T-Cham
A Programming Language Based on Transactions and the Chemical Abstract Machine

Wanli Ma

A thesis submitted for the degree of

Doctor of Philosophy

at

The Australian National University

Statement

I hereby state that this thesis contains only my own original work except where explicit reference has been made to the work of others.

[illegible]

Wanli Ma

To

My Parents: *Heling Ma, Yuxiu Jia*
and
My Wife and Daughter: *Ting Lu, Tianhui (Lulu) Ma*

for Their Love

献给

我 的 父 母: 马 鹤 龄, 贾 毓 秀
妻 女: 卢 婷, 马 天 卉

Acknowledgments

I'd like to express my sincere thanks to the past and current members of my supervisory panel: Richard P. Brent, Christopher W. Johnson, E. V. Krishnamurthy, J. Bruce Millar, Malcolm C. Newey, and Mehmet A. Orgun[1] for their advice, encouragement, and support, which directly shape this thesis. A special thank you goes to Prof. John Lloyd for his help and encouragement when I was revising the first version of the thesis.

I'd like to thank the examiners of the thesis for their constructive comments.

I'd like also to give my special thanks to my English tutor Ms June Hornby for her help with my English.

A thank you also goes to all the people, to name a few: Dennis Andriolo, Joe Elso, Arthur McGuffin, and Michelle Moravec, for their wonderful work at CSL, ANU.

I'd like to extend my thanks to Katarina Christenson and Geoff Rozenberg[2] for their understanding and support me to continue the thesis after I joined them to work for the University of Canberra.

Finally, my deepest thanks go to my family: my parents, my wife and my daughter. Their love and assistance made the thesis possible.

The work presented in this thesis was supported by the Overseas Postgraduate Research Scholarship (OPRS) of the Australian government and the ANU PhD Scholarship of the Australian National University.

[1]Department of Computing, Macquarie University.
[2]Both are at the University of Canberra.

Abstract

The last two decades witnessed the fast development of parallel and distributed computers, but their applications are still obstructed by the facts that the design and implementation of a parallel program are very complex, and only a few of those who have been well trained in this area can barely manage.

Programming languages play a vital role in program development and implementation. Although a plethora of concurrent, including both parallel and distributed, programming languages and models have been proposed, the parallel programming is still far more difficult than in the sequential case. We believe one of the most important reasons is that the differences between the concurrent and sequential programming are not lying in *the single thread* nature versus *the multi-thread with communications* but a *functionality program* versus a *reactive one*, and therefore, concurrent programming languages should be designed to reflect those new features.

In this thesis, we propose a new concurrent programming language—T-Cham. It extends the *Chemical Abstract Machine* (Cham) with transactions. A Cham is an interactive computational model based on chemical reaction metaphor, where a computation proceeds as a succession of chemical reactions. A transaction is a piece of programming code which has the properties of ACID (*Atomicity*, *Consistency*, *Isolation*, and *Durability*). A T-Cham program can be executed in a parallel, distributed, or sequential manner based on the available computer resources. As a newcomer to the crowded parallel and distributed programming language community, T-Cham emphasizes *simplicity*, *efficiency*, and a sound *theoretical background*.

Recent Publications

During the period of my PhD research, I published several papers. Some of them are directly related to T-Cham, while the others contribute to my future work. A list of the related publications follows:

1. W. Ma, M. A. Orgun and C. W. Johnson. Towards a Temporal Semantics for Frame. In *Proceedings SEKE'98, Tenth International Conference on Software Engineering and Knowledge Engineering*, pages 44–51. Knowledge Systems Institute, 3420 Main Street, Skokie, IL 60076, USA, 1998.

2. W. Ma, C. W. Johnson and R. P. Brent. Programming with Transactions and Chemical Abstract Machine. In Guo-Jie Li, D. F. Hsu, S. Horiguchi and B. Maggs, editors, *Proceedings of Second International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'96)*, pages 562–564. IEEE Computer Society Press, 1996.

3. W. Ma, C. W. Johnson and R. P. Brent. Concurrent Programming in T-Cham. In K. Ramamohanarao, editor, *Nineteenth Australasian Computer Science Conference Proceedings (ACSC'96)*, pages 291–300. Vol. 18, No. 1, Australian Computer Science Communications, 1996.

4. W. Ma and M. Orgun. Verifying Multran Programs with Temporal Logic. In M. Orgun and E. Ashcroft, editors, *Intensional Programming I*, pages 186–206, World Scientific Publishing, 1996.

5. W. Ma, V. K. Murthy and E. V. Krishnamurthy. Multran — A Coordination Programming Language Using Multiset and Transactions. In S. K. Aityan, L. T. Hathaway, et al., editors, *Proceedings of the First International Conference on Neural, Parallel, and Scientific Computations*, pages 301–304. Dynamic Publishers, Inc., 1995.

6. W. Ma, E. V. Krishnamurthy and M. A. Orgun. On Providing Temporal Semantics for the GAMMA Programming Model. In C. B. Jay, editor, *CATS: Proceedings of Computing: the Australian Theory Seminar*, pages 121–132, University of Technology, Sydney, Australia, 1994.

7. M. A. Orgun and W. Ma. An Overview of Temporal and Model Logic Programming. In Dov M. Gabbay and H. J. Ohlbach, editors, *The First International Conference on Temporal Logic, LNAI 827*, pages 445–479. Springer-Verlag, 1994.

8. K. Zhang and W. Ma. Graphical Assistance in Parallel Program Development. In A. L. Ambler and T. D. Kimura, editors, *Proceedings of IEEE Symposium on Visual Languages*, pages 168–170. IEEE Computer Society Press, 1994.

Contents

1 Introduction 1

1.1 Introduction . . . 2

1.1.1 Observations . . . 2

1.1.2 Programming Language Design Criteria . . . 4

1.1.3 Our Proposal . . . 4

1.2 Major Contributions . . . 6

1.3 Outline of the Thesis . . . 7

2 Background and Related Work 9

2.1 Parallel Computer Systems . . . 11

2.2 Parallel Programming Models and Languages . . . 12

2.2.1 Petri Nets . . . 14

2.2.2 GAMMA Model and the Chemical Abstract Machine . . . 16

2.2.3 The Linda Paradigm . . . 18

2.2.4 Unity and Swarm . . . 20

2.2.5 Argus and the Transaction Programming Paradigm . . . 23

2.2.6 Strand, PCN and Bilingual Programming Languages . . . 25

2.2.7 The DINO Programming Language . . . 27

2.2.8 Others . . . 29

2.3 Conclusion . . . 32

3 Motivation **33**

3.1 Observations on Parallel Programming 35

3.1.1 Functionality versus Interaction 35

3.1.2 Single Thread, Multi-thread and Non-thread 37

3.1.3 Shared Memory versus Distributed Memory 38

3.1.4 Granularity 39

3.1.5 Variables versus Resources 39

3.1.6 Debugging versus Verification 41

3.2 Motivation: Questions Answered 42

3.2.1 Why Yet Another Parallel Programming Language? 42

3.2.2 Why the Chemical Abstract Machine? 43

3.2.3 Why Coordination? 44

3.2.4 Why Tuple Space? 44

3.2.5 Why Transactions? 45

3.2.6 Why Theoretical Background? 45

3.2.7 Why Program Composition? 46

4 Basic T-Cham Notations **47**

4.1 Notational Conventions 48

4.2 The Essential Components of a Chemical Abstract Machine 49

4.3 Non-leaf transactions 50

4.3.1 Tuples 51

4.3.2 Initialization 53

4.3.3 Reactionrules 54

4.3.4 Termination 55

4.3.5 Sub-transactions 56

4.4 Leaf Transactions 56

4.5 A Small Example 57

4.6 The Execution of a T-Cham Transaction 59

4.7 Summary 61

5 Programming in T-Cham 63

5.1 The Sieve of Eratosthenes 64

5.2 Vending Machine 66

5.3 The Producer-Consumer Problem 68

5.4 The Dutch Flag Problem 70

5.5 Sleeping Barber 72

5.6 The Meeting Scheduler 74

5.7 The Fibonacci Numbers 75

5.8 The Calculation of the Value of π 79

5.9 Discussion 81

6 T-Cham Implementation 83

6.1 Introduction 84

6.2 The Execution Model of T-Cham 85

6.2.1 Tuple Space Partition and Duplication 88

6.2.2 The Communications Between Task Managers 90

6.2.3 The Communications Between Task Managers and Executors . . 95

6.3 The Basic Performance Measurements: T-Cham Machine Implementation Case Study 96

6.3.1 The Basic Performance Experiment 97

6.3.2 The Performance Measurements of Matrix Multiplication 101

6.4 Conclusion 104

7 Towards the Temporal Logic Proof System of T-Cham 107

7.1 Temporal Logic 108

7.2 The Temporal Logic Model of T-Cham 113

7.2.1 From a T-Cham Program to its Temporal Logic Formulae 113

7.2.2 Stuttering 116

7.3 The Impact of Race between Reaction Rules 117

7.3.1 The Problem 117

7.3.2 The Solution 122

7.3.3 How to Decide No Race Under Race Condition 124

7.3.4 Dining Philosophers: An Example 126

7.4 T-Cham Program Verification 128

7.4.1 The Producer-Consumer Problem 128

7.4.2 The Dutch Flag Problem 130

7.4.3 The Meeting Scheduler Problem 132

7.5 The Impact of T-Cham Termination Conditions 135

7.6 Conclusion 138

8 Advanced Notations: Hierarchical Tuple Spaces and Tuple Mapping 141

8.1 Tuple Structures, Hierarchical Tuple Spaces and Transaction Granularities 142

8.1.1 Case Study 142

8.1.2 New Concepts: Hierarchical Tuple Spaces, Tuple Mapping and Transaction Granularities 146

8.2 Tuple Mappings and Plain Masks 148

8.3 Regular Masks 150

8.4 Matrix Multiplication: an Example of Using Masks 151

8.5 Implementation Issues 154

8.6 Summary 156

9 A Compositional Proof System 157

9.1 Union of Transactions 157

9.2 Union of Temporal Formulae 158

9.3 Properties of the United Transactions 161

9.4 Superposition 163

9.5 Summary 164

10 Conclusion and Future Work 167

A The BNF Definition of T-Cham Syntax 173

B The AP1000 Multicomputer 177

Bibliography 179

List of Figures

2.1 Shared Memory versus Distributed Memory Computer Architectures . . 13
2.2 The Firing of a Transition in a Petri Net 14
2.3 Producer-Consumer in Petri Net 15
2.4 Sorting an Array in GAMMA 17
2.5 The Pairwise Summation of Two Arrays in C-Linda 20
2.6 An Observation on the Tuple Space 21
2.7 Sorting an Array in Unity 22
2.8 A Parallel Array Summation Program in Swarm 24
2.9 Producer-Consumer Program in PCN 28
2.10 Matrix-Vector Multiplication Program in DINO 30
3.1 The Spectrum of T-Cham between GAMMA and C 34
3.2 The Conceptual Levels of Programming 37
4.1 The T-Cham Program of Element Summation 58
4.2 A Possible Computation Process of Element Summation 60
5.1 The T-Cham program of Eratosthenes Sieve 64
5.2 Another T-Cham program of Eratosthenes Sieve 65
5.3 The T-Cham Program of Vending Machine 67
5.4 The T-Cham Program of Producer-Consumer Problem 69
5.5 The T-Cham Program of Dutch Flag 71
5.6 The T-Cham Program of Sleeping Barber 73
5.7 The T-Cham Program of Meeting Scheduler 76

5.8 The T-Cham Program of Fibonacci Number 78
5.9 The T-Cham Program to Calculate the Value of π 80
6.1 The Implementation of T-Cham 87
6.2 The Tuple Space Partition Algorithm 89
6.3 The Tuple Space Partition of Vending Machine Problem 91
6.4 The Tuple Migration Algorithm 93
6.5 The Bid Handling Algorithm 94
6.6 The Task Bidding and Receiving Algorithm 95
6.7 The Communication Overhead for a Reaction 98
6.8 Tuple Space Partition and Communication Patterns of Sleeping Barber 100
6.9 The Calculation of *TaskBidHist* and Hit Rate 101
6.10 The Choice Function 102
6.11 Performance of Matrix Multiplication (128x128) 103
7.1 Illustrative Examples of Temporal Logical Operators 110
7.2 Different Execution Paths 118
7.3 The Graphs for Tuple Re-producing Checking 125
7.4 The T-Cham program of Dining Philosophers 127
7.5 A small T-Cham program 137
8.1 The T-Cham program of Vector Summation, the First Approach 143
8.2 Vector Summation, the First Approach 143
8.3 Vector Summation, the Second Approach 144
8.4 The T-Cham program outline of Vector Summation, the Second Approach 145
8.5 Vector Summation, the Third Approach 146
8.6 The Vertical Mappings of Tuples 149
8.7 The Ordering Operator 152
8.8 The Multiplication of Two Matrices 153
9.1 The Transactions of `shop`, `customer`, and `barber` 159
B.1 The Architecture of the AP1000 Multicomputer 178

Chapter 1

Introduction

The last two decades witnessed a rapid development of parallel and distributed computation techniques, both in hardware and software. Many parallel computers have been manufactured, and many parallel programming models, languages and development environments have been proposed. The ever growing need for computational power and the maturity of techniques for connecting computers together have given an impetus to the rapid growth of parallel computer applications. In addition, widely installed computer networks provide the opportunities for computers to work together, so called distributed computation, to provide more power than any individual computer.

The computational power of sequential computers is approaching its limits because of the maximum speed limit of electron transmission and the minimum feasible integrated circuit chip size. An alternative way to achieve more computational power is to use parallelism, i.e., to bind a number of sequential computers together and program them to cooperate in solving a problem. The major obstacle that prevents parallel computers from more general usage today is not the design and manufacturing of the hardware but the design and implementation of parallel programs, i.e., the lack of widely accepted methodologies, programming models and languages, and the related supporting tools for parallel program development. These software techniques are still under development and far from maturity.

In this thesis, we suggest a united paradigm of parallel computational model, programming language, and programming development environment.

This chapter first gives a brief introduction to the idea and then our major contributions. Finally, the outline of the thesis is given.

1.1 Introduction

1.1.1 Observations

A plethora of concurrent, both parallel and distributed, programming languages and models has been proposed, yet parallel programming is far more difficult than sequential programming, as most of the languages just extend the existing sequential programming languages with thread control facilities. Those approaches take the assumption that the difference between concurrent and sequential programming is the number of active control threads at a given time. In fact, the difference, realised recently by the research communities, is not in *the single thread* nature versus *the multi-thread with communications*, but in the contrast, between a *functionality program*[1] versus a *reactive one* [111]. The same observation is also realised by distinguishing a *closed* program from an *open* one in [47].

A functionality program is the one which maps an input state (or data) into an output state. Traditionally, we explain a program in this way: it accepts some input data and then, according to the instructions of the program, produces the output data. The instructions are executed in a step by step manner. There may be some procedure and function calls, but, at any time instance, only a single control flow, which is also known as a *thread*. The flow starts from an initial point, normally the first instruction of the main module of that program, and terminates at a stop point. For example, the value of the n^{th} Fibonacci number $f(n)$ is,

$$f(n) = \begin{cases} f(n-1) + f(n-2) & \text{if } n > 2 \\ 1 & \text{if } n = 1 \text{ or } n = 2, \end{cases}$$

where the input data is the n while the output is $f(n)$. Inside of the program, there are a number of instructions which operate on the input data n to produce the final

[1]A functionality program is *not* a functional program. See Section 3.1.1 for details.

result. Denotational semantics [160, 82] is the formal description of the idea, where each program statement transfers the program from the state before the execution of itself to the state after it. The behaviour of the whole program is to map the initial state to a final state.

The same conceptual model also exists in our daily life. For example, assembling lines in the manufacturing industries, first-aid procedures, check list style work sheets and so on are the examples of functional model. But the simple model fails in a more complex situation. For example, when we drive a car approaching a roundabout, we have to interact with other cars, the one on the right and the one in the opposite direction (the car may stop the one on the right). An interactive model is necessary to describe this kind of behaviour.

