_3 has a lower priority than t_1. According to our approach, the competition set of event t_1 is written as $C(t_1) = \{+t_2, -t_3\}$.

In the Producer-Consumer system, if the consumer is given a higher priority than producer, then the competition set of t_1 is $C(t_1) = \{+t_3\}$, and t_3 is $C(t_3) = \{-t_1\}$. The transition function of Producer and Consumer system, with consumer having a higher priority, is defined as tabular form depicted as Table 4.9.

There is a difference between the two specifications of Producer-Consumer system. In the first one, producer and consumer have equal right to access the buffer when the buffer value is 1. Hence, on the global event (t_1, t_3), there are two possibilities for the next state. In the second one, since the consumer has a higher priority, there is only one choice for the next state, namely consumer consumes the product and enters into the next state and producer stays in the same state.

4.3.2 The Equivalent Relation of Two Specification

After specifying the transition function of concurrent systems with tabular expressions, it is nature to ask "whether the tabular form specification is equivalent to traditional counterpart". This is the fundamental principle that needs to be established. Moreover, for practice specification of concurrent system, if this principle

cannot be guaranteed, then it is an incorrect specification of underlay system.

It can be proved that the tabular expressions specification of Producer-Consumer system is equivalent to the traditional one. It can be established a *onto* mapping from traditional specification to the tabular one. Moreover, for every element of partitioned set in tabular form, the corresponding element can be matched into the traditional transition table. For example, when the global state is $(s_1, s_3, 0)$, the global event is (t_1, t_3), the next state is defined as $\{(\delta_1(s_1, t_1), \delta_2(s_3, \bar{t}_3), 1)\}$ in the traditional transition function table, and this term can be mapped to the tabular specification term, namely the term $('v_{buf} = 0 \wedge a_1 = t_1, a_2 = t_3) \Rightarrow \{(\delta_1(s_1, t_1), \delta_2(s_3, \bar{t}_3), 1)\}$. However, the later term defines more terms than the traditional one. It also includes global states $(s_1, s_4, 0)$, $(s_2, s_3, 0)$, $(s_2, s_4, 0)$ on global events (t_1, t_3). These three cases lead the global state to error, which is also true for the traditional transition function table. These arguments can be applied to the other terms of the specifications. Therefore, it is safe to claim that the two specifications are equivalent if the tabular expression approach is used correctly.

4.3.3 The Properties of Tabular Form of Concurrent System Specification

An *invariant* is a property or set of properties that must always be true for any state to which the system may evolve. The crucial and important activity in building any specification is the identification and specification of invariants for the underlay system. It will be shown in the example that the invariants can be maintained by the tabular expressions.

It can also be proved from the tabular expression that the specification is deadlock free. This properties will be showed in the specification of Reader-Writer system on next chapter.

It should be noted that the global actions is modeled as a tuple with individual process actions as elements, including possible empty actions from each process in the tabular specification. Hence, it is obvious that global actions is the joint effort of individual process actions, which means that all the processes activities can be processed simultaneously with the specification.

Following the formal definition of "global" automata, it is obvious that the concurrent system starts at its initial state, and executes global transition which consists of each individual process action (possibly ε action at process level and system level) as an element. At each global state, on given global event the concurrent system executes transition by following the transition function, which is based on the system restriction and system properties. The computational result of composite automata is the total history of underlay concurrent system events.

Chapter 5

An Example of Readers-Writers System

In this chapter, an example of Readers-Writers problems is given to illustrate how to apply this approach.

5.1 Readers-Writers Problems

The Readers-Writers problem is associated with accessing to a shared database by two kinds of processes, namely readers and writers. Readers execute transitions and examine database while writers examine and update database. To update a database correctly, writers must have an exclusive right to access the database while they are updating it. If no writers are accessing the database, any numbers of readers can concurrently access it. In this chapter, tabular expressions are used to specify and document a program that must satisfy these properties.

Before writing documentation for the Readers-Writers problem, it is necessary to establish a set of policies that govern their usage.

The following situations are considered:

- If there are already one or more active (executing) readers, can a newly arriving reader immediately joins them even there is also an acquiring writer?
- If some readers as well as some writers are trying to access the database while database is free (no readers and writers are accessing the database). Should a bias policy be specified toward readers or writers?

The basic policy to specify is that whenever there are already one or more active (executing) readers, a newly arriving reader can immediately join them even there

is a writer waiting to access the database. However, when database is free, either readers or one writer can enter it, but not simultaneously.

It is assumed that there are n readers and m writers in the system. Let $R_i(1 \leq i \leq n)$ represent a reader process i, and $W_j(1 \leq j \leq m)$ represent a writer process j. In the system, each reader competes with other writers to access the database, and each writer competes with other readers plus other writers to access the database when database is free.

5.1.1 Finite State Automata for Individual Process

Each of readers and writers is modeled as a NFA, respectively, and the database is modeled as an object, named D. The value of D is modeled as a set Γ. The elements of the set are the readers processes and writers processes. If a process's name is in the set Γ, it means that the corresponding process is accessing the database. When database is free, $\Gamma = \emptyset$. The set of values of database is represented by V_D. Due to the characteristics of Readers-Writers system, at most one writer process can be in the set Γ or any numbers of readers can be in the set Γ at any given time. Moreover, when one writer is in the set Γ, other writers and readers cannot be in the set.

Notations

A set of symbols are used to represent both states and actions of individual process.

- The first capital letter in a string represents the kind of process it belongs to. For example, R represents a reader, and W indicates a writer.
- *REW*: Reader enters the waiting state of reading.
- *RAq*: Reader acquires for accessing database.
- *RRl*: Reader releases the accessed database.
- *R.Lp*: Reader is in a local processing state without interfering with the database.
- *R.Wt*: Reader is in the waiting state.
- *R.Rd*: Reader is in the state of reading database.
- *WEW* : Writer enters the waiting state for writing.
- *WAq* : Writer acquires for updating database.
- *WRl* : Writer releases the updated database.

- $W.Lp$: Reader is in the local processing state, without interfering with database.
- $W.Wt$: Writer is in the waiting state.
- $W.Wr$: Writer is in the updating database state.

Figure 5.1: The Finite State Automata of Individual Reader and Writer

In this model, each process is modeled as a nondeterministic finite state automaton, Figure 5.1 is the automaton of each individual reader and writer, respectively.