When more than one computer is put together to make a parallel computer system, each component computer works on one thread. The multi-thread manner contributes to the execution of a parallel program. At first, it is natural to think that the main task of parallel programming is the thread control, e.g., thread *fork*, *join* and *termination*, but with the increasing number of element computers, hundreds and thousands in a massive parallel computer system, handling the threads becomes more and more difficult.

A reactive system emphasizes the interactions among the components of a program: different parts of a program interact with each other in response to stimuli from the "outside" world. The *intrinsic property* of the systems is the *interactions among the components* rather than the co-existing execution flows (or, multi-control-thread) although the latter can also be observed from the outside of the systems.

The widely accepted object-oriented programming techniques [80, 132, 155] provide auxiliary evidence that a program consists of interacting components. An object-oriented program comprises a number of objects. Each of them has its interface to communicate with other objects and some methods to conduct the required internal operations. The interactions among those objects, not the control flows, are the primary concern in the object-oriented programming.

The concepts of *sequential*, *parallel*, or *distributed* belong to the execution of a program on a particular computational resource rather than the program itself. There are no sequential, parallel, or distributed programs but only *functionality* or *interactive* programs. Accordingly, the programming language for concurrent activities should not be judged only by its abilities of thread control and communications but also the abilities to express the interactions. In other words, a concurrent programming language should not be just an extension to an existing sequential programming language with some thread control and communication facilities, but a new one based on an interactive computation model.

1.1.2 Programming Language Design Criteria

There are many choices available when designing a programming language, and there are also many selection criteria in making the choice. The three most important criteria, we believe, are the *programmability*, *execution efficiency*, and *provability* of the programming language. As a programming language is the tool for human beings to instruct computers to work properly, it should be easy for human beings to manage and use, by which we call it *programmability*. On the other hand, the programs written in the programming language have to be executed on a real computer (or computers). The *execution efficiency* is the ultimate goal for the overall performance. The most efficient programming language is the assembly language of a particular computer, but it is far from human-friendly (i.e., it has low programmability). The higher level a programming language is, the less efficient it is, and most likely, the more human-friendly. Some compromise between programmability and execution efficiency has to be made. With the increasing use of computers in every aspect, including many mission-critical applications, the correctness of a program is essential. If there is a rigid mathematic reasoning process which could formally prove the correctness of the programs written in a programming language, we say that the programming language has good *provability*.

1.1.3 Our Proposal

We propose a new programming language called *T-Cham* [123, 124][2]. It extends the Chemical Abstract Machine (Cham) [29, 34] with (sequentially executed) transactions [4]. A Cham is an interactive computational model based on the chemical reaction

[2]It is a successor of our former programming language Multran [126, 127].

metaphor, where a computation proceeds as a succession of chemical reactions. The molecules (also known as *tuples*) for the reactions are floating and interacting in "chemical solution"—*tuple space*. A transaction is a piece of programming code which has the properties of ACID (*Atomicity*, *Consistency*, *Isolation*, and *Durability*) [4]. It could be written in any language, such as C, Pascal, or Fortran etc. The tuple space, where the molecules of the Cham reside and interact, is used to coordinate those transactions. A transaction may begin its execution whenever its execution condition is satisfied. A T-Cham program can be executed in a parallel, distributed, or sequential manner based on the available computer resources.

The T-Cham approach is inspired by the coordination idea [77] of Linda. On the one hand, the control part, which is responsible for coordinating concurrent reactions and communications, is based on tuple spaces. *Reaction rules* are used to specify when and which actions (transactions) can happen. On the other hand, the "computational part" could be written in any programming language, even T-Cham itself; thus, transactions can be nested, or are hierarchical. The execution of a T-Cham program starts from a special transaction called `root`—the *main transaction* of the program. It is the only transaction which could be exempt from termination (not really atomic). The transactions referred to by the reaction rules in a transaction, say T, are called *sub-transactions* of the transaction T. From the point of view of any transaction, its sub-transactions are merely atomic operators. Finally, a reaction rule also has its temporal logic interpretation [125, 127, 122], which makes correctness proofs possible for a T-Cham program.

As a newcomer to the parallel and distributed programming language community, T-Cham emphasizes *simplicity*, *abstraction*, *efficiency*, and a sound *theoretical background*. The chemical reaction model makes it easy to express concurrent or parallel tasks in T-Cham; the hierarchical transaction structure is good for program abstraction and refinement; the explicit declaration of tuple space will help the optimisation of data (tuples) and task distribution, and hence the efficient execution of T-Cham programs; finally, the temporal logic interpretation of reaction rules promotes program verification.

1.2 Major Contributions

The research work undertaken in this thesis consists of three main parts: (i) the design of the T-Cham programming language, (ii) the application of temporal proof system to T-Cham program verification, and (iii) a prototype implementation of T-Cham programs.

The Chemical Abstract Machine (Cham) is a mathematical model for computation. It is to T-Cham what the *Turing Machine* [163, 114] is to an imperative programming language, such as *Fortran*, *Pascal*, or *C*. To the best of our knowledge, T-Cham is the first *programmable* language based on the chemical abstract machine. T-Cham also introduces some new features, for example, termination conditions, bulk reaction operations (i.e., transactions), hierarchical chemical reaction sub-systems (i.e., hierarchical tuple spaces and nested transactions), molecule mapping, and mapping mask etc., to the original Cham model. They make the T-Cham programs more efficient and easier to manage than the original model.

T-Cham semantics observes the same operational semantics of the Chemical Abstract Machine [98]. We adopt a temporal logic proof system for T-Cham program verification. The purpose of the verification system is to show programmers how easy to prove the correctness of T-Cham programs. With the increasing use of computers in every aspect, including many mission-critical applications, of modern society, the correctness of a program is essential. There are many proposals for using rigid mathematic reasoning processes to prove the correctness of the programs, but they emphasize more on the theoretical sides and, quite often, scare programmers away. In this thesis, we will concentrate on the application of mathematic reasoning instead of the mathematic theory itself. Similarly, in our daily life, we always say directly that "$3+4$ is 7" instead of "*the third successor of the number* 0 *plus the fourth successor of the* 0 *is proven to be its seventh successor*" as done in a rigid algebraic way; otherwise, it is very likely that most of us cannot do calculation at all.

To justify our T-Cham programming language, we developed some prototype implementations. One implementation is on the top of the C-Linda programming language [123], where there is a logically shared memory, and another is directly using the C compiler on the Fujitsu AP1000 multicomputer, which is a distributed memory parallel computer. An early test was also on the CM-5 parallel computer [126]. Some implementation techniques, such as *task managers*, *task executors*, *task bidding*, and *bid history* array were developed and used. In addition, we also developed the *tuple space partition* and *duplication* algorithms to ease the bottle-neck problem on the task managers.

1.3 Outline of the Thesis

The thesis deals with the three major problems mentioned in the previous section in two steps: first the basic features of T-Cham and then the advanced concepts. The rest of the thesis is organised as follow:

Chapter 2 covers previous research work in this area. Background knowledge is also provided. In this chapter, we first discuss the parallel computer hardware development and their classification, and then, most of the chapter concentrates on the software aspects of parallel computation by studying some abstract programming models and programming languages which are closely related to T-Cham.

Chapter 3 discusses the motivations behind our research based on our observations of parallel programming. Basically, it answers the questions, such as, why do we need yet another new parallel programming language, why do we choose the chemical abstract machine as the underlying computational model, why do we need transactions, why are the transactions nested, why do we need temporal logic proof system?

Chapter 4 introduces the basic notations of T-Cham. We use EBNF (Extended Backus-Naur Form) to define the syntax of T-Cham programming language, while the semantics of the language is defined by informal explanation and examples. The chapter first gives a brief introduction to the EBNF notation conventions, and then the syntax and semantics of the basic T-Cham programming language. Finally, it concludes with a simple example showing how to program in T-Cham and how a T-Cham program works.

Chapter 5 gives a number of examples to illustrate T-Cham programming. Some of them are computation oriented programs, and the others are interactive ones. The effect of non-determinism is also discussed in this chapter.

Chapter 6 proposes an implementation model for T-Cham programs. It is a generic model, called *T-Cham Machine*, for MIMD (Multiple Instruction streams over Multiple Data streams) computer architectures. It is an extension to the *master/worker* parallel computation model, where there could be more than one master. The prototype implementation on the AP1000 multicomputer is also discussed, and the basic performance measurement data are also given.

Chapter 7 develops a basic temporal logic proof system for T-Cham programs. The chapter first introduces a temporal logic system, and then proposes the temporal logic model for T-Cham programs. A set of rules, which could translate T-Cham programs into corresponding temporal logic formulae, is also developed. Finally, we give examples of T-Cham program verification.

Chapter 8 deals with the advanced T-Cham notations. They are used to construct hierarchically structured T-Cham programs. Program abstraction and refinement are also considered. The central idea of this chapter is *tuple mapping*, which can be used to decompose a large tuple into a number of smaller sub-tuples or *vice versa*. The mapping applies hierarchical views to T-Cham tuples and provides a means of constructing subtransactions to T-Cham programs: a transaction consists of a number of subtransactions (or sub-reaction-systems), which are isolated from the other systems. When put together, they specify the whole reaction system, while any changes to a transaction (or subtransaction) are transparent to the others.

Chapter 9 discusses the composition of the temporal logic proof systems. In this chapter, we study the temporal logic theory of two kinds of transaction compositions, *union* and *superposition*, and their effects on the T-Cham proof system. The techniques developed in this chapter will help us to build a large proof system from a number of small systems the same way as building a large transaction from a number of small transactions in T-Cham programs.

Chapter 10 gives the final summary of our research work and suggests some future work.

Chapter 2

Background and Related Work

The pursuit of high performance (automatic) computation tools has been driven by the ever-increasing demand for computational power in real-life applications. In the early days of human history, our ancestors used their own fingers, perhaps toes, as well as pebbles and sticks to help them to count and calculate. The first human-made sophisticated computational tool is the *abacus* [107], dated back to 5,000 years ago in China. It dramatically increased the computational power of human beings in that time and is still used in China and some other countries. Some thousands of years had passed since the invention of abacus to the *Mechanical Adder/Subtractor* [15] of Blaise Pascal in France in 1642 and to the *Difference Engine for Polynomial Evaluation* [36] of Charles Babbage in England in 1827. The real breakthrough of computational tools in human history is the invention of electronic computers. One of the first computers is known as *ENIAC* [14], built at the Moore School of the University of Pennsylvania in 1945. ENIAC was a giant monster: with 18,000 vacuum tubes, 70,000 resistors, and 5,000,000 soldered joints; weighing 30 tons, occupying a 30×50 square feet room, and consuming 160 kilowatts of electrical power. But, an addition or subtraction operation took as long as 200 μs.

A major drawback of ENIAC was that its programs were hard-wired; in other words, the physical electronic circuit has to be changed if the program needs to be changed.

The characteristic is always used by some people to deny that ENIAC was the first *modern* computer. The first electronic computer[1] which has the same architectures, the so-called the *von Neumann structure* [101], as today's computers is *EDVAC* [101].

Since then, computer hardware has advanced from *vacuum tubes*, *discrete transistor circuits*, *integrated circuits* (IC), *large scale integrated circuits* (LSI), to *very large scale integrated circuits* (VLSI), the so-called five hardware generations. Although their underlying fundamental architectures are still of the von Neumann structure, the computational power of computers has been rapidly developed.

At the very beginning, programs were coded in binary machine language (strings of `0`s and `1`s), and a program occupied the resource of the whole computer. Those programs were very small. For example, the evaluation of a complex arithmetic expression was a big task at that time. Of course, on the other hand, the processing speed of a CPU was very low during that time, and it could handle only one task at a given time instant. With the increasing of processing speed and memory capacity, a computer is to be expected to manage multi-tasks at the same time. Time-sharing operating systems begin to appear, and multiprogramming, or concurrency, becomes possible. A computer then could handle more than one task simultaneously. Those tasks share the same physical computer in a time-sharing mode and work together in an unpredictable order. To be executed, a task has to be on a queue to wait for CPU. When its turn comes, the task receives a slice of CPU time. After it runs out of the time, no matter whether it finishes or not, it gives up the CPU to the others. If it needs more CPU time to finish the job, it has to rejoin the queue and wait for another turn. Because of the high CPU speed and the short period of the time slice, it seems that the tasks are running concurrently. A major difficulty of multiprogramming is the synchronization among those tasks, in a more technical term, *processes*. A fair scheduler is needed to schedule those processes; in addition, any of those tasks should be free from interference from the others. *Semaphores*, *conditional critical region protection*, and *monitors* [65, 86, 90] are the most important and also commonly used mechanisms for process synchronization.

[1]There exists some dispute about which was the *real* first, but as this thesis is not on the history of computers, we only take ENIAC and EDVAC as our examples.

The demand for computational power has increased further nowadays. The areas of computational fluid dynamics, subatomic string dynamics, high temperature superconductivity, astrophysical particle dynamic, plasma physics, computer chess, artificial intelligence and so on [74] are among those which give a big challenge to computational power of any single computer, because the development of the computer processing speed is restricted by the bounded electron transmission speed, which is the same as the speed of light, and the memory capacities are restricted by the integration density of a VLSI chip. The only way to answer the challenge is to combine individual computers and make them work together with the strength accumulated from each member of the computer system. Thus, parallel computation made its debut to take the challenge. The idea of parallel computation is, amazingly, very old. Hockney and Jesshope [92, pp 5–7] credited the honor to General Menabrea's publication in the *Bibliothèque Universelle de Genève*, October, 1842. It was still in the Charles Babbage era! However, it is only the electronic computers that make the dream of parallel computation come true.

2.1 Parallel Computer Systems

The computer architectures can be roughly classified into four classes, *SISD*, *SIMD*, *MISD* and *MIMD*, according to Flynn [68].

A single von Neumann computer is an SISD (*Single Instruction stream over a Single Data stream*) architecture machine. Conventionally, it is called a sequential computer.

An SIMD (*Single Instruction stream over Multiple Data streams*) architecture machine is also known as a *vector computer* or a *data parallel computer*, where a single instruction stream controls a number of vector processors which execute the same instruction on different elements of a vector. A vector is often called an array in a programming language. A major application of SIMD computers is matrix calculation.

A well-known example of MISD (*Multiple Instruction streams over a Single Data stream*) is the *systolic arrays* structure [106], where a same data stream flows through an array of processors which execute different operations on the streams.

An MIMD (*Multiple Instruction streams over Multiple Data streams*) computer system comprises of a number of individual computers, or computer nodes, which interact, i.e., communicate and synchronize, with each other and work concurrently. Each computer node, also called *processor element* (PE), is autonomous and can execute its own control thread independently. There may be some shared memory among those PEs, or not. If there is, it is called a *shared memory parallel computer system*, Figure 2.1(a), where the communication and synchronization are realised via the *shared memory*. The concurrent processes on different nodes share a global address space. Processes interact with each other by *reading*, *writing*, *locking* and *unlocking* a certain piece of the shared memory. If there is no such kind of shared memory, it is called a *distributed memory computer system*, Figure 2.1(b), where the communication and synchronization are achieved by a *message passing* facility.

Of the four architecture models, SISD is just a sequential computer; SIMD and MISD, parallel though, are more suitable for special purposes; only MIMD structure can be considered for general purpose usage. MIMD structure machines are also the most popular commercially available parallel computers. In this thesis, we constrain our terminology of *parallel computer* to MIMD machine, if without explicit explanation.

One of the first MIMD computers was delineated in the paper of Slotnick *et al* [158], where the authors described a new computer architecture: a 32×32 array of processing elements, each with a memory of 128×32 bit numbers. A processing element is an autonomous computer with its own CPU, arithmetic unit and memory. The new computer was called *SOLOMON*. Although it was never built as the authors wished, it catalysed *ILLIAC IV* [150].

There are many commercially available MIMD computers nowadays. The two computer systems we used to conduct our experimental implementation of T-Cham are *AP1000* [13] and *CM-5* [55, 54].