The reader i is described as a nondeterministic finite state automaton, as shown in Figure 5.1, formally written as M_i^R=$(Q_i^R, \Delta_i^R, \tilde{\delta}_i^R, R_i.Lp)$, where

1. $Q_i^R = \{R_i.Lp, R_i.Wt, R_i.Rd\}$,
2. $\Delta_i^R = \{R_i.REW, R_i.RAq, R_i.RRl\}$, $C(R_i.RAq) = \{W_j.WAq | 1 \leq j \leq m\}$
3. $\tilde{\delta}_i^R$ is described as

$\tilde{\delta}^R$	REW	RAq	$\overline{RAq}$	RRl	ε
$R.Lp$	$\{R.Wt\}$				$\{R.Lp\}$
$R.Wt$		$\{R.Rd\}$	$\{R.Wt\}$		$\{R.Wt\}$
$R.Rd$				$\{R.Lp\}$	$\{R.Rd\}$

The writer is also described as a finite state automaton, formally written as M^W=$(Q_j^W, \Sigma_j^W, \delta_j^W, W_j.Lp)$, where

1. $Q_j^W = \{W_j.LP, W_j.Wt, W_j.Wr\}$,
2. $\Delta_j^W = \{W_j.WEW, W_j.WAq, W_j.WRl\}$, $C(W_j.WAq) = \{R_i.RAq | 1 \leq i \leq n\} \cup \{W_k.WAq | 1 \leq k \leq m \wedge k \neq j\}$,
3. $tilde\delta_j^W$ is described as

$\tilde{\delta}^W$	WEW	WAq	$\overline{WAq}$	WRl	ε
$W.Lp$	$\{W.Wt\}$				$\{W.Lp\}$
$W.Wt$		$\{W.Wr\}$	$\{W.Wt\}$		$\{W.Wt\}$
$W.Wr$				$\{W.Lp\}$	$\{W.Wr\}$

From the transition function of a reader, it is noticed that when a reader is in $R.Wt$ state, on an event RAq the reader either enters $R.Rd$ state or $R.Wt$ state. This means that the next state is nondeterministic, and the next state cannot solely depend on the information of the individual process's event itself. The same argument is also true for the writer, namely when it is in W.Wt state the next state on an input $W.Aq$ is nondeterministic. Also it should be mentioned that sufficient information cannot be obtained from the specification of individual finite state automaton. More powerful tools are required for this purpose.

Through analysis of the finite state automata of individual process, it is found that constraints of the concurrent system are always in the nondeterministic part of finite state automata. For example, when a writer is in the waiting state and the next state on event WAq is either a writing state or a waiting state, which one is the next state depends on the constraint of the whole system. Therefore, more attention should be given to these parts when combining automata together.

5.1.2 Global Finite State Automaton for Readers-Writers System

The Readers-Writers system is modeled as a globe state automaton, defined as $P = (Q, \Sigma, \delta, q_0)$, where

- $Q = Q_1^R \times \ldots \times Q_n^R \times Q_1^W \times \ldots \times Q_m^W \times V_D$ is a finite set called the *global states*, or simply *states*. An element of Q is $(q_1^R, \ldots, q_n^R, q_1^W, \ldots, q_m^W, v_D)$, where $(q_1^R, \ldots, q_n^R, q_1^W, \ldots, q_m^W, v_D) \in Q \Rightarrow q_1^R \in Q_1^R \wedge \cdots \wedge q_n^R \in Q_n^R \wedge q_1^W \in Q_1^W \wedge \cdots \wedge q_m^W \in Q_m^W \wedge v_D \in V_D$

- $\tilde{\Sigma} = \Sigma_1^{R^\varepsilon}$[1] $\times \cdots \times \Sigma_n^{R^\varepsilon} \times \Sigma_1^{W^\varepsilon} \times \cdots \times \Sigma_m^{W^\varepsilon}$ is a finite set called the *alphabet*, or *input symbol*. An element in the set of global events is $A = (a_1, \ldots, a_{n+m})$, where $a_i \in \Sigma_i^{R^\varepsilon} (1 \le i \le n) \wedge a_{n+j} \in \Sigma_j^{W^\varepsilon} (1 \le j \le m)$.

- $\delta : Q \times \tilde{\Sigma} \to 2^Q$ is the *transition function*, $\delta((q_1, \ldots, q_{n+m}, v_D), (a_1, \ldots, a_{n+m})) = \{(\tilde{\delta}_1^R(q_1, \tilde{a}_1), \ldots, \tilde{\delta}_n^R(q_n, \tilde{a}_n), \tilde{\delta}_1^W(q_{n+1}, \tilde{a}_{n+1}), \ldots, \tilde{\delta}_m^W(q_{n+m}, \tilde{a}_{n+m}), v'_D)\}$, where $(q_1, \ldots, q_{n+m}, v_D) \in Q, (a_1, \ldots, a_{n+m}) \in \tilde{\Sigma}, (x_1 \ldots x_{n+m}) = \varphi((q_1, \ldots, q_{n+m}, v_D), A)$, and $\tilde{a}_i \in \{\varepsilon, a_i, \overline{a}_i\}$.

[1] $\Sigma_i^{R^\varepsilon} = \Sigma_i^R \cup \varepsilon$

- $q_0 = (R_1.Lp, \cdots, R_n.Lp, W_1.Lp, \cdots, W_m.Lp, \emptyset) \in Q$ is the initial state, where $\emptyset$ is the initial value of the shared database v.

The transition function is given as tabular expressions depicted in Table 5.1 and it specifies and documents the constraints that the concurrent system imposes, for which an implementation should be followed. The other system properties can be referred from the individual automata model since these activities can be executed independently without interfering with other processes. In such a manner, the traditional automaton and the extended automaton model can be extended to specify the concurrent system.

In the table $\exists!j$ means that there exists only one j in the corresponding set which satisfies the predicate expressions. $?j$ means that only one element of j satisfies the predicate expressions that is selected. Moreover, it should be aware that $\overline{\varepsilon}$ and ε are equivalent.

In the table, the traditional convention of tabular expressions used in the SERG is followed. Each element in the H_1 is a predicate expressions of the current events. For example, the first element $(\exists i|(1 \leq i \leq n) : {}'a_i = R_i.RAq) \wedge (\exists j|(1 \leq j \leq m) : {}'a_j = W_j.WAq \wedge (1 \leq k \leq m))$ in the H_1 represents a set of global events that at least one reader acquires for accessing the database and at least one writer acquires for accessing the database as well. Each element in the header H_2 is the predicate expressions of the current states. For instance, the second element $(|{}'\Gamma| = 1) \wedge (\exists!j|(1 \leq j \leq m) : {}'q_j^W = W_j.Wr) \wedge (\forall i|(1 \leq i \leq n) : q_i^R \neq R_i.Rd)$ in header H_2 represents that one writer is accessing the database. The central part of the table is called *grid*, which represents the predictions of next states in global automaton. For example, the content in grid (1,2) represents a set of next states when one writer is accessing the database and one or more readers and writers acquire to access it as well.

From Cell(1,1) in Table 5.1, it indicates that whenever the database is free, one or more readers acquire for accessing the database, and one or more writers acquire for writing to the database. The set of next states are either all of the acquiring readers in the reading state or one of the acquiring writer in the next state.

5.2 The Completeness [23] of Tabular Expressions

There are three predicate expressions in header H_2 of Table 5.1, each represents part of global state of the readers-writer system. The first one, ${}'v_D = \emptyset$, represents that neither readers nor writers are accessing the database; the second one, $(|{}'v_D| = 1) \wedge (\exists!j|(1 \leq j \leq m) : {}'q_j^W = W_j.Wr) \wedge (\forall i|(1 \leq i \leq n) : q_i^R \neq R_i.Rd)$, represents that one writer is accessing the database; the third one, $(|{}'v_D| \geq 1) \wedge (\forall j|(1 \leq j \leq m) : {}'q_j^W \neq W_j.Wr) \wedge (\exists i|(1 \leq i \leq n) : q_i^R = R_i.Rd)$ indicates that at least one

δ	$'v_D = \emptyset$	$(\lvert 'v_D \rvert = 1) \wedge (\exists! j \lvert (1 \le j \le m) : 'q_j^W = W_j.Wr)$ $\wedge(\forall i \lvert (1 \le i \le n) : q_i^R \ne R_i.Rd)$	$(\lvert 'v_D \rvert \ge 1) \wedge (\forall j \lvert (1 \le j \le m) : 'q_j^W \ne W_j.Wr)$ $\wedge(\exists i \lvert (1 \le i \le n) : q_i^R = R_i.Rd)$
$(\exists i \lvert (1 \le i \le n) :$ $'a_i = R_i.RAq) \wedge$ $(\exists j \lvert (1 \le j \le m) :$ $'a_j = W_j.WAq)$ $\wedge(1 \le k \le m)$	$((q_i^{R'} = \delta_i^R('q_i^R, a_i) \wedge$ $(('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{n+j}))) \wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(v'_D = 'v_D \cup \{R_i \lvert a_i = R_i.RAq\})) \cup$ $(('a_i = R_i.RAq \Rightarrow q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i) \wedge$ $(('a_i \ne R_i.RAq \Rightarrow q_i^{R'} = \delta_i^R('q_i^R, a_i) \wedge$ $(?j('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(('a_{n+k} = W_k.RAq) \wedge (k \ne j))$ $\Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, \overline{a}_{n+k})))$ $(('a_{n+k} \ne W_k.RAq) \Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, a_{n+k})))$ $\wedge(v'_D = 'v_D \cup \{W_j \lvert ?j(A_{n+j} = W_j.WAq)\}))$	$(('a_i = R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i))) \wedge$ $(('a_i \ne R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_j))) \wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(v'_D = 'v_D \setminus \{W_j \lvert a_{n+j} = W_i.WRl\})$	$(('a_i = R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i))) \wedge$ $(('a_i \ne R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_j))) \wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(v'_D = 'v_D \cup \{R_i \lvert a_i = R_i.RAq\}$ $\setminus\{R_i \lvert a_i = R_i.RRl\})$
$(\forall i \lvert (1 \le i \le n) :$ $'a_i \ne R_i.RAq) \wedge$ $(\exists j \lvert (1 \le j \le m) :$ $'a_{n+j} = W_j.WAq) \wedge$ $((\exists k \lvert (1 \le k \le m) :$ $'a_{n+k} = W_k.WAq)$ $\wedge(k \ne j))$	$((\forall i \lvert (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(?j('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_j)))$ $\wedge((('a_{n+k} = W_k.RAq) \wedge (k \ne j)) \Rightarrow$ $(q_{n+k}^{W'} = \delta_k^W('q_k^W, \overline{a}_k)))) \wedge$ $(('a_{n+k} \ne W_k.RAq)) \Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, a_{n+k})))$ $\wedge(v'_D = 'v_D \cup \{W_j \lvert ?j(a_{n+j} = W_j.WAq)\})$	$(\forall i \lvert (1 \le i \le n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{n+j}))) \wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(v'_D = 'v_D \setminus \{W_j \lvert a_{n+j} = W_i.WRl\})$	$(\forall i \lvert (1 \le i \le n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{n+j})) \wedge$ $('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D \setminus \{R_i \lvert a_i = R_i.RRl\})$
$(\forall i \lvert (1 \le i \le n) :$ $'a_i \ne R_i.RAq) \wedge$ $(\exists! j \lvert (1 \le j \le m) :$ $'a_{n+j} = W_j.WAq)$	$(\forall i \lvert (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(\forall j \lvert (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D \cup \{W_j \lvert a_{n+j} = W_j.WAq\})$	$(\forall i \lvert (1 \le i \le n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{j+n}))) \wedge$ $('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{j+n})))$ $\wedge(v'_D = 'v_D \setminus \{W_j \lvert a_{n+j} = W_i.WRl\})$	$(\forall i \lvert (1 \le i \le n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{n+j})))$ $\wedge(v'_D = 'v_D \setminus \{R_i \lvert a_i = R_i.RRl\})$
$(\exists i \lvert (1 \le i \le n) :$ $'a_i = R_i.RAq) \wedge$ $(\forall j \lvert (1 \le j \le m) :$ $'a_{n+j} \ne W_j.WAq)$	$(\forall i \lvert (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(\forall j \lvert (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D \cup \{R_i \lvert a_i = R_i.RAq\})$	$(('a_i \ne R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('a_i = R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i))) \wedge$ $(\forall j \lvert (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})$ $\wedge(v'_D = 'v_D \setminus \{W_j \lvert a_{n+j} = W_i.WRl\})$	$(\forall i \lvert (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i) \wedge$ $(\forall j \lvert (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})$ $\wedge(v'_D = 'v_D \cup \{R_i \lvert a_i = R_i.RAq\}$ $\setminus\{R_i \lvert a_i = R_i.RRl\})$
$(\forall i \lvert (1 \le i \le n) :$ $'a_i \ne R_i.RAq) \wedge$ $(\forall j \lvert (1 \le j \le m) :$ $'a_{n+j} \ne W_j.WAq)$	$(\forall i \lvert (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(\forall j \lvert (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D)$	$(\forall i \lvert (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(\forall j \lvert (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D \setminus \{W_j \lvert a_{n+j} = W_i.WRl\})$	$(\forall i \lvert (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(\forall j \lvert (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D \setminus \{R_i \lvert a_i = R_i.RRl\})$

Table 5.1: Readers-Writer Competes to Access Database

reader is accessing the database and no writer is accessing the database. These three predicate expressions partition a set of global states into a disjoint set. It is obvious that the pairwise conjunction of these three predicate expressions equals to *false*, and the union of them is equal to *true*.

There are five predicate expressions in header H_1 of tabular expressions. The first one, $(\exists i|(1 \leq i \leq n) : {}'a_i = R_i.RAq) \wedge (\exists j|(1 \leq j \leq m) : {}'a_j = W_j.WAq) \wedge (1 \leq k \leq m)$, represents that at least one reader or writer is acquiring for accessing the database. The second one, $(\forall i|(1 \leq i \leq n) : {}'a_i \neq R_i.RAq) \wedge (\exists j|(1 \leq j \leq m) : {}'a_{n+j} = W_j.WAq) \wedge ((\exists k|(1 \leq k \leq m) : {}'a_{n+k} = W_k.WAq) \wedge (k \neq j))$, represents that no reader and more than one writers are acquiring for accessing the database. The third one, $(\forall i|(1 \leq i \leq n) : {}'a_i \neq R_i.RAq) \wedge (\exists! j|(1 \leq j \leq m) : {}'a_{n+j} = W_j.WAq)$, represents that no reader and only one writer is acquiring for accessing the database. The fourth one, $(\exists i|(1 \leq i \leq n) : {}'a_i = R_i.RAq) \wedge (\forall j|(1 \leq j \leq m) : {}'a_{n+j} \neq W_j.WAq)$ represents that no writer and one or more readers are acquiring for accessing the database. The fifth one, $(\forall i|(1 \leq i \leq n) : {}'a_i \neq R_i.RAq) \wedge (\forall j|(1 \leq j \leq m) : {}'a_{n+j} \neq W_j.WAq)$, indicates that neither reader nor writer is acquiring for accessing the database. It is obvious that the pairwise conjunction of these five predicate expressions equals to *false*, and the union of them is equal to *true*.