2.2 Parallel Programming Models and Languages

Programming languages play a vital role in program development and implementation. On the one hand, a programming language provides a means of algorithm presentation; on the other hand, it reflects the underlying abstract computational model. For example, an imperative programming language reflects the Turing machine, and a

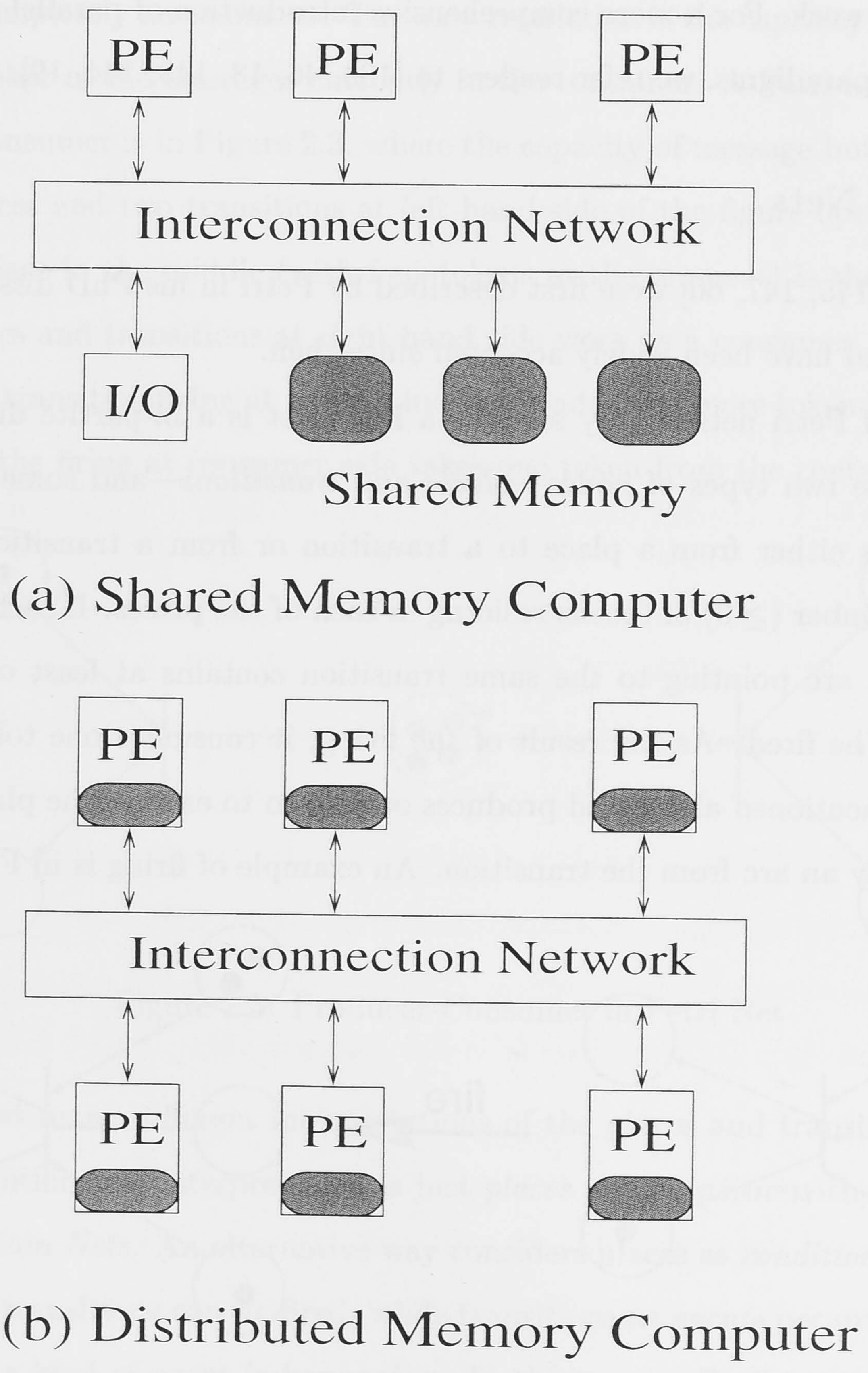


Figure 2.1: Shared Memory versus Distributed Memory Computer Architectures

functional programming language reflects the λ-calculus.

We give a brief introduction to parallel programming models and languages in this section. The models and languages listed here are by no means exhaustive, but are all directly related to our work. They are introduced in order to outline the background of our research work. For a more comprehensive introduction of parallel programming languages and paradigms, we refer readers to [103, 46, 18, 143, 144, 19].

2.2.1 Petri Nets

Petri nets [145, 147, 60] were first described by Petri in his PhD dissertation [146] in the 1960s and have been widely accepted since then.

The idea of Petri nets is very simple: a Petri net is a bi-partite directed graph, where there are two types of nodes—*places* and *transitions*—and some arcs, each of which connects either from a place to a transition or from a transition to a place. There are a number (≥ 0) of tokens residing in each of the places. If each of the places which have an arc pointing to the same transition contains at least one token, the transition can be fired. As the result of the firing, it consumes one token from each of the places mentioned above and produces one token to each of the places which are connected to by an arc from the transition. An example of firing is in Figure 2.2.

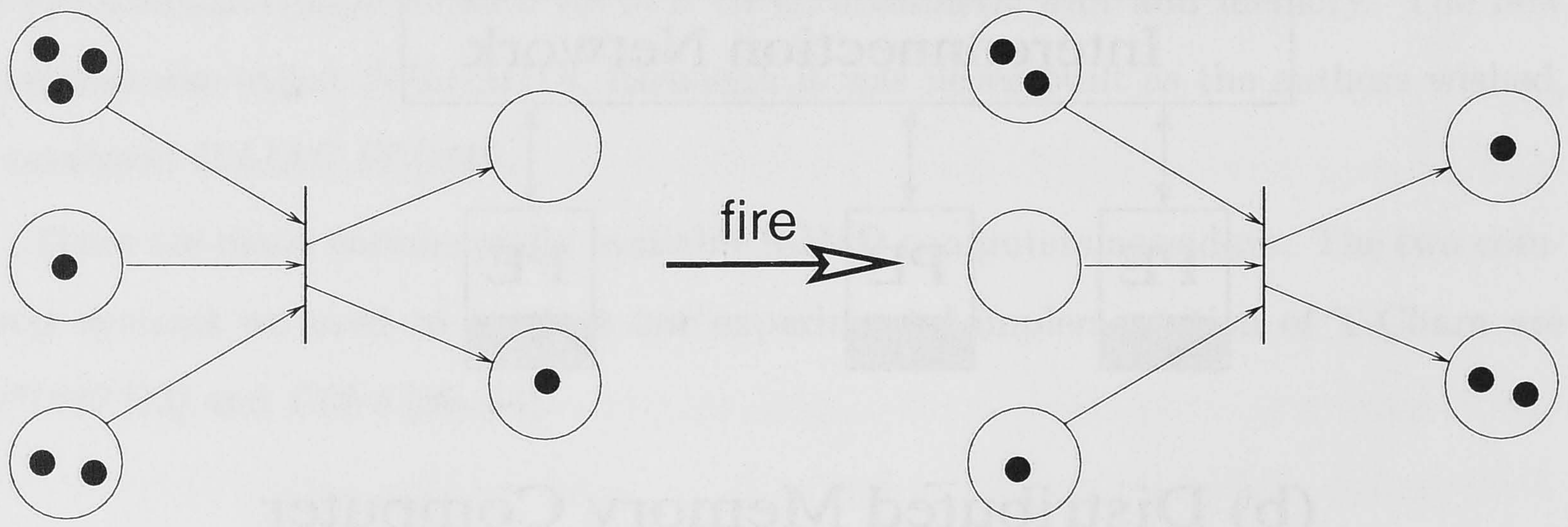


Figure 2.2: The Firing of a Transition in a Petri Net

There is no restriction on the orders of transition firings; thus, a Petri net is an inherently concurrent model. A simple example of concurrency is the *Producer-Consumer* problem: a producer produces a product—say, a message—at a time to a container, while a consumer consumes a message, at a time, from the container. The producer constantly produces messages to the container until it is full, meanwhile the consumer constantly consumes the messages from the container until it is empty. When it is full, the producer suspends until there is more room available; similarly when the container is empty, the consumer is waiting for an available message. The producer and the consumer are completely autonomous. The only restriction is the capacity of the container and the number of the messages currently in the container. A Petri net version of the Producer-Consumer is in Figure 2.3, where the capacity of message buffer is unlimited. The two places and two transitions at left hand side of the figure operate as the producer, the place in the middle (with four tokens at the moment) is the container, and the two places and transitions at right hand side work as a consumer. It is quite easy to see that a transition firing at the producer side adds one more token to the container place, while the firing at consumer side takes one token from the container place.

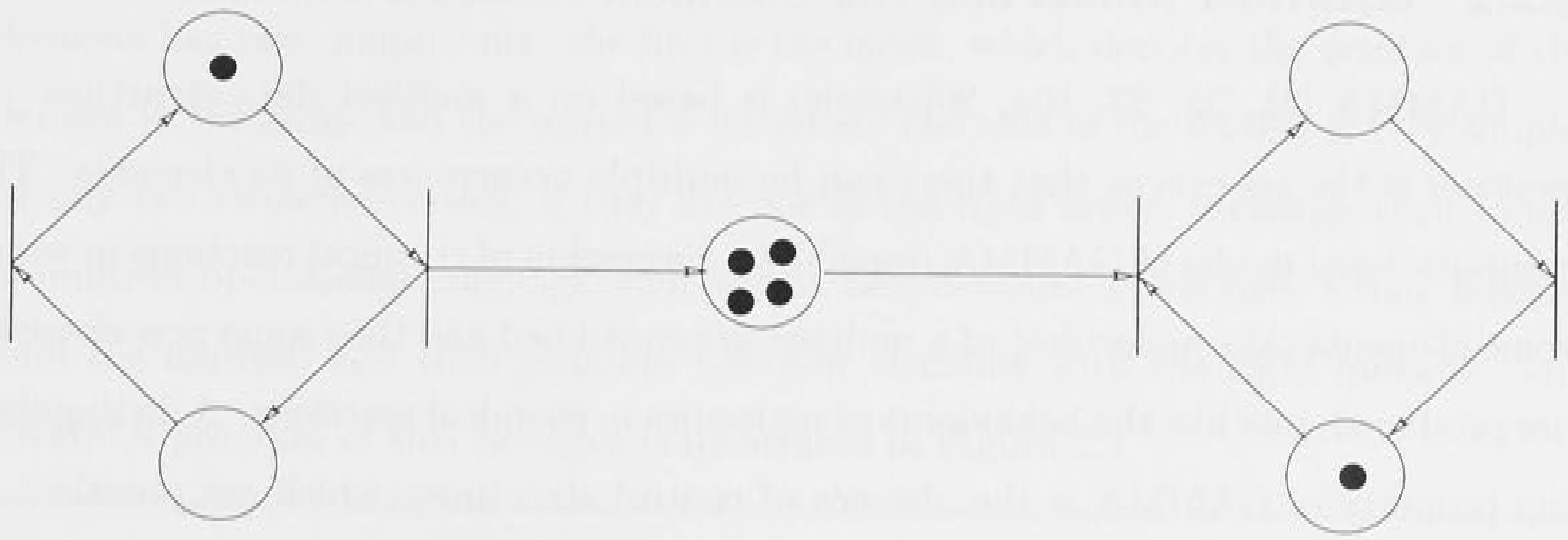


Figure 2.3: Producer-Consumer in Petri Net

There exist many different interpretations of the places and transitions in a Petri net. A straightforward interpretation is just *places* and *transitions* themselves, called *Place/Transition Nets*. An alternative way considers places as *conditions* because they decide which transitions can be fired, while transitions as *events* because firing a transition means a kind of event is happening. In this case, a Petri net is interpreted as a *Condition/Event System*. A third way treats places as *predicates*, and thus, *Predicate/Event Nets* [147].

There are also lots of extensions to the original Petri net. A most important one is so-called Colored Petri nets [99], where tokens are not identical any more but belong to different types or colors; transition firing conditions are composed by not only the numbers of tokens but also the types of tokens. The others, for example, assigning a transition some actions, setting time restriction on tokens or transitions, or adding extra conditions on each arc so that some tokens cannot pass through etc., cannot be exhaustively listed in this thesis.

The most important contribution of Petri nets is the recognition of resource-like tokens, which can be consumed and produced like real resources. This property is lacking in the conventional programming languages. We believe that there should be two kinds of variables in a program: one is the normal "variable" and the other is "resource". Those two are equally important. The difference between them is not so significant in a sequential program, but without fully understanding of the difference, a concurrent program is very hard to harness (see Section 3.1.5 for the detailed discussion).

2.2.2 GAMMA Model and the Chemical Abstract Machine

GAMMA [21, 22, 23, 104, 20] model is based on a multiset data structure. A *multiset* is the *set* except that there can be multiple occurrences of its elements. The computational model of GAMMA resembles a succession of chemical reactions in which some elements (aka molecules) of a multiset are consumed and then some new elements are produced, just like the behaviours of molecules in chemical reactions. A distinguishing property of GAMMA is the absence of control structures, which are prevalent in the imperative programming paradigm. Compared to the logic and functional programming paradigms, GAMMA model has a very simple structure, and it can reveal parallelism easily.

A GAMMA program is a set of transformation rules, also known as reaction rules, which transform the original multiset to a new multiset. Programmers need only to consider the two possible states of a multiset: the original state (a multiset of tuples) and the new state produced after the action. A GAMMA program is defined as the function:

$$\Gamma(R,A)(\mathcal{T}) = \texttt{if } \exists x_1, x_2, \cdots x_n \in \mathcal{T} \texttt{ such that } R(x_1, x_2, \cdots, x_n)$$
$$\texttt{then } (\mathcal{T} - \{x_1, x_2, \cdots, x_n\}) + A(x_1, x_2, \cdots, x_n)$$
$$\texttt{else } \mathcal{T},$$

where $\mathcal{T}$ is the original multiset before the action, and the operators $+$ and $-$ denote multiset union and difference. The boolean type function R is the *reaction condition*. When it is evaluated to the value TRUE, the function A, known as an *action text*, is activated, and after its execution, it returns a multiset of elements as the result of the action.

An example program is the sorting of an array $A[1..N]$, which is decomposed into a multiset of N elements, $(1, A[1])$, $(2, A[2])$, $\cdots$, $(N, A[N])$. Each of the multiset elements has two components: the first is the index, which denotes the position of the element in the array, and the second is its value. The idea of the sorting is very simple: for any two elements chosen, if they are not in the right order, exchange their index. In multiset operational language, remove the two elements which have wrong indexes from the multiset and then generate two new elements with the right indexes. The GAMMA program of this example is illustrated in Figure 2.4.

$$R((i,x),(j,y)) = i > j \wedge x < y$$
$$A((i,x),(j,y)) = \mathcal{T} - \{(i,x),(j,y)\} + \{(i,y),(j,x)\}$$

Figure 2.4: Sorting an Array in GAMMA

Although GAMMA model is powerful yet concise in expressing program logic, its implementation on today's computers is not so efficient [21]. The major reason is, ironically, due to the lack of control structures in GAMMA model. Take the sorting program in Figure 2.4 for example: as there is no control on how to choose tuples (i, x) and (j, y) to evaluate the reaction condition R, it is highly possible that the two tuples chosen are already in the right order, i.e., $i < j$, and this may happen over and over.

A lot of effort has been taken by the research community to improve the efficiency of the GAMMA model. Hankin et al. [84, 85] proposed a serial of refinement and equivalence rules based on the IO behaviours of GAMMA programs. With the help of the rules, a simple GAMMA program can be rewritten into an equivalent but less nondeterministic (or more deterministic) program, therefore, increasing the efficiency of the program. Ciancarini et al. [48] took a different approach by suggesting the refinement and equivalence rules based on bisimulation of CCS [136]. Chaudron and Jong [44], on the other hand, tried to impose some orders on the reactions of GAMMA programs. They resort to an external schedule to reduce the nondeterminism of a GAMMA program. Weichert [169] incorporate the approaches of both Hankin [84, 85] and Chaudron [44] by proposing a "pipelining" technique to refine GAMMA programs.

In this thesis, we take yet another completely different approach to improve the implementation efficiency of GAMMA model. By realizing the continuous dominance of the von Neumann structure in modern computer architectures, we use self-contained and atomic imperative program segments—transactions—to pack more operations into reactions. Please see Chapter 3 for more detailed discussion.

Cham (Chemical Abstract Machine) [29, 34] is a theoretical refinement of GAMMA. The rigid mathematical definitions of molecules, reactions and reaction rules are given, and so are the structured molecules and their transformation rules. Recently, Cham has also been used to specify a multi-phase compiler by Inverardi and Wolf [98].

2.2.3 The Linda Paradigm

Linda [42, 40, 7] is the first coordination parallel programming paradigm based on a global tuple space. There are a couple of fully implemented Linda languages. The most popular one is C-Linda [52]. Since its debut, Linda remains as an active research area. For the history of Linda and its possible future development, we refer readers to [30], and for the up to date research work on Linda, we refer readers to the Linda Group at Yale University [64]. Coordination [77] is the basic idea promoted by Linda. Instead of simply mixing different languages together, Linda provides a global shared tuple space to coordinate the activities of each individual programming languages. We will discus the idea of coordination in Section 3.2.3.

There are four tuple space operators in Linda:

1. `in`: withdraws a tuple from the tuple space if it exists; otherwise the action is blocked until the tuple is available. If there are more than one tuple of this type available, one of them is chosen arbitrarily;

2. `rd`: has the same functionality as `in` except that the tuple is not deleted from the tuple space;

3. `out`: outputs a tuple to the tuple space;

4. `eval`: outputs a tuple containing at least one field of active data, which needs to be executed before the result can be reached. For example, we call "3" and "4" *passive data* while "3 + 4" an *active data*. A tuple which contains active data is called an *active tuple*; otherwise a *passive tuple*. Any active tuple will eventually evolve to its passive form, for example, "3 + 4" to "7".

Linda is not a full-fledged programming language. It can only coordinate the activities written in other ordinary (sequential) programming languages. According to different choices of underlying computational languages, we can get C-Linda, Fortran-Linda, and Pascal-Linda etc. Taking C-Linda as an example, the activities, i.e., the chunks of computation, are written in the C programming language. They interact and communicate with each other on the tuple space by the four Linda operators. A C-Linda program of summing two arrays $B[0 \cdots N-1]$ and $C[0 \cdots N-1]$ to $A[0 \cdots N-1]$ pairwise is given in Figure 2.5, where the elements of the two arrays are first injected into the tuple space by the `for` loop, and then the `summation` function does the pair-wise summation. If we have a function `sum(x,y)` which returns the summation of its two arguments, the `out` statement of `summation()` can be replaced by "`eval(ArrayA, i, sum(x,y));`". The latter version reveals more parallelism.

The elegant idea of coordination becomes awkward when the four tuple space operators reside in a sequential host language. The sequential skeleton of the host language forces programmers to consider its sequential control structures first instead of the concurrently accessible tuple space. Furthermore, the syntax structure of an existing host language also blurs the globality of the tuple space.

A better way to realise the idea of coordination is to view the tuple space operations

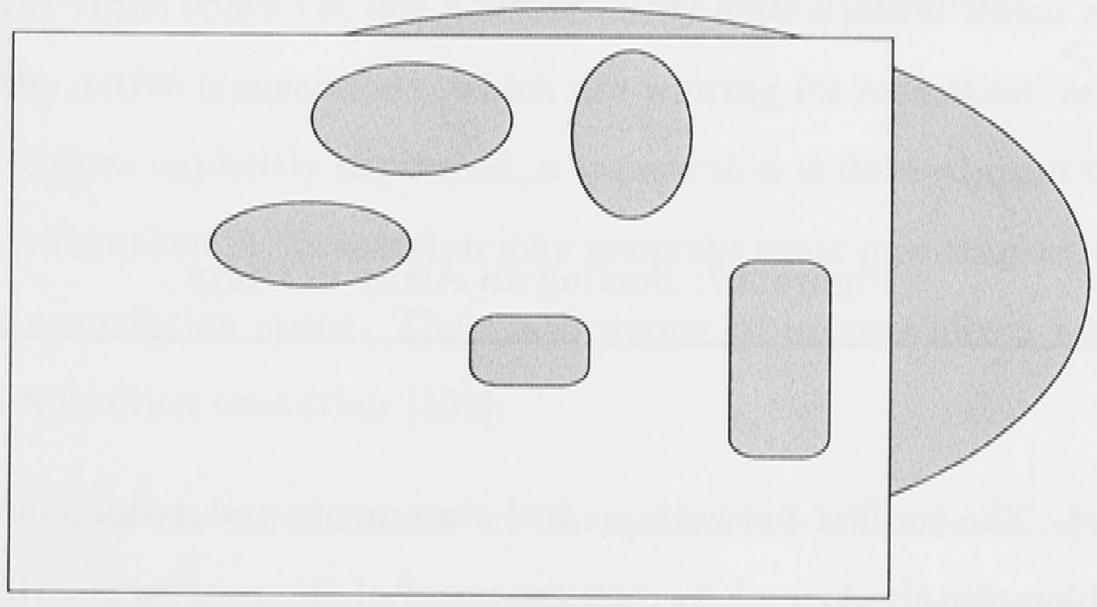
```
main()
{
        int B[N]={...}, C[N]={...};
        int A[N];
        int i;
        for (i=0; i<N; i++) {
                out(ArrayB, i, B[i]);
                out(ArrayC, i, C[i]);
        } /* for */
        summation();
} /* main */

summation()
{
        int i;
        for (i=0; i<N; i++) {
                in(ArrayB, i, ?x);
                in(ArrayC, i, ?y);
                out(ArrayA, i, x+y);
        } /* for */
} /* summation */
```

Figure 2.5: The Pairwise Summation of Two Arrays in C-Linda

as a skeleton with the computational chunks as pieces of flesh which are fitted onto this skeleton. T-Cham promotes this approach. A comparison of Linda and T-Cham is illustrated in Figure 2.6, where in Linda approach, the tuple space (the shadowed area) can only be seen through the holes of a sequential programming language front-end, while in T-Cham approach, the tuple space is in front of the computational chunks. In short, rather than extending a sequential computational programming language by adding a parallel tuple space, we attach the sequential computational chunks to a concurrent accessible tuple space.

2.2.4 Unity and Swarm

Unity is based on Dijkstra's do [66] structure and has no control statements: it retains the assignment statement of the imperative programming paradigm but abandons



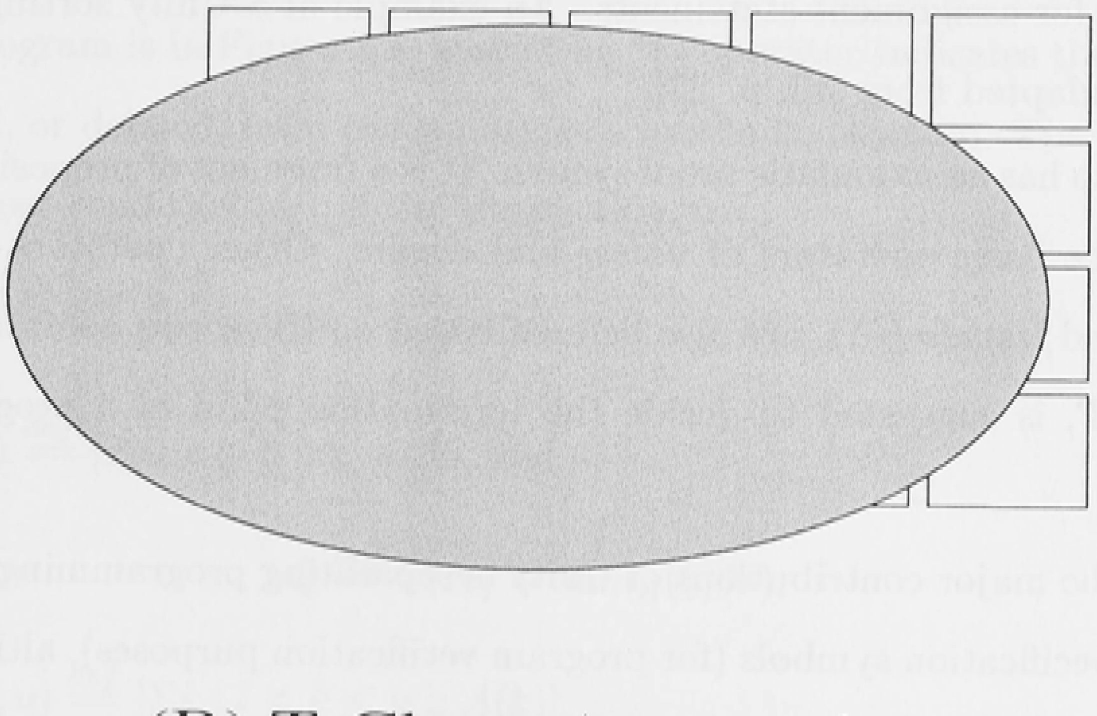


Figure 2.6: An Observation on the Tuple Space

```
Program sort3
    assign
        < ▯ $j : 0 \le j \le 1$ ::
            <|| $i : 0 \le i < N \wedge j = i \bmod 2$ ::
                $A[i], A[i+1] := A[i+1], A[i]$    if $A[i] > A[i+1]$
            >
        >
end {sort3}
```

Figure 2.7: Sorting an Array in Unity

its control part. The conflict between control statements and assignment statements is the main problem for the formal correctness proof of the programs written in imperative programming languages [16]. A Unity program consists of a group of assignment statements which are executed infinitely and fairly. A statement can assign different values to different variables in one step. "||" is used as the sub-assignment separator, and "▯" for assignment statements. An example of a Unity sorting program is in Figure 2.7, adapted from [43, p. 33].

Unity also has an axiomatic proof system. It is a fragment of propositional temporal logic with the basic operators of *unless* and *ensure*. Other operators, such as *stable*, *invariant*, and *leadsto* ($\mapsto$), are also defined based on those two operators. A fix point operator, *FP*, is suggested to decide the termination point of a program, if it does terminate.

One of the major contributions of Unity is separating programming notations from its formal specification symbols (for program verification purposes), although there is a one-to-one relationship between them. The verification is transparent to the programmers who do not like mathematical reasoning, but the correctness of a program can still be proved by some other people who do like such reasoning.

Swarm [152, 153] improves Unity by introducing *dataspace*, *transactions* and *synchronous groups* into the language. The name Swarm [152] evokes a swarm of

"large (number), rapidly moving aggregation of small, independent agents cooperating to perform a task."

The dataspace of a Swarm program consists of a *tuple space*, a *transaction space*, and a *synchronous relation space*. The tuple space is used to store the data needed for program execution. It is the same as Linda tuple space. A Swarm transaction is an atomic action upon the tuple space. It has a set of *query-action* pairs, which are executed in parallel. All the active transactions, which are waiting for execution, are in the transaction space. Unless explicitly expressed, a transaction is deleted from the transaction space after its execution. A transaction may generate some new transactions and insert them into the transaction space. The synchronous group provides a means to specify the order of transaction execution [152]:

> *"When one of the transactions in an equivalence class is chosen for execution, then all members (transactions) of the class which exist in the transaction space at that point in the computation are also chosen."*

Swarm also introduces the concepts of *pre-conditions* and *post-conditions* to a transaction. They are very useful in program verification. An example of a parallel array summation program is in Figure 2.8, where the "†" operator indicates the tuple before it is consumed, or deleted, from the tuple space by the transaction. The pre-condition (P) and the post-condition (Q) of the transaction are:

$$P \overset{def}{=} pow2(N) \wedge j = 1 \wedge [\forall i : 1 \leq i \leq N :: x(i) = A(i)],$$

where $pow2(k) \overset{def}{=} [\exists p : p \geq 0 :: k = 2^p]$, and

$$Q \overset{def}{=} x(N) = sum_A(0, N),$$

where $sum_A(l, u) \overset{def}{=} [\Sigma k : l < k \leq u :: A(k)]$.

2.2.5 Argus and the Transaction Programming Paradigm

Argus [116, 117] is the first instance of integrating the atomicity property of transactions into the fundamental programming concepts at the *programming language level.*

An Argus program consists of a group of *guardians.* A guardian is actually an *atomic* object, which encapsulates and controls the access to and the operations on its data, or *resources.* By atomic objects we mean that an action completes by either

```
**program** *ArraySum(N, A: [ ∃ p: p ≥ 0 :: N=2^p ], A(i: 1 ≤ i ≤ N))*

    **tuple types**
        [ i,s: 1 ≤ j < N :: x(i,s) ]

    **transaction types**
        [ j: 1 ≤ j < N::
            Sum(j) ≡
                [ ‖ k: 1 ≤ k ≤ N ∧ k mod (j * 2)=0::
                        v1,v2: x(k-j,v1)†, x(k,v2)† ⟶ x(k,v1+v2)
                ]
                ‖ j < N ⟶ Sum(j * 2)
        ]

    **initialization**
        Sum(1); [ i: 1 ≤ i ≤ N :: x(i,A(i)) ]

**end**
```

Figure 2.8: A Parallel Array Summation Program in Swarm

committing or *aborting*. A committed action successfully finishes its operations and brings the object to a new state, while an aborted action has no effect at all just as if the action had never happened. The inside of any guardian is composed of some private (i.e., for this guardian only) data and a number of processes which perform the operations on the data. The operations are organized in procedures. There is a special kind of procedure called a *handler*, which can be called by other guardians and provide the operations on the data of this guardian on behalf of them. The handler calls are the only channels of communication and synchronization among the guardians.

A guardian resides at a single computer node (not necessarily a physical one) and can survive the crashes on this node because of its atomicity property. Guardians in different nodes coordinate, communicate and synchronize via handler calls to work on a computation task. In other words, guardians are logical computers, while handlers are the communication network among them.

Argus was originally designed for the implementation and execution of distributed systems, like a bank system, on an unreliable computer network. Examples of Argus programming can be found in papers [116, 117].

The atomicity property was originally defined in database management systems (DBMS) to protect data from corruption caused by concurrently executed transactions. From the point of view of a programmer, all the operations within a transaction are executed in "exactly" one step without any interruption and molestation. The atomicity property sets a clear boundary around a task. It is also proved to be a very useful property for parallel programming: programmers can concentrate on atomic tasks of a parallel job without worrying about the interferences among them. Every task begins its execution as soon as its execution condition is satisfied. The concurrence among those tasks is decided by the test of their execution conditions.

Atomicity is also a desirable property for program verification. Manna and Pnueli use the concept of *grouped statements* to make a number of conventional programming language statements in a program to be uninterruptable, i.e., atomic [130, Chapter 1]. By grouping some already grouped statements together, a larger grouped statement is constructed, and hence, different granularities of atomicity can be achieved. If the atomicity property can be introduced into a programming language, there is no need for grouping the statements, and it will be easier for both programming and program verification.

2.2.6 Strand, PCN and Bilingual Programming Languages

Strand [73, 70, 72] (later *PCN* [71, 69]) is a bilingual programming language claimed to solve the problems of *portability*, *expressiveness*, *efficiency*, and *compatibility* with existing software.

The motivation of Strand is based on a straightforward observation: a very-high-level[2] programming language, such as a logic or functional one, or even higher, has two highly desirable properties, *scalability* and *portability*, for parallel programs, as they free the programs from the details of computer architectures. There are no concepts such as variables (alias memory cells), control structures (alias the changing of program

[2]We use *very-high-level* to refer to languages which is in a level higher over the traditional imperative programming languages, such as the C programming language, which is at a *lower-level* according to Foster in the paper [72], but we'd like call it as a *high-level* programming language.

counter—PC—in the control unit of a computer), and assignment (alias the moving of data among memory cells). A programmer thus can concentrate on the problem solving strategies and does not have to have the knowledge of underlying computers and the way of program execution.