In header H_2, all of the possible current states regarding the readers-writers concurrent system are considered, and in header H_1 all of the events regarding readers and writers are covered. Therefore, Table 5.1 is complete.

5.3 The Invariant of the System

It should be mentioned that the invariant is an important part of specification and documentation for the concurrent system.

Invariant of the readers-writers system for every global state is:

1. The total number of readers in the local processing state, waiting state and reading state are invariant, namely number n.
2. The total number of writers in the local processing state, waiting state and writing state are also invariant, namely number m.
3. When one writer is writing, other writers and readers cannot be in writing or reading state; Also the number of readers in reading state are always between 0 and n. If no readers are reading and no writer is writing, the database should be in its free state.

Proof: First, it can be referred from the global automata specification of the Readers-Writers system that the initial state implies the invariants. For example, the initial state of system is $q_0 = (R_1.Lp, \cdots, R_n.Lp, W_1.Lp, \cdots, W_m.Lp, \emptyset) \in Q$, which

means that all of the readers and writers are in the local processing state and the database is free. It is obvious that the initial state satisfies all of the above invariants.

Second, it can be proved from the specification of tabular expressions that every legal global event maintains the invariants.

Starting from the initial state, the global automata transacts to one or other states. In the initial state, all of the processes are in the local processing state and database is free. On a global event, automaton transfers to other state by following the cell(5,1), the event on that cell is $(\forall i|(1 \leq i \leq n) : {}'a_i \neq R_i.RAq) \wedge (\forall j|(1 \leq j \leq m) : {}'a_{n+j} \neq W_j.WAq)$ and state is ${}'v_D = \emptyset$, In this case, the next states is $(\forall i|(1 \leq i \leq n) : q_i^{R'} = \delta_i^R({}'q_i^R, a_i)) \wedge (\forall j|(1 \leq j \leq m) : q_j^{W'} = \delta_j^W({}'q_j^W, a_{n+j}))$. This means that every process executes transitions according to its local transition table and the value of database stays the same. For instance, if a reader processing event is $a_i = R_i.REW$, then this reader transacts to the next state according to $\delta_i^R(R_i.Lp, REW) = R_i.Wt$, indicating that this reader enters the waiting state. The number of readers in the local processing state decreases by 1 and the number of readers in waiting state increases by 1. Therefore, the total number of readers in the Lp, Wt, Rd states does not change. A similar situation exists for the writer process j on the event $a_{n+j} = W_j.REW$. In that case, the number of writers in the local processing state decreases by 1 and the number of writers in waiting state increases by 1. The total number of writers in Lp, Wt, Wr states does not change at all. Meanwhile, neither reader nor writer is accessing the database and the database is free. If a reader event is ε, the reader stays in the same state. Similar argument holds for a writer process. Then the number of readers in the Lp, Wt, Rd state and the number of writers in the Lp, Wt, Wr state does not change, and processes in the database do not change neither. From the above arguments, it is safe to claims that the transition in *cell(1,5)* guarantees all of the three invariants.

Considering *cell(1,2)*, the state is $(|{}'v_D| = 1) \wedge (\exists! j|(1 \leq j \leq m) : {}'q_j^W = W_j.Wr) \wedge (\forall i|(1 \leq i \leq n) : q_i^R \neq R_i.Rd)$, which means that no reader and one writer is accessing the database; The event is $(\exists i|(1 \leq i \leq n) : {}'a_i = R_i.RAq) \wedge (\exists j|(1 \leq j \leq m) : {}'a_j = W_j.WAq) \wedge (1 \leq k \leq m)$ which means that at least one reader and at least one writer are acquiring for accessing the database. The set of next states are divided into two categories. One of them is $((q_i^{R'} = \delta_i^R({}'q_i^R, a_i) \wedge (({}'a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W({}'q_j^W, \overline{a}_{n+j}))) \wedge (({}'a_{n+j} \neq W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W({}'q_j^W, a_{n+j}))) \wedge (v_D' = {}'v_D \cup \{R_i | a_i = R_i.RAq\}))$, which means that the readers get the chance to access the database and the writers that acquire for accessing the database are denied. It is assumed that the previous state guarantees the invariants. After the transition of global events, the number of readers in waiting state decreases by the number of acquiring readers, and the number of readers in the reading state increases by the same number corresponding to the reader processes' acquiring event. Meanwhile, other readers' events decrease by the number of readers in one state and increase by

the same number of readers in another particular state. The total number of readers in the Lp, Wt, Rd states keeps the same, namely n. Since the acquiring writers are denied for accessing the database, the number of writers in waiting state remains the same as well. Other writers' events decrease by the number of writers in one state and increase by the same number of writers in another particular state according to the processes' local transition function. Hence, the total number of writers in Lp, Wt, Wr states keeps the same. The number of readers in the database is the number of readers acquiring for accessing, which is larger than 1 and smaller than n since the total number of readers is n. From the above argument, it can be concluded that this category keeps all the above three invariants.

For another category, the set of next events is $(('a_i = R_i.RAq \Rightarrow q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i) \wedge (('a_i \neq R_i.RAq \Rightarrow q_i^{R'} = \delta_i^R('q_i^R, a_i) \wedge (?j('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))) \wedge (('a_{n+k} = W_k.RAq) \wedge (k \neq j)) \Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, \overline{a}_{n+k})))(('a_{n+k} \neq W_k.RAq) \Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, a_{n+k}))) \wedge (v'_D = 'v_D \cup \{W_j | ?j(A_{n+j} = W_j.WAq)\})$, which means that the acquiring readers are denied to access the database and only one of acquiring writers gets the chances to access the database. The denied readers stay in the waiting state, and other readers transfer to another state, which decreases the number of readers in one state and increases by the same number of readers. Hence the total number of readers in the Lp, Wt, Rd states does not change. One of the acquiring writers enters into the database, which decreases the number of writer in waiting state by 1 and increases the number of writer in writing state by 1. The other acquiring writers remain in the waiting state after the transition. The rest of the writers transfer to another state according to the local transition table. Therefore, the total number of writers in the Lp, Wt, Wr states is as the same as before. Moreover, after the transition, only one writer is accessing the database, which satisfies the third invariants. Therefore, it is safe to claims that this category also maintains the invariants.

Other combinations of global events and states can be proved similarly, i.e. they keep the system invariants. Therefore, the tabular expressions of global transition function keep the underlay concurrent system invariants.

5.4 Deadlock Free Specification

It can be proved that the tabular expressions of system transition function will not lead to the deadlock state.