In contrast to those nice properties, on the other side, these kinds of very-high-level languages are normally poor, at least, related to a high-level one, in implementation efficiency. There is no theoretical reason which accounts for the phenomenon but today's techniques favour the high-level programming language because it is closer to the current computer structures. Efficient as a high-level programming language is, it involves too many details of underlying computer hardware. This reduces the scalability and portability of a program, especially in the parallel programming situation.

Naturally, we may ask if we can combine a very-high-level programming language with one or more high-level languages, which together can make a new programming paradigm. The very-high-level language would be responsible for the logic of a program while the high-level language(s) would be responsible for the computationally intensive tasks of the program. In this way, we can keep the scalability and portability without sacrificing much of efficiency. The Strand approach gives a *yes* answer to the question as put by Foster and Overbeek [72]:

> *"The key idea in bilingual programming is to construct the upper level of applications in a high-level language while coding selected low-level components in low-level languages. This approach permits the advantages of a high-level notation (expressiveness, elegance, conciseness) to be obtained without the cost in performance normally associated with high-level approaches. In addition, it provides a natural framework for reusing exiting codes."*

Four basic ideas contribute to the design of Strand. They are *single-assignment variables*, *concurrent processes*, *non-deterministic choices* and the *separation of sequential code*. A single-assignment variable can be assigned and referred to, respectively, once and only once. It can be used to synchronize two concurrent processes which share the same single-assignment variable, or do the communication between the two processes[3]. A running Strand program has a number of concurrent processes, which

[3]This kind of variables is known as *resources* in a T-Cham program. We will discuss the benefit of

are non-deterministically chosen to be executed. The execution of a process is on a computer node in a sequential mode. The process which is written in a sequential programming language is known as foreign code to a Strand program. *User-defined operations* and *user-defined data type* are used to encapsulate foreign codes and foreign data, respectively. In other words, they are the interface between the higher-level part and the lower-level one in a Strand program.

PCN (Program Composition Notation) enhances Strand in three ways: (i) PCN introduces program composition concepts, (ii) PCN has richer and more flexible syntax, and (iii) PCN supports the implementation and the use of reusable *parallel* modules.

A PCN version of Producer-Consumer program is given in Figure 2.9. The procedure `stream_comm` in the first line of the program creates two processes, `stream_producer` and `stream_consumer`, and set the maximum number (`N`) of messages which could be produced by the producer (`stream_producer`). The `stream_producer` continuously creates a new message called "`message`" before the counter `N` gets to 0, and the `stream_consumer` prints the message out.

2.2.7 The DINO Programming Language

DINO (DIstributed Numerically Oriented language) [154] is an SIMD programming language built on the top of the C programming language. DINO provides a top-down description of a distributed parallel algorithm: a programmer first defines a virtual machine which fits the problem best and then maps it to a real machine. The three essential concepts of DINO are *environment structures*, *distributed data* and *composite procedures*. An environment is a virtual processor which contains the same procedures and data structures. There is one—and only one—special environment called *host* and an array of ordinary environments in each DINO program. The host environment acts as the master of all the others. Distributed data are used to map the global data structure of an algorithm into the structures of environments, i.e., virtual processors. The mapping determines how an environment accesses and shares those data. It also makes the inter-process communication implicit. The idea of distributed data can be simply and logically considered as partitions on data arrays. A composite procedure is

distinguishing them from normal variables later in Section 3.1.5.

```
stream_comm(n)
{
    || stream_producer(n,x), stream_consumer(x)
}

stream_producer(n,out)
{
    ?
        n > 0 ->
            {|| out=[Message|out1], stream_producer(n-1,out1)
            },
        n == 0 -> out=[]
}

stream_consumer(in)
{
    ?
        in ?= [Message|in1] ->
            stdio:printf("%s\n", Message), stream_consumer(in1),
        in ?= [] -> stdio:printf("STOP\n")
}
```

Figure 2.9: Producer-Consumer Program in PCN

a set of identical procedures, each of which resides in an environment.

A simple example of matrix-vector multiplication is in Figure 2.10. The matrix in the example is partitioned, or distributed, by rows and each processor calculates one element of the result vector.

There are three kinds of distributed data in the `node` environment: `BlockRow`, `All` and `Block`. The first one, `BlockRow`, maps one row of the matrix `M[n][n]` to each of the `n` node environments, the second maps the whole vector `v` to each node environment, and the third maps one element of the vector `a` to each node environment. `BlockRow`, `All` and `Block` are the pre-defined DINO mapping functions. The `main` function in the host environment calls function `MatVec`, which is a composite procedure defined in `environment node[n:id]`, to fork the same process on each of the environments. The suffix `#` symbol indicates a remote name reference as `MatVec` is not defined within the `host` environment itself. The `MatVec` in each environment is a row-wise algorithm to calculate the production of matrix `M` and vector `v`.

A major contribution of DINO is its distributed data structures, or the environments, and the mappings between them. They provide abstract data structures to a parallel program. A programmer can define the abstract structures which fit exactly the problem to be solved. The problems with DINO are that the mapping to a real computer is not going to be easy; it is also hard for DINO to manage the data which are not in array structures, and finally, the new distributed data structures are mixed together with the data structures of the C programming language itself, which makes the programming even harder and more confusing.

2.2.8 Others

There are many other approaches that we have not mentioned in the previous sections. It does not mean that they are not important, but are not *directly* related to our work. This section gives a brief introduction to some other very important achievements in this area.

In 1978, Hoare introduced the idea of *CSP* (Communicating Sequential Processes) [91]. Originally, CSP was not intended to be used as a programming language but as a medium to study a system with co-existing, i.e., concurrent, activities. Three

```
#define n       512
#include        "dino.h"

environment node[n:id]
{
    composite MatVec(in M, in v, out a)

    float distributed M[n][n] map BlockRow;
    float distributed v[n] map All;
    float distributed a[n] map Block;

    {
        int j;

        a[id]=0;
        for(j=0; j<n; j++) a[id] += M[id][j]*v[j];
    }
}

environment host
{
    main()
    {
        long int i,j;
        float Min[n][n];
        float vin[n];
        float aout[n];

        MatVec(Min[][],vin[],aout[])#;
    }
}
```

Figure 2.10: Matrix-Vector Multiplication Program in DINO

major issues of a concurrent system were addressed. They are *parallelism*, *communication* and *non-determinism*; therefore, any programming language for these kinds of systems—concurrent, distributed and parallel—should have the abilities to express parallelism, communication and some extent of non-determinism. CSP directly gives birth to the programming language *Occam* [120, 121], which was developed by Inmos Ltd. for their *transputers* [119]. CSP also has had a very strong influence on Ada [140, 51]. Actually, CSP can be regarded as the common antecedent of all message passing programming languages.

CCS (Communication and Concurrent System) [136] is a concurrent model developed by Milner in the 1980s. The most important thing in CCS is the *communication behaviours* of a number of agents, unlike all the other traditional approaches, where the main concerns are the activities of the agents[4] themselves. Two concurrent systems are considered exactly the same if their communication behaviours are the same by the external observations. The communication behaviours among the agents via their ports are similar to chemical reactions. There is an equivalence between CCS and the Chemical Abstract Machine [34, 29].

The π-calculus [138] is an algebra also developed by Milner to study the behaviours of the interactive systems which consist of the interactions among its different agents. The equivalence relations, say, bisimulation, of the π-calculus are the theoretical foundations for transforming an interactive system to another equivalent system, in term of their semantics. As the thesis does not focus on program transformation, we won't discuss the details of the π-calculus. We refer interested readers to [137, 139, 138].

The Game Semantics [2, 3, 1] is a theory which studies the interactive actions of a system. In this system, there are two participants, a player `P` and an Opponent `O`, where `O` represents the environment, while `P` is the player in the environment. `O` provides stimuli, and `P` responses to those stimuli. The theory is another attempt taken by the research community to study program behaviour from the point of the view of interaction.

PRAM (Parallel Random Access Memory) [135, 9] is perhaps one of the oldest parallel programming models. It has four varieties: EWER (Exclusive Write and Ex-

[4]An agent can be considered as a program module, or an object.

clusive Read) model, EWCR (Exclusive Write and Concurrent Read) model, CWER (Concurrent Write and Exclusive Read) model, and CWCR (Concurrent Write and Concurrent Read) model. The concepts of EWER, EWCR, CWER, and CWCR help a programmer to be aware of the concurrent nature of read and write activities in a parallel program. However, PRAM is less useful in the context of parallel programming languages because it ignores communication costs. The main contribution of PRAM was in the complexity analysis of parallel algorithms.

2.3 Conclusion

In the parallel software development environment, there is a plethora of parallel programming paradigms, models, languages which have been proposed by research communities and computer manufacturers, yet few of them has been widely accepted. Parallel programming techniques are still in the same situation as sequential programming in the early days of computer history: relatively powerful hardware versus clumsy programs of assembly programming languages. The research communities and manufacturers are aware of this situation, but still cannot get rid of the dilemma of efficiency vs manageability. If the programming language is too close to a parallel computer hardware structure, we get efficiency but lose manageability. Programming becomes notoriously hard to handle, and it is impossible to port any of this kind of programs across different platforms. While on the other hand, if the programming language structure is far higher than the parallel computer hardware structures, the programs can be easy to write—an extreme example is a super-compiler which can extract parallelism from an existing sequential program—but it may not always be easy to be implemented efficiently. It is sometimes even worse than the performance of a sequential program on a sequential computer. The major research in this area is to find a balance point, where the programming is manageable by ordinary programmers, while not sacrificing much efficiency. It is this goal that motivated our proposal of T-Cham. In the next chapter, we will discuss our observations and motivations.

Chapter 3

Motivation

The basic idea behind T-Cham is very simple. We believe that the GAMMA model (or Chemical Abstract Machine) is an ideal underlying computational model for broadband computer programs, especially for the so-called[1] parallel and distribution problems. But due to the lack of control structures (although it is the feature which makes GAMMA model concise and powerful), it is very difficult to efficiently implement the programs based on GAMMA model. To improve the implementation efficiency, the research community has proposed different ways of program transformation, which rewrite a program into different formats or structures. The later version of the program is equivalent to the original one, but more efficient. Section 2.2.2 has more detailed discussion about the latest research work in this area. In this thesis, we propose a different approach to attack the implementation efficiency problem. We realise the predominance of the von Neumann structure computer architectures nowadays, and the trend may still be kept for another decade or so. We believe it is not wise to completely abandon the imperative structures in programs. As long as we are aware of the problems caused by imperative programming structures [16] and try to avoid the problems as much as possible, we shall be able to enjoy the efficiency of implementation

[1]We believe that the concepts of sequential, parallel, or distribution programming do not belong to programming language level. GAMMA model has no mention of the concepts either.

while still preserve the essential properties of GAMMA. Based on the belief, we propose using imperative program segments, which are self-contained and atomic, to pack up more operations into the reactions of GAMMA model. This kind of program segments are called transactions. As a transaction can have more or less operations and executes in an atomic manner, i.e., like a single basic operator, we can enjoy the efficient implementation of transactions without destroying the beauty of GAMMA model. By combining GAMMA model (Chemical Abstract Machine) and transactions together, we propose the T-Cham programming language.

Essentially, a T-Cham program is just a group of transactions wrapped by the reaction rules of GAMMA model. In other words, the top level of the program belongs to GAMMA model, while the lower level of the program consists of transactions. The top level is responsible for the coordination of the reactions (i.e., the transactions of the lower level) and the logical correctness of the program. The lower level is responsible for the computationally intensive tasks. The portion of the top level and the lower level in a T-Cham program can vary from no transactions at all (a pure GAMMA model program) to a program with a trivial top level which consists of a reaction rule with only a one-off reaction and a big single transaction. The purpose of the only reaction on the top level is to start the big single transaction. Virtually, the later program is just a conventional imperative program. Figure 3.1 gives a diagram of the spectrum of T-Cham programs, which spread from GAMMA to a conventional imperative programming language (the C programming language).

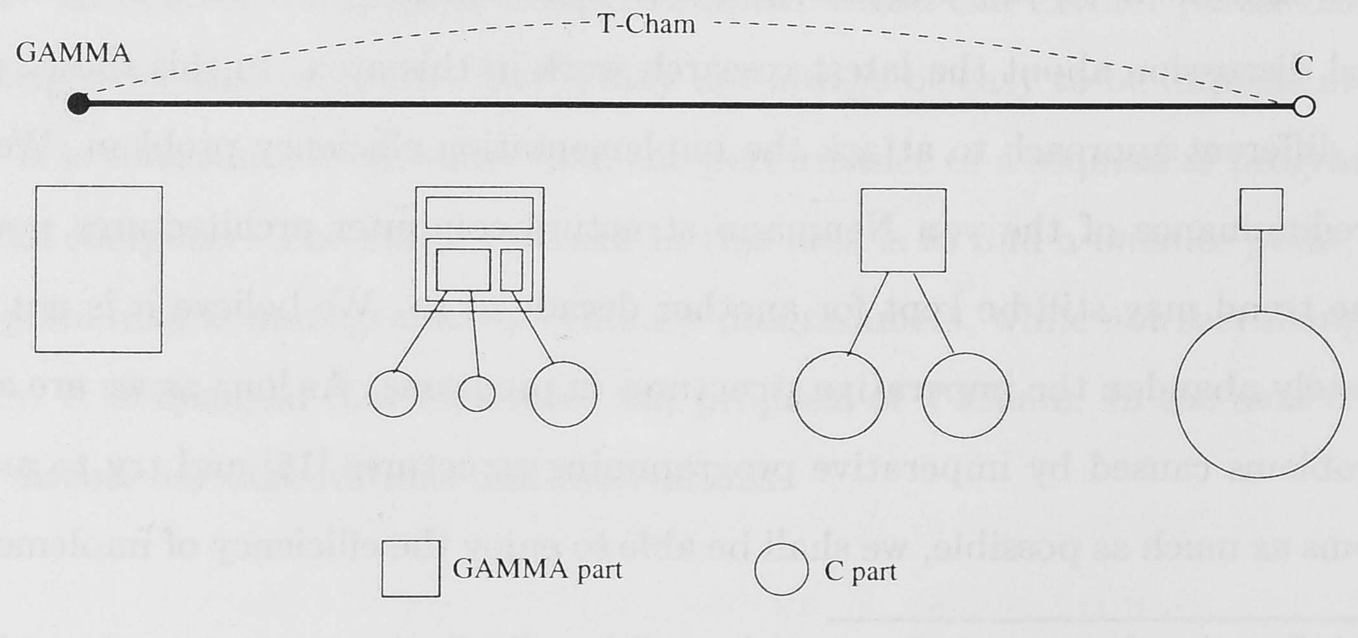


Figure 3.1: The Spectrum of T-Cham between GAMMA and C

The idea of combining different programming paradigms or languages into one programming language to enjoy the benefit from each of the paradigms or languages, for example, easy to program, simple to prove the correctness of the programs, and good performance etc., can also be found in [115, 72, 73, 71]

In the rest of the chapter, we will discuss the motivation of our work in more details. The proposal of the T-Cham programming language is initially motivated by our observations on programming models and languages. The components of T-Cham are chosen as the result of those observations. We first discuss those observations and then answer questions on the choice of T-Cham components.

3.1 Observations on Parallel Programming

3.1.1 Functionality versus Interaction

A functionality program[2] is the one which maps an input into an output. Functionality is the traditional way of thinking about the behaviour of a sequential program. The *Turing machine* [163, 114], which is the foundation of all the imperative programming languages, works in this way. A program starts from the starting point of the program with initial data and then halts at the ending point with the resulting data of this computation. Functional programming [16, 88, 96] and logic programming [93, 118] carry out the same idea. A functional program is a mathematical mapping, which maps a type of data to another type of data, while a logic program starts from a query and ends with an answer (or answers) to the query. Although in functional and logic programming situations, the execution order is not necessarily sequential, the basic idea is still of functionality.

A functionality program fits well in a sequential computer. When it comes to a parallel computer, which is built by connecting sequential computers together with a communication network, the idea of functionality still dominates the method of programming, because it is straightforward to think that the difference between a sequential program and a parallel one is in the number of control threads. A parallel program has more than one control thread so that it can reach the termination point and get

[2]A functionality program is distinct from a functional program based on λ calculus.

the results faster than a sequential program. In practice, the approach emphasizes the design of control threads, see Section 3.1.2 for more details. The synchronisation and data exchange among those control threads are implemented by message passing or the shared memory, see Section 3.1.3. The implementation of this idea, in both theory and practice, becomes awkward when there are tens of thousands of control threads.

A reactive program emphasizes the interactions among the components of a program: different parts of a program interact with each other in response to the stimuli from outside world. The phenomenon was noticed as early as the beginning days of concurrent programming and operating system design. That is why Dijkstra introduced the *Dining Philosophers* problem, but it takes a very long time for computer scientists to accept the "new" idea in the parallel programming area, according to Lamport [111]:

> *"Computer scientists originally believed that the big leap was from sequentiality to concurrency. ... We have learned that, as far as formal methods are concerned, the real leap is from functional to reactive systems."*

where the honour of this discovery was credited to Harel and Pnueli [87].

Milner [136] also realised the problem in his CCS, where "*interaction or communication is the central idea*". Ciancarini [47] has the same observation by distinguishing a *closed* program from an *open* one. Abramsky's Game Semantics [2, 3, 1] is another theory emphasizing the interactive actions of a system. Wegner [168] also advocates this idea.

The concepts of *sequential*, *concurrent*, *parallel*, or *distributed* should belong to the execution of a program on particular computational resource instead of the program itself, Figure 3.2. At the programming language level, we should not focus on sequential, concurrent, parallel, or distributed programs but only *functional* and *interactive* programs. Accordingly, any non-sequential programming language should not be judged only by its abilities of thread control and communications but also the abilities to express the interactions. In other words, a concurrent programming language should not be just an extension to an existing sequential programming language with some thread control and communication facilities but a new one based on an interactive computation model.

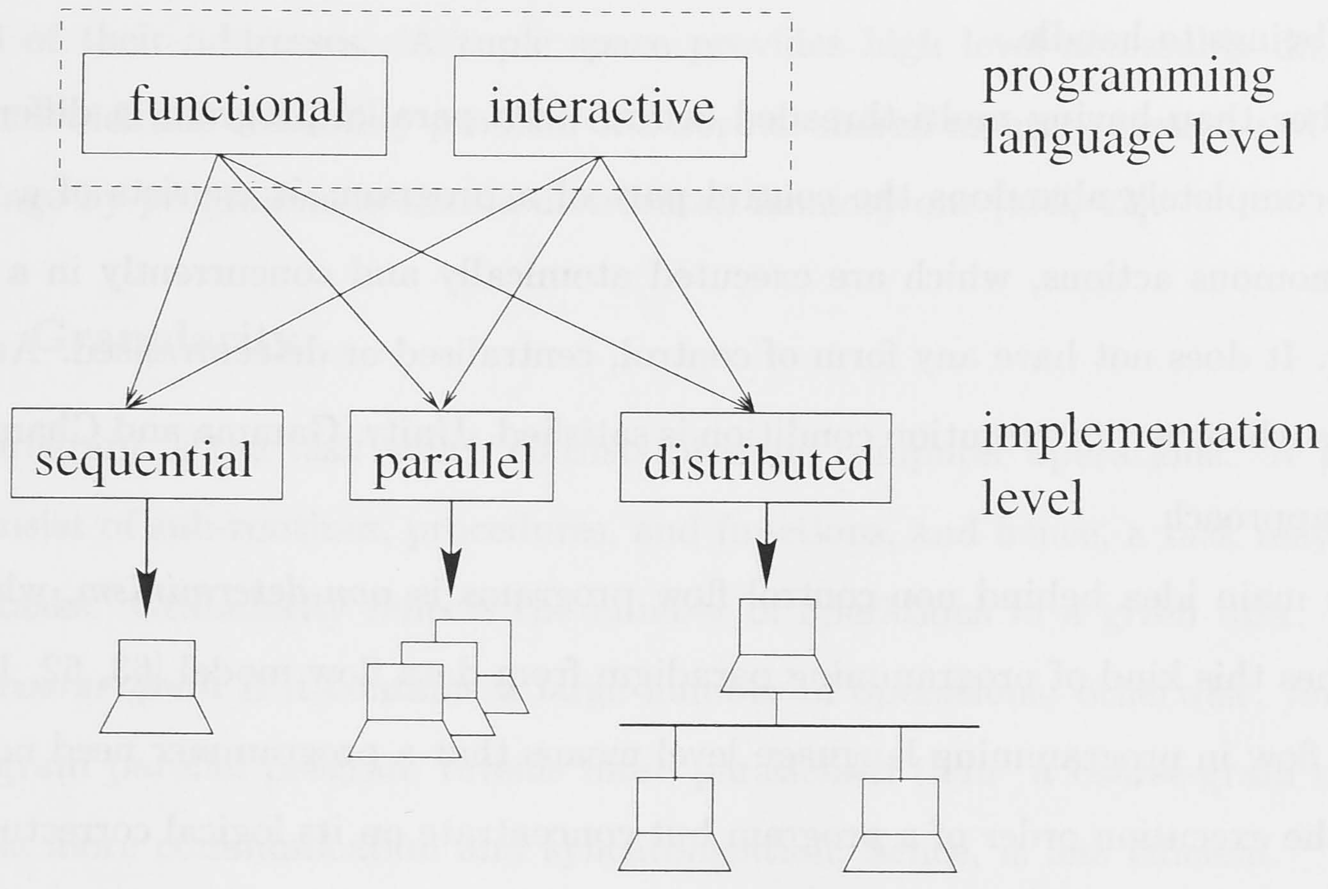


Figure 3.2: The Conceptual Levels of Programming

3.1.2 Single Thread, Multi-thread and Non-thread

Control flow and *control thread* are two different concepts in the description of a program. Control flow relates to a static program. It can be built on three basic constructs, *sequence*, *branch* and *goto*[3]. During execution, only one branch of a branch construct can be chosen at any time on a given set of data. The operation sequence of the execution of a program is unrolled by control thread. Control thread is a concept related to the execution of the program.

A sequentially executed program has only one control thread, which unrolls the control flows of a program step by step in a sequential manner. When the program goes to a parallel machine, more than one control thread may exist concurrently. It is possible to unroll the operations of a program in a parallel manner. To achieve this goal, a programmer needs to consider and write down the strategies of thread control, for example, the *creation*, *termination*, and *join* etc. of threads. At the first sight, multi-threading is a natural way to go from sequential to parallel programming, but when the number of processing elements in a parallel computer scales up, the physically available threads become very large; therefore, thread control becomes very difficult for

[3]The three basic constructs are the complete set for programming [57].

human beings to handle.

Rather than having multi-threaded control in a parallel program, a different approach completely abandons the control part of a program: it consists of a number of autonomous actions, which are executed atomically and concurrently in a chaotic manner. It does not have any form of control, centralised or de-centralised. An action happens whenever its execution condition is satisfied. Unity, Gamma and Cham belong to this approach.

The main idea behind non-control flow programs is *non-determinism*, which distinguishes this kind of programming paradigm from data flow model [63, 62, 159]. No control flow in programming language level means that a programmer need not worry about the execution order of a program but concentrate on its logical correctness. For example, a sorting algorithm can be concisely described as "*choosing any two elements and sorting them in a right order*". In most cases, non-control flow paradigm provides a natural way to describe the logic of an algorithm.

On the other hand, a non-deterministic program cannot be implemented so efficiently as a deterministic one. Too much non-determinism may lead to inefficiency. What we need to do is to find a balance point between non-determinism and determinism: use non-determinism whenever it is necessary while use determinism whenever it is possible. The same observation was also noticed by Beguelin and Nutt [26].

3.1.3 Shared Memory versus Distributed Memory

The concepts of *shared memory* and *distributed memory* originally come from computer architectures, Figure 2.1. As most of parallel programming languages still cannot free programmers from computer hardware architectures, the same concepts apply to them as well. It is a prolonged discussion on which one is better [42]. The proponents of shared memory programming paradigm claims it is easy to handle while the opponents criticize its bad scalability. In the distributed memory case, synchronizations and data exchanges are achieved by message passing mechanism. The pros and cons are exactly in a reverse order as those of shared memory.

The third approach is an *associatedly accessible* logical shared memory—tuple space. Tuples in a tuple space are accessed by pattern-matching on their content instead of their addresses. A tuple space provides high level accessable distributed data. It avoids the scalability problem of a normal shared memory system and is easier to manage by programmers than a distributed memory one [170, 42].

3.1.4 Granularity

A program is the task which consists of some computer operations. A program may consist of sub-routines, procedures, and functions, and hence, a task may consist of sub-tasks. Granularity reflects the number of operations in a given task. We call a task *coarse-grain* if it contains a large number of operations; otherwise, *fine-grain*. A fine-grain parallel program reveals more parallelism than a coarse-grain one, but demands more communication and synchronization; hence, is less efficient. "*There's no reason in theory why this kind of program can't be supported efficiently, but on most current parallel computers there are substantial overheads associated with creating and coordinating large numbers of processes*" [41, p. 33]. How to choose the right granularity depends on the problem to be solved and the computer resources available. An ideal way to do this is to apply fine-grain first to explore all potential parallelism and then amalgamate several fine-grain tasks into a coarse-grain task to reduce unnecessary communication overheads. The procedure is called *grain packing* [97, pp. 61–70]. It is one of the most difficult tasks in parallel programming. We believe the difficulty coming from the lack of autonomy in task description. In other word, the tasks are interwoven together.

3.1.5 Variables versus Resources

Some of the best things in our life are free. Truth is free. You can tell the truth of "*The earth is orbiting the sun*" as many times as you like. Some other things are not. Food is not free. Whenever you bake a loaf of bread, you can only enjoy the loaf once. After the bread is eaten, you have nothing left. The first kind of fact is known as *information* while the latter *resources*.

Traditionally, in a sequential programming language, *variables* are used to hold both information and resources. We may use a variable, say `pi`, to hold the value of π, which is $3.1415926\cdots$. The variable `pi` can be accessed repeatly. But when we deal with bread, we have to use another variable, say `loaves`, to record how many loaves of bread we have. For example, after you bake a loaf of bread, the variable `loaves` is increased by 1, and after you eat one, the variable `loaves` is decreased by 1. This variable works well in a sequential execution situation. In other words, you are just yourself in a closed world and isolated from others. You bake and you eat.

In the concurrent case, using a simple variable to hold a resource-like thing is troublesome. For example, if we have two processes (persons) to consume the bread:

```
Process_1:                            Process_2:

    a0: ...                               b0: ...

    a1: if (loaves>0)                     b1: if (loaves>0)

    a2:     loaves := loaves-1;           b2:     loaves := loaves-1;

    a3: ...                               b3: ...
```

when `loaves=1`, a possible execution path may be `a1b1a2a3b2b3`, which consumes *two* loaves of bread from *one*!

To avoid this unreal outcome, the visit to a shared variable, `loaves` in this case, is regarded as a *critical section* and is indivisible. No other actions are allowed to cut into the sequence of `a1a2a3` or `b1b2b3`. Some techniques, such as *semaphores* and *monitors* etc., were developed to protect the sections from the intervention of other co-existing processes.

By introducing the resource concept, where a resource can be produced and consumed, to a programming language, programs will be much easier to be written and understood. We believe that Petri net [145, 147, 60] is the first one to study the consumption and production of resources. Linda [42, 40, 7] and PCN [71] (the so-called *single-assignment* programming language) are among the other very few programming languages which have the mechanism to deal with resources.

Coincidentally, the same observation on the importance of resources in a logic system[4] was spotted by Girard when he studied the classical logic. Even more interesting, the report [79] of this discovery, known as *linear logic*, was not published in a logic or a philosophy journal but in the *Theoretical Computer Science*, which suggests that it

[4]The traditional logic system does not have the concept of resources, either.

is closer and more important to computer science than philosophy or logic.

3.1.6 Debugging versus Verification

Testing and debugging a program are real challenges to a programmer. In the sequential situation, there exists a lot of development tools and environments, for example, `adb`, `sdb`, `dbx` and `lint` etc. in a Unix system, Turbo environments (Turbo C, Turbo Pascal, and Turbo C++ etc.) in PCs, to help a programmer to do the job. In general, those tools and environments provide a means to trace out the execution behaviors of a program on a set of input data. They can set break points on, check the run-time environments and modify dynamic data values of the program. Despite the abundant auxiliary tools and environments, the debugging is still a painstaking and time-consuming procedure.

Unlike in the sequential situation, where the execution path—the order of the operations taken by a CPU—of a program remains the same on the same set of input data no matter how many times the program is executed, a parallel program has no such property. Take the bread baking-eating procedures in Section 3.1.5 for example: the execution order could be `a1a2a3b1b2b3`, `b1b2b3a1a2a3`, `a1b1a2a3b2b3`, or `a1b1b2b3a2a3` etc. on a same set of input data with different executions. Those paths are not predictable. The basic reason is that whenever there is more than one CPU involved, the communication time between any two CPUs is unpredictable, the drift of the synchronization (hardware) clock pulses inside of each CPU is unpredictable, and the exact processing speed of each CPU is unpredictable either; furthermore, the use of registers and cache contributes even more uncertainty.

Lots of effort have been made to guarantee that parallel programs only take acceptable execution paths. Many parallel programming environments and tools have been developed. Most of them use intuitive visual tools, so-called *program visualization* [171, 151, 134, 25, 27, 81], to reveal the static structures and the dynamic activities of a parallel program. To get the execution behavior data of a parallel program, we have to do instrumentation, i.e., inserting some extra instructions into the program and using the data captured by those instructions to recover the execution path of that program. The dilemma is that the instrument instructions themselves always change the run-time behavior of a parallel program [151, 128]. Bearing the uncertainty mentioned before in mind, an extra instruction may change the starting time of a communication session, cause more CPU and cache time, and even trigger chain reactions from other processes. All in all, it is extremely difficult to locate and fix a bug in a parallel program.

Automatic program verification is a long desired goal in the program development community. We wish that one day, after we finish the programming work, a mathematical system could prove the program is correct. A great deal of achievement has been made in this area. Some of them become more and more mature and applicable to real applications. A good example is that Clark, Grumberg and *et al* using temporal logic model checking method proved the correctness of a couple of IEEE communication protocol standards and found a few bugs in IEEE Futurebus+ standard (IEEE Standard 896.1-1991) [49, 50]. Given the fact of that IEEE standards are carefully designed and well debugged by the *élite* of the related areas, bugs are still not avoidable. To conclude, we'd like to cite Dijkstra's famous words: "*program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence*" [66, p. 20]. In contrast, a formal verification system can prove the absence of any bugs.

3.2 Motivation: Questions Answered

Programs should be judged by their logic operations instead of the execution order of these operations. In other words, the concepts of *sequential*, *parallel*, or *distributed* should not be a main issue of programming. They are the issues of program implementation on a particular computer system. We believe that a program should only contain the description of the logic of the program.

3.2.1 Why Yet Another Parallel Programming Language?

A large number of parallel programming languages have been proposed in the last two decades. Why do we suggest yet another parallel programming language? As discussed in Section 3.1.1, parallel activities have inherently interactive nature, and hence a parallel programming language should reflect this kind of nature. In other words, it should have the ability to easily express the interactions among the components of a program. To the best of our knowledge, no such kind of parallel programming languages have been proposed yet. Most of the parallel programming languages proposed in the past are the result of extending the existing sequential programming languages with thread control primaries (operators or statements). Those approaches emphasize the difference between multi-thread and single thread execution of a program, but have no concept of interaction. This kind of approach makes the parallel programming much harder than it should be. Analogously, it is just like using a spanner to drive a screw. To try a different approach, we suggest a new programming language based on an interactive abstract computation model, the Chemical Abstract Machine (Cham).