From Table 5.1, when $'v_D = \emptyset$, database is in the free state. If the event is $(\exists i | (1 \leq i \leq n) : 'a_i = R_i.RAq) \wedge (\exists j | (1 \leq j \leq m) : 'a_j = W_j.WAq) \wedge (1 \leq k \leq m)$, then either the acquiring readers can perform a transition to access the database, or one of the acquiring writers can enter to access the database. If the event is

$(\forall i|(1 \leq i \leq n) : 'a_i \neq R_i.RAq) \wedge (\exists j|(1 \leq j \leq m) : 'a_{n+j} = W_j.WAq) \wedge ((\exists k|(1 \leq k \leq m) : 'a_{n+k} = W_k.WAq) \wedge (k \neq j))$, then one of the acquiring writers can perform a transition to access the database. If the event is $(\forall i|(1 \leq i \leq n) : 'a_i \neq R_i.RAq) \wedge (\exists! j|(1 \leq j \leq m) : 'a_{n+j} = W_j.WAq)$, then the acquiring writer can perform a transition to access the database. If the event is $(\forall i|(1 \leq i \leq n) : 'a_i \neq R_i.RAq) \wedge (\forall j|(1 \leq j \leq m) : 'a_{n+j} \neq W_j.WAq)$, the readers and writers can perform the specific transition.

When global state is $(|'v_D| = 1) \wedge (\exists! j|(1 \leq j \leq m) : 'q_j^W = W_j.Wr) \wedge (\forall i|(1 \leq i \leq n) : q_i^R \neq R_i.Rd)$, which represents that one writer is accessing the database. If the event is $(\exists i|(1 \leq i \leq n) : 'a_i = R_i.RAq) \wedge (\exists j|(1 \leq j \leq m) : 'a_j = W_j.WAq) \wedge (1 \leq k \leq m)$. If the event is $(\forall i|(1 \leq i \leq n) : 'a_i \neq R_i.RAq) \wedge (\exists j|(1 \leq j \leq m) : 'a_{n+j} = W_j.WAq) \wedge ((\exists k|(1 \leq k \leq m) : 'a_{n+k} = W_k.WAq) \wedge (k \neq j))$, or if the event is $(\forall i|(1 \leq i \leq n) : 'a_i \neq R_i.RAq) \wedge (\exists! j|(1 \leq j \leq m) : 'a_{n+j} = W_j.WAq)$, then the non-acquiring readers and writers can perform the specific transition, and the writing writer can release the database transition. If the event is $(\forall i|(1 \leq i \leq n) : 'a_i \neq R_i.RAq) \wedge (\forall j|(1 \leq j \leq m) : 'a_{n+j} \neq W_j.WAq)$, the readers and writers can perform the specified transition.

When global state is $(|'v_D| \geq 1) \wedge (\forall j|(1 \leq j \leq m) : 'q_j^W \neq W_j.Wr) \wedge (\exists i|(1 \leq i \leq n) : q_i^R = R_i.Rd)$ which represents that at least one reader is accessing the database and no writer is accessing the database. If the event is $(\exists i|(1 \leq i \leq n) : 'a_i = R_i.RAq) \wedge (\exists j|(1 \leq j \leq m) : 'a_j = W_j.WAq) \wedge (1 \leq k \leq m)$, then the acquiring readers can perform a transition to access the database. If the event is $(\forall i|(1 \leq i \leq n) : 'a_i \neq R_i.RAq) \wedge (\exists j|(1 \leq j \leq m) : 'a_{n+j} = W_j.WAq) \wedge ((\exists k|(1 \leq k \leq m) : 'a_{n+k} = W_k.WAq) \wedge (k \neq j))$, or If the event is $(\forall i|(1 \leq i \leq n) : 'a_i \neq R_i.RAq) \wedge (\exists! j|(1 \leq j \leq m) : 'a_{n+j} = W_j.WAq)$, then the non-acquiring writers can perform the specified transition, and the reading reader can release the database transition. If the event is $(\forall i|(1 \leq i \leq n) : 'a_i \neq R_i.RAq) \wedge (\forall j|(1 \leq j \leq m) : 'a_{n+j} \neq W_j.WAq)$, the readers and writers can perform the specified transition.

From above analyses, it is clear that for any combinations of global states and events, some transition can always be performed. Therefore, our tabular expression is a deadlock-free specification.

5.5 Priority Analyses

As both a graphical and mathematical representation of the concurrent computer systems, global automata are simple and intuitively understandable. Furthermore, when there is a need to express the priority of the different processes, it can specify this property with less effort in a natural manner.

For example, in the Readers-Writers system, when the database is free and both

readers and writers are acquiring to access the resources, the priority is given to the writers over the readers. In this case, the competition set of a reader acquiring event change to $C(R_i.RAq) = \{+W_j.WAq|1 \leq j \leq m\}$. It is noticed that a " + " symbol is added to each writer event in the competition set of reader's acquiring event, which indicates that writers have a high priority over readers when both of them acquire to access the resources at the same time.

The competition set of a writer acquiring event change to
$C(W_j.WAq) = \{-R_i.RAq|1 \leq i \leq n\} \cup \{W_k.WAq|1 \leq k \leq m \wedge k \neq j\}$. A " − " symbol is added to each reader event in the competition set of writer acquiring event to indicate that readers has a less priority over writers. The transition function of global automaton with priority is given as tabular expressions, as shown in Table 5.3. The nondeterministic choice of next state of cell(1,1) in Table 5.3 becomes deterministic or at least more deterministic than the previous specification, namely only the writer can enter to access the resources in this global state.

Furthermore, it can be easily expressed the situation that whenever there are acquiring writers out there, the acquiring readers will not be allowed to enter for accessing resources even the resources still have capacity to allow more readers to enter. Table 5.3 is the transition function for this special case. It can be seen from this table that the acquiring reader is not allowed to enter the reading state even the resources have the capacity, which means that the readers' activity can be downplayed without sacrificing the locks that exist in other readers.

These few cases have demonstrated the natural way to document the design of concurrent software system. Moreover, it can be argued that the tabular expressions significantly reduce the size of traditional transition function. More importantly, it can be used in different development stages to specify and to document the concurrent system properties.