T-Cham extends the Chemical Abstract Machine (Cham) with transactions. A Cham is an interactive computation model based on chemical reaction metaphor, where a computation proceeds as a succession of chemical reactions. The molecules (also known as *tuples*) for the reactions are floating and interacting in a solution (*tuple space*). A transaction is a piece of programming code which has the properties of ACID (*Atomicity*, *Consistency*, *Isolation*, and *Durability*). It could be written in any programming language. The tuple space, where the molecules of the Cham reside and interact, is used to coordinate those transactions. A transaction may begin its execution whenever its execution condition is satisfied.

The reason why we design T-Cham in this way is discussed in the following sections.

3.2.2 Why the Chemical Abstract Machine?

Abstract computation machines play very important roles in program description and implementation, for example, the *Turing machine* to an imperative programming language, e.g., Pascal or C; *Warren Abstract Machine* [167, 8] to a logic programming language, e.g., Prolog, and its implementation; *SECD* [88, 102] to a functional programming language, i.e., Lisp, and its implementation.

The *Chemical Abstract Machine* (Cham) provides a natural and easy way to express the *interactions* among the components of a program. It also has formal operational semantics, much easier to understand than a declarative one. As discussed in the previous sections, interactions are the central focus of a parallel programming language. Given the interactive nature of a Cham, it is the most qualified candidate to be the abstract computation model for parallel programming languages.

3.2.3 Why Coordination?

We always try to assemble well-behaved constructs together to build better tools, but a simple assembling method introduces interference among these different components and is liable to create untamed complex "monsters", such as *PL/1* [12] and the *Algol68* [35, 161] programming language as well as the *Multics* [67] operating system.

The basic idea of coordination [77] is *orthogonally* gluing together: different parts are *orthogonally* glued together to let the final product take advantage of each individual part while without suffering from interference among the parts.

Orthogonal coordination maintains the independence of each component. Adding or removing one component has no effect on the others.

3.2.4 Why Tuple Space?

A tuple space is a logically shared memory used for data exchange and synchronization control among the interactive components of a program. Unlike traditional data structures, a tuple space is inherently distributed and naturally offers parallel access. Parallelism specification and implementation thus become much easier.

A hierarchical tuple space structure provides different abstract views and a means of refinement to a T-Cham program. It can be used to localise a group of tuples and their reactions, i.e., dividing a global tuple space to a number of smaller sub-tuple-spaces. Each of the sub-tuple-spaces is relatively independent to the others.

The tuple mapping mechanism transforms one tuple (or a group of tuples) to another (or another group of tuples). With tuple masks, a tuple can have many different appearances to meet different requirements.

Unlike the tuples in a general tuple space, where they are of the same generic data type (just known as tuples), in T-Cham, the tuples in a tuple space belong to different types, for example, an integer tuple, a real number tuple, or a tuple of a compound structure. For the detailed discussion about tuple types and their declaration, we refer readers to Section 4.3.1. The purpose of introducing types to the tuples is twofold. On the one hand, it makes the optimisation of tuple spaces easier, and hence, better implementation efficiency. On the other hand, the typed tuples can reduce some potential programming errors. R. van der Goot et al. made the same arguments when proposing Blossom [165], a strongly typed tuple space C++ version of Linda.

3.2.5 Why Transactions?

A transaction[5] is a piece of self-contained program code which has the properties of ACID (*Atomicity*, *Consistency*, *Isolation*, and *Durability*) [4] and executes sequentially on a computer node. The atomicity property of a transaction means that the transaction is regarded as an un-dividable single step operator, no matter how big it may be. Consistency and isolation actually stand for the same property: a transaction is a closed system and won't be affected by the change of the context it is in. Durability means that the effect of the transaction, when it is committed, won't be rolled back. If we only look at a single transaction, the property seems so obvious. When talking about many transactions running concurrently, durability is essential for the correctness of the transaction system and their efficient implementation.

Just as a parallel computer system is a number of sequential computer nodes, which are suitable for the efficient execution of program code in a sequential manner, bundled together by a communication network, a T-Cham program is a number of sequential tasks (transactions) bundled together by a chemical abstract machine.

With the concept of transactions, task granularity can be easily adjusted by changing the operators contained in the transactions, for example, a transaction can do a very complex function (coarse-grain), or only a simple summation operation (fine-grain). The changing of one transaction is isolated from the others; furthermore, the orthogonally integrated transactions can be re-used from program to program.

3.2.6 Why Theoretical Background?

Although mathematics and logic are the better way to achieve a correct program as discussed in Section 3.1.6, most programmers are not so comfortable with the rigid

[5]It is a leaf transaction. See Chapter 4 for more precise definition.

process of mathematical reasoning. People tend to use a natural and intuitional way to express their ideas. For example, people prefer the *Venn Diagrams* [53] to the mathematical definitions of the set operations.

Formal temporal logic semantics provides a means of correctness proof for T-Cham programs, but the proof system is separated from programming, or kept in the background. T-Cham programming notations serve as the Venn Diagrams in set theory, while the temporal logic interpretation of a T-Cham program is like the mathematical definitions, by which the reasoning is carried out.

3.2.7 Why Program Composition?

The experience of program development suggests that a large program should be constructed from a number of smaller components. The formal proof systems for program verification also prefer this kind of composition property [24]. Like in Unity, we consider two kinds of transaction (program) compositions, *union* and *superposition*. The union is used to juxtapose the corresponding sections of two different T-Cham transactions, while the superposition is responsible for the layers, or a hierarchical structure, of the final transaction. We also study their effects on the T-Cham proof system.

A T-Cham program can be constructed by the *union* and/or *superposition* of transactions. The union combines two small transactions into a big one, while the superposition makes a transaction to be a sub-transaction of another one. With union and superposition composition, a large T-Cham Program can be built from a number of small transactions; besides, the composition also provides the modular and abstract views to T-Cham programs.

Chapter 4

Basic T-Cham Notations

A T-Cham program consists of a number of transactions. A transaction which does not have any sub-transactions is called a *leaf transaction*; otherwise, a *non-leaf transaction.*

A leaf transaction could be a C function (or other programming language units) with the enhancement of the transaction concept made it ACID (*Atomicity*, *Consistency*, *Isolation*, and *Durability*) [4]. The execution of a T-Cham program starts from a special non-leaf transaction `root`, called *main transaction* of the program. It is the only transaction which may not terminate. The transactions referred to by the reaction rules in a transaction are called *sub-transactions* of it. From the point of view of any transaction, its sub-transactions are atomic operators.

The chapter first gives a brief introduction to the extended BNF (Backus-Naur Form) conventions, and then the syntax and semantics of the basic T-Cham programming language. We use extended BNF for the syntax description of the T-Cham programming language, while plain English for its semantics. We conclude this chapter with a simple example showing how to program in T-Cham and how a T-Cham program works.

4.1 Notational Conventions

In the thesis, we adopt the following notational conventions [6]:

1. Strings and characters with typewriter fonts, for example, `if`, `then` and `else`, are terminals, and a double-quoted string or symbol, e.g., "`{`", means the string itself.

2. Strings and characters with Roman (italic) fonts and beginning with capital letters, for example, *Expr* and *Stmt*, are nonterminals.

3. Lower-case Greek letters, for example, α, β and γ, represent strings of grammar symbols.

4. If $A \to \alpha_1$, $A \to \alpha_2$, $\cdots$, $A \to \alpha_k$ are all productions with the same nonterminal A on the left, they can be written as $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$ for short.

5. Square brackets denote an optional part of a production, for example,

$$Stmt \;\to\; \texttt{if}\ Expr\ \texttt{then}\ Stmt\ [\texttt{else}\ Stmt]$$

6. Braces denote a phrase which can be repeated *zero* or more times, for example,

$$Stmt \;\to\; \texttt{begin}\ Stmt\ \{;\ Stmt\}\ \texttt{end}$$

7. Braces followed by a + denote a phrase which can be repeated *one* or more times, for example,

$$Stmt \;\to\; \text{"}\texttt{\{}\text{"}\ \{Stmt;\}+\ \text{"}\texttt{\}}\text{"}$$

Under the convention, the grammar of simple one digit arithmetic (plus and minus) expressions can be specified as follows:

$$\begin{aligned} E &\to E + E \\ E &\to E - E \\ E &\to \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \end{aligned}$$

A T-Cham program consists of a number of leaf and non-leaf transactions. The grammar is

$$Program \rightarrow \{NonLeafTrans \mid LeafTrans\}+$$

where *Program* is the start symbol, and *NonLeafTrans* and *LeafTrans* are nonterminals standing for non-leaf transactions and leaf transactions respectively. We will discuss the details of them in the consequent sections. For the complete definition of T-Cham syntax, we refer readers to Appendix A.

4.2 The Essential Components of a Chemical Abstract Machine

T-Cham is a programming language based on the Chemical Abstract Machine [29, 34], where a computation proceeds as a succession of chemical reactions in a chemical reaction system. In T-Cham, a program can be considered as the specification of a "chemical reaction system". To be able to describe a chemical reaction system, T-Cham has to have the ability of specifying the essential components of a Chemical Abstract Machine.

Take an ordinary chemical reaction system[1] as an example:

two hydrogen molecules and one oxygen molecule make two water molecules:

$$2H_2 + O_2 = 2H_2O$$

Let's study the essential components a chemical reaction system must have. For example, a certain amount of hydrogen molecules and oxygen molecules are in a container, a reaction happens whenever the reaction condition is satisfied, and the reaction stops when there are no hydrogen molecules (at least two of them) or oxygen molecule left. When we introduce this chemical reaction system to computers as a model for computation, the essential components include:

- a **container**, which contains all the molecules of this chemical reaction system;

[1]The details of a chemical system and chemical reactions are certainly not the focus of this thesis. We only discuss them based on the common sense for analogy purpose only.

- the **types** of molecules possibly appearing in the reaction system, e.g., hydrogen, oxygen, and water molecules;
- the **initial status** of the container, i.e., how many of each type of molecules, and the status of those molecules before any reaction;
- a number of **reaction rules**, which specify how and when a reaction proceeds and the impact of the reaction on the molecules;
- the **termination conditions**, i.e., when the reactions stop. The hydrogen and oxygen reaction example above terminates when there are no hydrogen molecules (at least two of them) or oxygen molecule left.

If we look at a reaction from different points of view, such as those of atoms, electrons, and so on, different actions can also be spotted. For example, in the hydrogen and oxygen reaction system, at the electron level, we can see how the outer layer electrons of the two hydrogen molecules and one oxygen molecule react to each other. In other words, a chemical reaction system may also have

- multiple **views** of a reaction from different levels.

To be able to specify the programs of the Chemical Abstract Machine model, the T-Cham programming language has the corresponding components as well.

The *tuple space* of the T-Cham is actually the "container", which contains all the molecules, called *tuples* in T-Cham.

A T-Cham program consists of a number of leaf and non-leaf transactions (chemical reaction systems). Inside of a non-leaf transaction, there are `tuples`, `initialization`, `reactionrules`, `termination`, and `sub-transactions` sections. They are, in turn, corresponding to the **types** of molecules possibly appearing in a reaction system, the **initial status** of the container (tuple space), the **reaction rules**, the **termination conditions**, and the different **views** of this reaction system discussed above.

4.3 Non-leaf transactions

Syntactically, a non-leaf transaction is composed of (i) the specification of its tuple space, including the types of tuples and the initial state, (ii) reaction rules, and (iii) sub-transaction interfaces—pre-conditions and post-conditions of its sub-transactions. The tuple space specified is visible to the transaction and its sub-transactions only.

A non-leaf transaction consists of a name and a body:

NonLeafTrans	→	`transaction` *NName NBody* `endtrans`
NName	→	*PlainIdentity*
NBody	→	*Tuples Init React* [*Term*] [*SubTrans*]

The keywords `transaction` and `endtrans` are used to encapsulate this non-leaf transaction. The *NBody* is the transaction body and contains information about participating molecules (*Tuples*), the initial state of those molecules (*Init*), the reaction rules (*Reac*), termination conditions (*Term*), and the actions carried out by those rules (*SubTrans*)[2]. We call each of them a section. The last two sections, i.e., termination and sub-transaction, are optional.

4.3.1 Tuples

The tuples section declares all possible tuple types (molecules) which may appear in the tuple space of this transaction. The declaration only specifies the type of possible tuples. How many of the declared tuples, when, and where they enter the tuple space depend on a particular execution and cannot be predicted in advance. In general, the specification starts with a keyword `tuples` followed by a list of tuple declarations. The keywords `fifo`, `filo` and `random` are used to specify the order of tuple consumption (the default is `random`), and `tuple` is used to declare a tuple. If a tuple has only one field, the key word `tuple` can be omitted.

Tuples	→	`tuples` *TupleDcl* { *TupleDcl* | *MaskDcl*}
TupleDcl	→	[*DclHead*] *DclBody*
DclHead	→	`fifo` | `filo` | `random`
DclBody	→	*Type NameList* ;

[2]From now on, a sub-transaction is called a transaction for brevity if there is no confusion.

We use a type system similar to that of the C programming language for these declarations. Tuple names are visible to the transaction and its sub-transactions. A tuple in T-Cham corresponds to a `struct` in the C programming language or a `record` in Pascal, for example,

```
tuples
          tuple {
                    int A[100];
                    int gridsieved;
          } num;
          boolean token;
          fifo char msg[256];
```

The keyword `tuple` is omitted if the tuple has only one field. The above declaration defines three tuple names: `num`, `token`, and `msg`. The tuple `num` has two fields; `msgs` are consumed in first-in-first-out (`fifo`) order.

The consumption order of a certain type of tuples is quite interesting, because those tuples may be distributed to and consumed by different computer nodes.

Theoretically, a tuple is randomly chosen for a reaction, and the choice is fair. It means that any available tuple will be *eventually* chosen. This kind of fairness is called *weak fairness* because the time when a particular tuple is chosen might be indefinitely far away [75]. Weak fairness works fine in a theoretical setting, where the time needed to get to a result is not the primary concern. However, in the most cases of real programs, results should be worked out in a limited time period. Take *Producer-Consumer* problem for example. If the Producer generates a message, say `msg1`, at a time while the Consumer does not consume the message immediately. After this time instance, every produced message will be consumed immediately. It can be claimed that the message `msg1` will be *eventually* consumed, provided time is unlimited, but nobody, although he/she is interested in this message but with limited life span, has the chance to see the message.

A first-in-first-out (`fifo`) or first-in-last-out (`filo`) ordering is used to enhance the fairness of a T-Cham program in the choices of tuples. As we know, a T-Cham program can be executed in a sequential, parallel, or distributed mode. In the sequential case, tuples have a linear time order. There can be an absolutely `fifo` or `filo` ordering. But in a parallel or distributed execution, T-Cham `fifo` ordering is only committed to the tuples coming from the same node; so is `filo`. Because in this two cases, a tuple can be generated by any execution node at any time instance. For example, suppose we have two execution nodes `PE1` and `PE2`. `PE1` generates a tuple `msg1` at its local time `t1` (local clock time), and `PE2` generates a tuple `msg2` at its local time `t2`. We call `t1` and `t2` *time-stamps*. If `t1` and `t2` come from the same computer, they can be used to indicate the age of `msg1` and `msg2`. But they are from two different execution nodes `PE1` and `PE2`. Each of those two execution nodes uses its own clock, i.e., the two execution nodes are on different *time systems*. It is very hard, even impossible to some extent, to compare `t1` and `t2`. The clocks of both `PE1` and `PE2` may be able to be synchronized against a real time clock, but the synchronization is not 100% precise [108].

In summary, an absolute linear time order cannot be pursued in the parallel situation. Although *time-stamps* can give some hints on the time when a tuple is generated, they only make sense for the tuples generated by the same computational node, for they are on the same time system.