δ	$'v_D = \emptyset$	$(\lvert 'v_D \rvert = 1) \wedge (\exists ! j \mid (1 \le j \le m) : 'q_j^W = W_j.Wr)$ $\wedge(\forall i \mid (1 \le i \le n) : q_i^R \ne R_i.Rd)$	$(\lvert 'v_D \rvert \ge 1) \wedge (\forall j \mid (1 \le j \le m) : 'q_j^W \ne W_j.Wr)$ $\wedge(\exists i \mid (1 \le i \le n) : q_i^R = R_i.Rd)$
$(\exists i \mid (1 \le i \le n) :$ $'a_i = R_i.RAq)\wedge$ $(\exists j \mid (1 \le j \le m) :$ $'a_j = W_j.WAq)$ $\wedge(1 \le k \le)$	$(('a_i = R_i.RAq \Rightarrow q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i)\wedge$ $(('a_i \ne R_i.RAq \Rightarrow q_i^{R'} = \delta_i^R('q_i^R, a_i)\wedge$ $(?j('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(('a_{n+k} = W_k.RAq) \wedge (k \ne j))$ $\Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, \overline{a}_{n+k})))$ $(('a_{n+k} \ne W_k.RAq) \Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, a_{n+k})))$ $\wedge(v_D' = 'v_D \cup \{W_j \mid ?j(a_{n+j} = W_j.WAq)\}))$	$(('a_i = R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i)))\wedge$ $(('a_i \ne R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, a_i)))\wedge$ $(('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_j)))\wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(v_D' = 'v_D \setminus \{W_j \mid A_{n+j} = W_i.WRl\})$	$((\forall i \mid (1 \le i \le n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i)))\wedge$ $(('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_j)))\wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(v_D' = 'v_D \cup \{R_i \mid a_i = R_i.RAq\}$ $\setminus\{R_i \mid a_i = R_i.RRl\})$
$(\forall i \mid (1 \le i \le n) :$ $'a_i \ne R_i.RAq)\wedge$ $(\exists j \mid (1 \le j \le m) :$ $'a_{n+j} = W_j.WAq)\wedge$ $((\exists k \mid (1 \le k \le m) :$ $'a_{n+k} = W_k.WAq)$ $\wedge(k \ne j))$	$((\forall i \mid (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i))\wedge$ $(?j('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_j)))$ $\wedge((('a_{n+k} = W_k.RAq) \wedge (k \ne j)) \Rightarrow$ $(q_{n+k}^{W'} = \delta_k^W('q_k^W, \overline{a}_k))))\wedge$ $(('a_{n+k} \ne W_k.RAq)) \Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, a_{n+k})))$ $\wedge(v_D' = 'v_D \cup \{W_j \mid ?j(a_{n+j} = W_j.WAq)\})$	$(\forall i \mid (1 \le i \le n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i)))\wedge$ $(('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{n+j})))\wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(v_D' = 'v_D \setminus \{W_j \mid a_{n+j} = W_i.WRl\})$	$(\forall i \mid (1 \le i \le n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i)))\wedge$ $('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{n+j}))\wedge$ $('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v_D' = 'v_D \setminus \{R_i \mid a_i = R_i.RRl\})$
$(\forall i \mid (1 \le i \le n) :$ $'a_i \ne R_i.RAq)\wedge$ $(\exists ! j \mid (1 \le j \le m) :$ $'a_{n+j} = W_j.WAq)$	$(\forall i \mid (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i))\wedge$ $(\forall j \mid (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v_D' = 'v_D \cup \{W_j \mid A_{n+j} = W_j.WAq\})$	$(\forall i \mid (1 \le i \le n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i)))\wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{j+n})))\wedge$ $('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{j+n})))$ $\wedge(v_D' = 'v_D \setminus \{W_j \mid a_{n+j} = W_i.WRl\})$	$(\forall i \mid (1 \le i \le n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i)))\wedge$ $(('a_{n+j} \ne W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge('a_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{n+j})))$ $\wedge(v_D' = 'v_D \setminus \{R_i \mid a_i = R_i.RRl\})$
$(\exists i \mid (1 \le i \le n) :$ $'a_i = R_i.RAq)\wedge$ $(\forall j \mid (1 \le j \le m) :$ $'a_{n+j} \ne W_j.WAq)$	$(\forall i \mid (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i))\wedge$ $(\forall j \mid (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v_D' = 'v_D \cup \{R_i \mid a_i = R_i.RAq\})$	$(('a_i \ne R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, a_i)))\wedge$ $(('a_i = R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i)))\wedge$ $(\forall j \mid (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})$ $\wedge(v_D' = 'v_D \setminus \{W_j \mid a_{n+j} = W_i.WRl\})$	$(\forall i \mid (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)\wedge$ $(\forall j \mid (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})$ $\wedge(v_D' = 'v_D \cup \{R_i \mid a_i = R_i.RAq\}$ $\setminus\{R_i \mid a_i = R_i.RRl\})$
$(\forall i \mid (1 \le i \le n) :$ $'a_i \ne R_i.RAq)\wedge$ $(\forall j \mid (1 \le j \le m) :$ $'a_{n+j} \ne W_j.WAq)$	$(\forall i \mid (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i))\wedge$ $(\forall j \mid (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v_D' = 'v_D)$	$(\forall i \mid (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i))\wedge$ $(\forall j \mid (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v_D' = 'v_D \setminus \{W_j \mid a_{n+j} = W_i.WRl\})$	$(\forall i \mid (1 \le i \le n) : q_i^{R'} = \delta_i^R('q_i^R, a_i))\wedge$ $(\forall j \mid (1 \le j \le m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v_D' = 'v_D \setminus \{R_i \mid a_i = R_i.RRl\})$

Table 5.2: Writers with the Priority to Enter for Accessing the Database

δ	$'v_D = \emptyset$	$(\lvert 'v_D \rvert = 1) \wedge (\exists ! j \mid (1 \leq j \leq m) : 'q_j^W = W_j.Wr)$ $\wedge (\forall i \mid (1 \leq i \leq n) : q_i^R \neq R_i.Rd)$	$(\lvert 'v_D \rvert \geq 1) \wedge (\forall j \mid (1 \leq j \leq m) : 'q_j^W \neq W_j.Wr)$ $\wedge (\exists i \mid (1 \leq i \leq n) : q_i^R = R_i.Rd)$
$(\exists i \mid (1 \leq i \leq n) :$ $'A_i = R_i.RAq) \wedge$ $(\exists j \mid (1 \leq j \leq m) :$ $'A_j = W_j.WAq)$ $\wedge (1 \leq k \leq)$	$(('A_i = R_i.RAq \Rightarrow q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i) \wedge$ $(('A_i \neq R_i.RAq \Rightarrow q_i^{R'} = \delta_i^R('q_i^R, a_i) \wedge$ $(?j('A_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(('A_{n+k} = W_k.RAq) \wedge (k \neq j))$ $\Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, \overline{a}_{n+k})))$ $(('A_{n+k} \neq W_k.RAq) \Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, a_{n+k})))$ $\wedge(v'_D = 'v_D \cup \{W_j \mid ?j(A_{n+j} = W_j.WAq)\}))$	$(('A_i = R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i))) \wedge$ $(('A_i \neq R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('A_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_j))) \wedge$ $(('A_{n+j} \neq W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(v'_D = 'v_D \setminus \{W_j \mid A_{n+j} = W_i.WRl\})$	$(('A_i = R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i))) \wedge$ $(('A_i \neq R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('A_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_j))) \wedge$ $(('A_{n+j} \neq W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(v'_D = 'v_D \cup \{R_i \mid A_i = R_i.RAq\}$ $\setminus \{R_i \mid A_i = R_i.RRl\})$
$(\forall i \mid (1 \leq i \leq n) :$ $'A_i \neq R_i.RAq) \wedge$ $(\exists j \mid (1 \leq j \leq m) :$ $'A_{n+j} = W_j.WAq) \wedge$ $((\exists k \mid (1 \leq k \leq m) :$ $'A_{n+k} = W_k.WAq)$ $\wedge (k \neq j))$	$((\forall i \mid (1 \leq i \leq n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(?j('A_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_j)))$ $\wedge((('A_{n+k} = W_k.RAq) \wedge (k \neq j)) \Rightarrow$ $(q_{n+k}^{W'} = \delta_k^W('q_k^W, \overline{a}_k)))) \wedge$ $(('A_{n+k} \neq W_k.RAq)) \Rightarrow (q_k^{W'} = \delta_k^W('q_k^W, a_{n+k})))$ $\wedge(v'_D = 'v_D \cup \{W_j \mid ?j(A_{n+j} = W_j.WAq)\})$	$(\forall i \mid (1 \leq i \leq n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('A_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{n+j}))) \wedge$ $(('A_{n+j} \neq W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge(v'_D = 'v_D \setminus \{W_j \mid A_{n+j} = W_i.WRl\})$	$(\forall i \mid (1 \leq i \leq n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $('A_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{n+j})) \wedge$ $('A_{n+j} \neq W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D \setminus \{R_i \mid A_i = R_i.RRl\})$
$(\forall i \mid (1 \leq i \leq n) :$ $'A_i \neq R_i.RAq) \wedge$ $(\exists ! j \mid (1 \leq j \leq m) :$ $'A_{n+j} = W_j.WAq)$	$(\forall i \mid (1 \leq i \leq n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(\forall j \mid (1 \leq j \leq m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D \cup \{W_j \mid A_{n+j} = W_j.WAq\})$	$(\forall i \mid (1 \leq i \leq n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('A_{n+j} \neq W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{j+n}))) \wedge$ $('A_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{j+n})))$ $\wedge(v'_D = 'v_D \setminus \{W_j \mid A_{n+j} = W_i.WRl\})$	$(\forall i \mid (1 \leq i \leq n) : (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('A_{n+j} \neq W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})))$ $\wedge('A_{n+j} = W_j.RAq) \Rightarrow (q_j^{W'} = \delta_j^W('q_j^W, \overline{a}_{n+j})))$ $\wedge(v'_D = 'v_D \setminus \{R_i \mid A_i = R_i.RRl\})$
$(\exists i \mid (1 \leq i \leq n) :$ $'A_i = R_i.RAq) \wedge$ $(\forall j \mid (1 \leq j \leq m) :$ $'A_{n+j} \neq W_j.WAq)$	$(\forall i \mid (1 \leq i \leq n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(\forall j \mid (1 \leq j \leq m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D \cup \{R_i \mid A_i = R_i.RAq\})$	$(('A_i \neq R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, a_i))) \wedge$ $(('A_i = R_i.RAq) \Rightarrow (q_i^{R'} = \delta_i^R('q_i^R, \overline{a}_i))) \wedge$ $(\forall j \mid (1 \leq j \leq m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})$ $\wedge(v'_D = 'v_D \setminus \{W_j \mid A_{n+j} = W_i.WRl\})$	$(\forall i \mid (1 \leq i \leq n) : q_i^{R'} = \delta_i^R('q_i^R, a_i) \wedge$ $(\forall j \mid (1 \leq j \leq m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j})$ $\wedge(v'_D = 'v_D \cup \{R_i \mid A_i = R_i.RAq\}$ $\setminus \{R_i \mid A_i = R_i.RRl\})$
$(\forall i \mid (1 \leq i \leq n) :$ $'A_i \neq R_i.RAq) \wedge$ $(\forall j \mid (1 \leq j \leq m) :$ $'A_{n+j} \neq W_j.WAq)$	$(\forall i \mid (1 \leq i \leq n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(\forall j \mid (1 \leq j \leq m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D)$	$(\forall i \mid (1 \leq i \leq n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(\forall j \mid (1 \leq j \leq m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D \setminus \{W_j \mid A_{n+j} = W_i.WRl\})$	$(\forall i \mid (1 \leq i \leq n) : q_i^{R'} = \delta_i^R('q_i^R, a_i)) \wedge$ $(\forall j \mid (1 \leq j \leq m) : q_j^{W'} = \delta_j^W('q_j^W, a_{n+j}))$ $\wedge(v'_D = 'v_D \setminus \{R_i \mid A_i = R_i.RRl\})$

Table 5.3: Reader Not Allowed to Enter Database When There Are Acquiring Writers

Chapter 6

Conclusion and Future Work

This chapter summarizes the contributions of this thesis, and suggests future research work in the area.

It is important to mention that it is unwise to use our constrain specification technique to write all of the specification documentation for concurrent systems. Instead, it is intended to use this technique as a complement for other specification techniques, i.e., functional (relational) documentation technique used by software engineering group at McMaster University. It should be clear that each approach serves a unique purpose and offers the ability to specify a particular aspect of a software system's external behavior. For example, as parts of a large concurrent system, some of component programs do terminate after running for a while, these component programs can be specified and described with the functional document method.

The goals of this thesis is to use the same tabular notation to document the concurrent systems and to find a way to couple these two schemes under one roof to fulfill our dream to specify the concurrent systems efficiently and precisely.

6.1 Contributions

In this thesis, we slightly extended the traditional finite state automata, associated two sets, namely competition set and synchnization set, to every event of automata, and provided a frame work to composite individual automata to global automata in order to document the characteristics of concurrency. The major contribution of this thesis is to extend the tabular expressions used by SERG to document concurrent programs explicitly. This new method is ready for use in practical software documents.

Another contribution of this thesis is to provide a way to couple automaton theory and tabular expressions to describe the concurrent system efficiently and precisely.

To the author knowledge, this is the first systematic attempt to model concurrency in the framework of tabular expressions. Automata have been used before but more restrictions [3, 7, 34, 35, 36] than we assumed. Hence our approach is most likely far from being perfect, but it works as we have shown. The beginning of our approach is philosophically similar to the petri net approach [19], but then it is different. It seems to rather clear, that for big system a set of tools is necessary. Unfortunately in comparison with Petri nets [19] the set of tools for Tabular Expressions is rather poor. Complexities of our method and Colored Petri Net methods looks similar.

6.2 Applications

In the thesis, the global state automata model is used to specify and to describe the software constrains inherited in the concurrent software system. This method can be used to specify the functionality of concurrent systems, and to record design decisions made during system development. For example, in the Reader-Writers system, we can specify the functionality of readers and writers as well as to record whether allowing Readers or Writers to enter database when both of them are acquired to enter the database.

It is obvious that our model can be utilized to write the specification of concurrent systems with a serial of processes to access multiple resources to carry its task. This method can be used to write a software requirement document, module interface specification, and module design document.

The tabular expressions can also be used to document constrains of concurrent programs. These tabular expressions can be utilized to generate test cases for concurrent programs. It can provide some guidelines for selecting input data to test programs as well as to determine the correctness of programs. Each condition row in the tabular expressions should be explicitly tested for different input events to valid the concurrent program to ensure that it satisfies the constrain to which the system exposes.

6.3 Limitations

Our model cannot express the situation when real time comes into play in the system. It is a very important area that needs to be researched.

Fairness is another aspect of concurrency program properties. Our model should be further extended by adding an artificial label (variable in the context of language) to control the fairness constrain.

6.4 Future Work

The work presented in this thesis can be further extended in several aspects.

First of all, it is important that the documentation written by this method should be complete and deadlock free. For the application of practical computer system, it is tedious and error prone to check these issues. Since transition relations are written in the tabular form for both individual automata and composite automata, such potential does exist that automate check the safety of concurrent systems. It is hoped that tools can be developed to check the deadlock free property based on our tabular specification.

Secondly, it is useful if a tool could be produced that generates a Test oracle [33] from our formal specification, and verifies that the implementation does what it is supposed to do.

Thirdly, in the context of concurrent programming, timing constrains inducted by synchronization and communication do arise, especially in the real-time system, in which time constrain is reinforced. Therefore, it is of interest to further extend our model to addresses the timing issues.

Finally, it will be nice that more power is added to solve the fairness problem. One of possibilities is to add more artificial variables to retain more information.

6.5 Conclusions

The main goal of this thesis is to develop a simple and precise way to write formal specification and documentation of concurrent software systems with the tabular form. The formal definition of composite automata and examples illustrated in the previous chapters have demonstrated that our model can be easily applied and well understood. We believe that it can be used to write the specification and documentation for practical applications with less efforts. However tools, at least as sophisticated as for Colored Petri Net benchmark are neccessary.

Bibliography

[1] Abraham, R.F., “Evaluating Generalized Tabular Expressions in Software Documentation”, CRL Report 346 February 1997.

[2] Baker, B., Parnas, D.L., “Applying Mathematical Software Documentation - An Experience Report”, in Proceedings of the Tenth Annual Conference on Computer Assurance, National Institute of Standards and Technology, Gaithersburg, Maryland, June 25 - 29, 1995, pp. 273 - 285.

[3] Beyga, L., Gajewski, T., Miadowicz, Z,. P.Siwak, P., Stoklosa, J., Bergandy, J., Mikolajczak, B. ”Algebraic and structural automata theory”, Elsevier Science Publishers B.V. 1991.

[4] Courtois P.-J., Parnas D.L., “Documentation for Safety Critical Software”, Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, 17 - 21 May, 1993.

[5] Denvir, B.T., Harwood, W.T., Jackson M.I. and Wray M.J., “The Analysis of Concurrent Systems”, Proceedings of a workshop on the Analysis of Concurrent system held at Clare college ,Cambridge, September 12-16, 1983, Lecture Notes in Computer Science 207, Springer-Verlag.

[6] Diekert, V., Rozenberg, G., “The book of traces”, World Scientific 1995.

[7] Gecseg, F., “Product of Automata”, EATCS Monograph on Theoretical Computer Science Volume 7, Springer-Verlag Berlin Heidelberg 1986.

[8] Godefroid, P., “Partial-Order Methods for the Verification of Concurrent Systems- An Approach to the State-Explosion Problem”, Lecture Notes in Computer Sciences 1032 Springer-Verlag 1996.

[9] Hailpern, Brent T., “Verifying Concurrent Processes Using Temporal Logic”, Ph.D. Thesis, Stanford University, 1980. Lecture Notes in Computer Science, volume 129. Springer Verlag, 1982.

[10] Janicki, R., Koutny M., "Structure of Concurrency", Theoretical Computer Science 112(1993), 5-52

[11] Janicki, R., "Towards a Formal Semantics of Tables", in Proc. Conference Software Engineering (ICSE), pp.231-240, Apr. 1995.

[12] Janicki, R., Parnas, D.L., Zucker, J., "Tabular Representations in Relational Documents", CRL Report 313, McMaster University, CRL, TRIO, November 1995.

[13] Janicki, R., "On a Formal Semantics of Tabular Expressions", CRL Report 355, October 1997

[14] Janicki, R., "Tabular Expressions and Their Relational Semantics", SERG report No.377, July 1999

[15] Janicki, R., Khedri, R., "On a Formal Semantics of Tabular Expressions", Science of Computer Programming 39(2001), 189-213.

[16] Janicki, R., Liu, Y., "On Trace Assertion Method of Module Interface Specification with Concurrency", Lecture Notes in Artificial Intelligence 2001, Springer Verlag 2001.

[17] Janicki, R., Sekerinski, E., "Foundations of the Trace Assertion Method of Module Interface Specification", IEEE Transactions on Software Engineering, 27,7 (2001)

[18] Janicki, R., "Formal Specification and Finitely Defined Automata with Interpreted States" , Proceedings of the International Conference on Software Engineering Research and Practice, Las Vegas, Nevada, USA, June 24-27, 2002, CSREA Press.

[19] Jensen, K., "Coloured Petri Nets", Vol. 1,2,3, Springer-Verlag 1994, 1996, 1999.

[20] Klarlund, N., Mukund, M., Sohoni, M., "Automata, Languages and Programming", 21st International Colloquium, ICALP94, Jerusalem, Israel, July 1994, Lecture Notes in Computer Sciences 820, 130-153, Springer-Verlag.

[21] Parnas, D.L., "Functional Specifications for Old (and New) Software", Proceedings of the 20th GI Jahrestagung, Stuttgart, Germany, 10 October 1990, edited by Reuter, A., Informatik Fachberichte 257, Springer-Verlag, Berlin.

[22] Parnas, D.L., Asmis, G.J.K., Madey, J., "Assessment of Safety-Critical Software in Nuclear Power Plants", Nuclear Safety, vol. 32, no. 2, April-June 1991, pp. 189-198.

[23] Parnas, D.L., "Tabular representation of relations", CRL Report 260, Communication Research Laboratory, Nov. 1992

[24] Parnas, D.L., Madey, J., "Functional Documentation for Computer Systems Engineering" in Science of Computer Programming (Elsevier) vol.25, number 1, October 1995.

[25] Parnas, D.L., "Predicate Logic for Software Engineering", IEEE Transactions on Software Engineering, Vol. 19, No. 9, September 1993, pp. 856-862.

[26] Parnas, D.L., "Mathematical Descriptions and Specification of Software", Proceedings of IFIP World Congress 1994, Volume I, August 1994, pp.354 - 359.

[27] Parnas, D.L., "Inspections of Safety Critical Software using Function Tables", Proceedings of IFIP World Congress 1994, Volume III, August 1994, pp. 270 - 277.

[28] Parnas, D.L., Madey, J., Iglewski, M., "Precise Documentation of Well-Structured Programs", IEEE Transactions on Software Engineering, Vol.20, No.12, December 1994, pp. 948 - 976.

[29] Parnas, D.L., "Using Mathematical Models in the Inspection of Critical Software", in Applications of Formal Methods, Hinchey M.G., Bowen J.P. (eds.), Prentice Hall International Series in Computer Science, 1995, pp. 17-31.

[30] Parnas, D.L., "A Logic for Describing, not Verifying, Software", Erkenntnis (Kluwer), volume 43, No. 3, November 1995, pp. 321-338.

[31] Parnas, D.L., "Precise Description and Specification of Software", in Mathematics of Dependable Systems II, edited by V. Stavridou, Clarendon Press, 1997, pp. 1 - 14.

[32] Shen H., Zucker J.I., Parnas, D.L., "Table Transformation Tools: Why and How", Proceedings of the Eleventh Annual Conference on Computer Assurance (COMPASS '96), published by IEEE and NIST, Gaithersburg, MD., June 1996, pp. 3-11.

[33] Peters, D., "Generating a Test Oracle from Program Documentation", CRL Report 302, April 1995

[34] Rattray, C., "Specification and verification of concurrent system", Springer-verlag, 1990.

[35] Shields, M.W., "An introduction to Automata Theory", Blackwell Scientific publications, 1987

[36] Zielonka,W., "Notes on finite asynchronous automata", RAIRO Theoretical Informatics and Applications, vol. 21,(1987), pp. 99–135

[37] IEEE Guide for Developing System Requirements Specifications, IEEE Std 1233, 1998 Edition