4.3.2 Initialization

The initialization section sets up the initial state of a tuple space.

$$\begin{array}{lcl} \textit{Init} & \rightarrow & \texttt{initialization}\ \textit{InitList} \\ \textit{InitList} & \rightarrow & \{\textit{ActvInit} \mid \textit{PassInit} \mid \textit{MappingInit}\} \end{array}$$

An initialization action could be *passive* (assigning values to tuples) or *active* (calling one or more leaf-transactions, where there could be some input operations to get data from an input device or a file), for example,

```
        initialization
                [i:0..9]::token={i*2};
                init_num;
```

The initialization of `token` is passive, while the call to `init_num()`, which is a leaf-transaction, to initialize tuple `num` is active. "`[i:0..9]::token={i*2}`" means that for every `i` from 0 to 9, `token={i*2}`, i.e., there are ten `tokens` in the initial tuple space and their values are even numbers from 0 to 18 respectively. The `i` is called an *index variable.*

A third initialization method (*MapInit*) is the mapping of some tuples to the tuples in the tuple space of its parent transaction or of one of its children's. See Section 8.2 for details.

4.3.3 Reactionrules

The reactionrule section consists of a number of reaction rules. The rules operate on the tuple space of a transaction and coordinate its computational actions—transactions.

React	$\rightarrow$	`reactionrules` {*ReactRule*}+
ReactRule	$\rightarrow$	*LHS* `leadsto` *RHS* [`by` *Trans*] [`when` *ReactBExp*] ;
LHS	$\rightarrow$	*SimpleTupleList*
RHS	$\rightarrow$	*SimpleTupleList*
Trans	$\rightarrow$	*SimpleTransName* | *OnLineTrans*
SimpleTupleList	$\rightarrow$	*Tuple*{, *Tuple*}
Tuple	$\rightarrow$	*PlainIdentity*

Informally, a reaction rule looks like

$$x_1, x_2, \cdots, x_n \texttt{ leadsto } y_1, y_2, \cdots, y_m \texttt{ by } T \texttt{ when } f(x_1, x_2, \cdots, x_p),$$

where x_1, x_2, $\cdots$, x_n (i.e., *LHS*), y_1, y_2, $\cdots$, y_m (i.e., *RHS*) are tuples whose types are declared in the `tuples` section, T (i.e., *Trans*) is the name of a transaction (known as a sub-transaction to this transaction), and $f(x_1, x_2, \cdots, x_p), p \leq n$—i.e., *Cond*—is a boolean expression. The rule means that whenever the tuples x_1, x_2, $\cdots$, and x_n are all currently in the tuple space and the function $f(x_1, x_2, \cdots x_p)$ evaluates to `TRUE`, (i) the tuples x_1, x_2, $\cdots$, and x_n are selected and consumed, (ii) the transaction T is executed, and (iii) new tuples y_1, y_2, $\cdots$, and y_m are generated and injected back into the tuple space. From the point of view of the transaction which contains the reaction rule, these three actions are indivisible. Both `by` and `when` qualifiers of a reaction rule can be omitted if the transaction used is null and/or the condition is trivially `TRUE`.

There may be some common tuples among $x_1, x_2, \cdots, x_n, y_1, y_2, \cdots,$ and y_m. This means that more than one tuple of a certain type is needed for the reaction and/or some selected tuples are sent back to the tuple space (with or without changes), for example, "`x,x leadsto x,y`." To distinguish the different appearances of tuples in the body of a sub-transaction and the `when` condition part, the "`$`" operator is used with a constant integer index (called *instance reference*), e.g., "`when (x$1==x$2-10)`."

A pair of curly-brackets on a tuple name, say `{x}`, means all tuples of this type together, i.e., selecting them all, and a pair of |'s (vertical bars), `|x|`, means the number of this kind of tuples currently in the tuple space. Furthermore, a transaction does not have to consume all the tuples on the left-hand-side of its reaction rule. We use "`!`" to denote that the tuple is just read by the rule but not consumed, i.e., it is still available in the tuple space.

4.3.4 Termination

The termination section gives conditions such that whenever any of them is satisfied, the corresponding final action is committed and the transaction then terminates.

Term	$\rightarrow$	`termininination` { *TermStmt* }+
TermStmt	$\rightarrow$	`on` (*ReactBExp*) `do` (*Trans* | *AssembleData*);

A termination specification looks like this:

```
termination
          on (|token|==0) do output;
```

A transaction automatically enters termination status when there is no more reaction available, no matter whether the transaction has `termination` section or not. For an interactive program, which may never terminate, there is no `termination` section. It always has reactions.

4.3.5 Sub-transactions

The sub-transaction section specifies the pre-conditions and the post-conditions of the sub-transactions referred to by the reaction rules defined in the `reactionrules` section of a non-leaf transaction.

SubTrans	$\rightarrow$	`subtransactions` {*TransStmt*}+
TransStmt	$\rightarrow$	*Trans* : *PreCond* // *PostCond* ;
PreCond	$\rightarrow$	*BExp*
PostCond	$\rightarrow$	*BExp*

For example,

```
        subtransactions
                prod: |token|>0 // |token|'=|token|-1;
```

where `prod` is the name of the transaction referred to by a reaction rule, "`|token|>0`" is the pre-condition of the transaction, and "`|token|'=|token|-1`" the post-condition. The ' postfix operator means the values after the execution of the transaction.

4.4 Leaf Transactions

A leaf transaction contains no reaction rules and subtransactions. It is merely a group of operations.

LeafTrans	$\rightarrow$	`transaction` *LName* *LBody* `endtrans`
LName	$\rightarrow$	*PlainIdentity*
LBody	$\rightarrow$	*Micros* *BodyCode*
Micros	$\rightarrow$	`#language` *LangName* [`#tuplein` *SimpleTupleList*]
		[`#tupleout` *SimpleTupleList*]

where *LName* is the name of this leaf transaction. It abides by the same rule as that of a non-leaf transaction name. *LBody* is the body of this leaf transaction, which consists of a *Micros* part and a *BodyCode* part. The *Micros* part provides the necessary information concerning the resources passed to and the language used to write this leaf transaction to the *BodyCode* part, which is written in the programming language declared in the *Micros* part and carried out the operations of this transaction.

A leaf transaction looks like this:

```
transaction my_name
#language my_language
#tuplein tuple_desp
#tupleout tuple_desp
        my_body
endtrans
```

where *my_language*, known as a *guest language*, is the programming language used to code this transaction, *tuple_desp* provides the type information of the tuples to the transaction, and *my_body* is the programming units written in "*my_language*". There could be none or many "`#tuplein`" and "`#tupleout`" lines. The tuples described in "`#tuplein`" line are resources passed to this transaction before its execution and consumed by it during its execution. They are *not* parameters or arguments in the sense of being passed by *call-by-value*, *call-by-reference*, and/or *call-by-name* mechanism as in an ordinary programming language. T-Cham tuples are resources and can only be consumed and generated but not copied. Similarly, the tuples described in "`#tupleout`" line are those generated by the transaction. They are not the value returned to the "caller" but new resources injected back to the tuple space. The "`#tuplein`" and "`#tupleout`" lines serve as the interface between a conventional programming language "*my_language*" and T-Cham.

4.5 A Small Example

Example 1 (Element Summation) *Figure 4.1 is a T-Cham program which calculates the summation of all tuples in a tuple space of integers.* ∎

```
transaction root                                      -- the main transaction
        tuples
                int m;                                -- tuple declaration
        initialization
                m=10, m=20, m=30, m=50, m=15;         -- there are five tuples in
                                                      -- the initial tuple space
        reactionrules
                m, m leadsto m by sum2up;             -- the only reaction rule
        termination
                on (|m|==1) do out_sum;               -- when only one tuple left,
                                                      -- output the result
        subtransaction
                sum2up: p//q;                         -- pre and post conditions
endtrans

transaction sum2up                                    -- a leaf transaction
#language C                                           -- the guest language
#tuplein int m$1, m$2;                                -- the input resource
#tupleout int m$3;                                    -- the generated resource
        sum2up() {                                    -- the real function body
                m$3 = m$1+m$2;
        }
endtrans

transaction out_sum                                   -- another leaf transaction
#language C
#tuplein int m;
        out_sum() {
                printf("The summation = %d\n", m);
        }
endtrans
```

Figure 4.1: The T-Cham Program of Element Summation

The T-Cham program has three transactions: a main transaction `root` and two leaf transactions, `sum2up` and `out_sum`. The tuple section of the root transaction declares only one kind of tuples named `m`. Each of the tuples has only one field, which is an integer.

The initialization section of the program sets up the initial state of the tuple space. In this example, there are five tuples in the initial tuple space. They are `10`, `20`, `30`, `50`, and `15`.

The only reaction rule says that any two tuples in the tuple space can be replaced by a new tuple which is the sum of the two. The reaction for the replacement is carried out by the leaf transaction `sum2up`. At the first step, there could be up to $\lfloor \frac{n}{2} \rfloor$ reactions depending on the available computer resources, where n is the number of tuples in the initial tuple space, and then $\lfloor \frac{n}{2^2} \rfloor$, $\lfloor \frac{n}{2^3} \rfloor$, $\cdots$. When there is only one tuple left in the tuple space, we output the result according to the termination condition section of the transaction.

The effect of `sum2up` upon the tuple space can be found in the subtransaction description section of this root transaction. If the values of the two input tuples to the transaction `sum2up` are x and y respectively, the value of result tuple generated by the transaction should be $x + y$. Thus, the pre- and post-condition of `sum2up` are:

$$\texttt{p} \equiv \texttt{true}, \qquad \texttt{q} \equiv \texttt{m\$3} = m\$1 + m\$2$$

The post-condition here syntactically resembles the assignment statement of the transaction `sum2up`, but it carries different meaning. It is a logic assertion, which can be either `true` or `false`. A nice property of subtransaction description is that subtransaction behavior can be figured out without knowing the details of its code, and also, the pre- and post-conditions can be used to build a constructive proof system for the program verification. The subtransaction section serves as the interfaces between transactions so that each of them can be treated in isolation.

4.6 The Execution of a T-Cham Transaction

The execution of a T-Cham transaction proceeds as follows: until a termination condition is satisfied, all of its reaction rules are *fairly* chosen and tested. Whenever

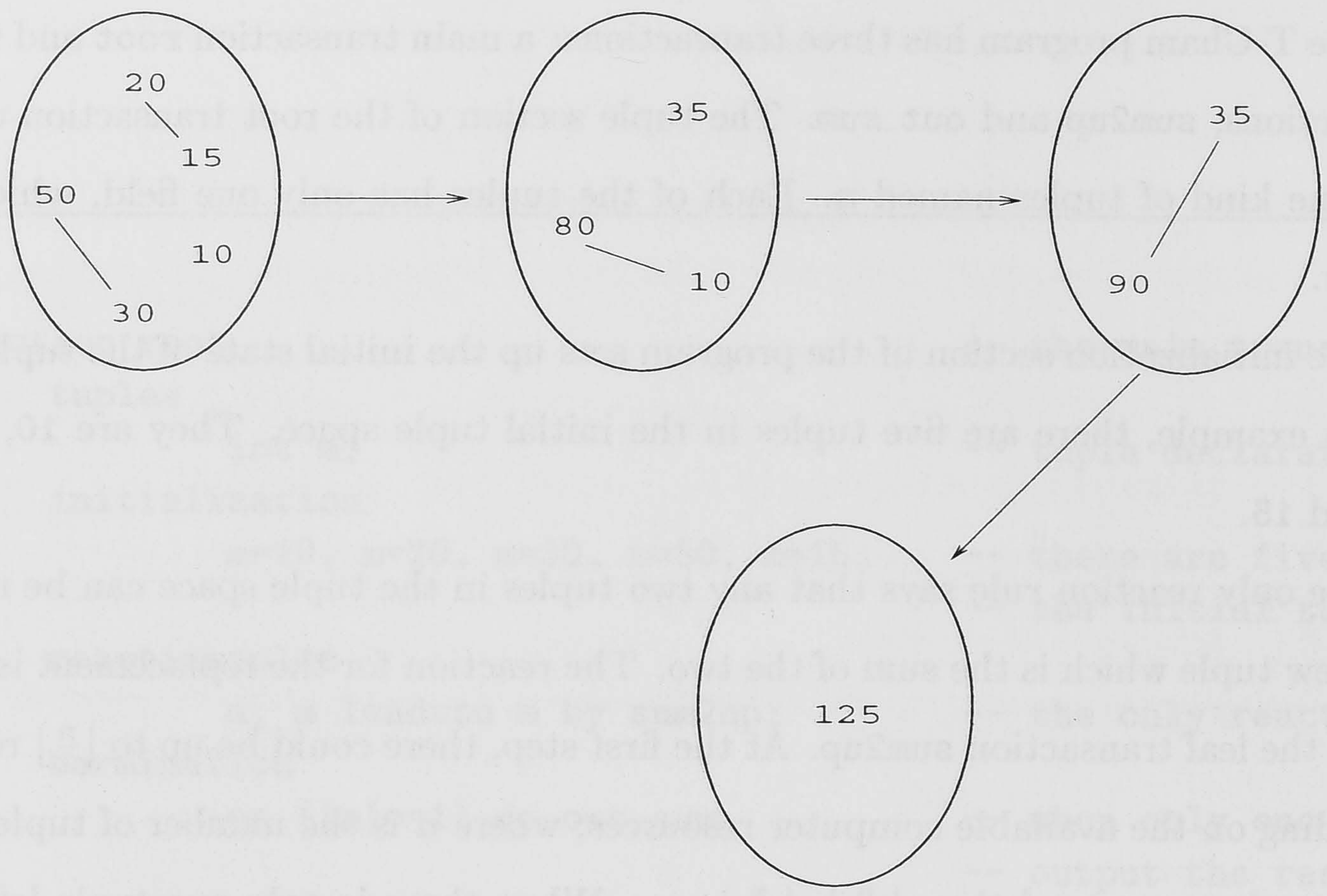


Figure 4.2: A Possible Computation Process of Element Summation

the *reaction condition* of a reaction rule holds, i.e., the tuples needed by the reaction rule are all currently in the tuple space and the boolean function of its `when` qualifier—if it exists—evaluates to `TRUE`, the corresponding sub-transaction (the `by` part of a reaction rule) is eligible to be invoked. By *fairly*, we mean that a reaction will eventually happen if its reaction condition is continuously satisfied. The pictorial description of a possible execution path of the Figure 4.1 program is in Figure 4.2, where arrows indicate the tuple space state changes.

If a sub-transaction to be executed is written in T-Cham, a new tuple space is established according to the specification of this sub-transaction. The relations between the tuples of the two level tuple spaces are also established. The simplest relationship is to project a subset of the tuples from a parent transaction to its sub-transaction. More complex mappings are described in Chapter 8. All the actions of the sub-transaction operate on the new tuple place, which will be destroyed after the execution. The tuple space of a T-Cham transaction corresponds to the run time environment of a function or a procedure in an imperative programming language. From the point of view of a transaction, each of its sub-transactions is an "operator" and is executed atomically.

4.7 Summary

The chapter discussed the basic T-Cham notations and a small example. The basic idea is that a T-Cham program consists of a number of transactions. Each of them is an autonomous operation unit and imitates an isolated chemical reaction system. Transactions are categorized into two classes: leaf transactions and non-leaf transactions. A leaf transaction is written in a language other than T-Cham and does not spawn any sub-transactions in the sense of T-Cham. A non-leaf transaction is written in T-Cham itself.

A transaction is an autonomous operation unit, but it does communicate with others. A large T-Cham system consists of a number of transactions. As those transactions are from the same system, each of them may be designed to solve part of the same problem. They cooperate with each other to solve the problem and achieve the ultimate goal. A transaction communicates with the others via its interfaces, which are specified as its initialization section, termination section, and subtransaction section. The initialization section sets the initial tuple space status of a transaction according to the current tuple space configuration of its parent transaction. The termination section brings the result (return value in terms of an imperative programming language) of a transaction to its parent transaction. Finally, the subtransaction section describes the behaviors and attributes of sub-transactions.

Chapter 5

Programming in T-Cham

In this chapter, we use some examples to illustrate the T-Cham programming style. Different programming languages have different programming styles. For example, if we are asked to calculate the summation of n numbers N_1, N_2, $\cdots$, and N_n in an imperative programming language, the n numbers may be assigned to an array, and a loop which repeats n times accumulates the result by adding from N_1 to N_n, but in T-Cham, the result is achieved by simulating the process of a chemical reaction system: the n numbers are represented by n tuples in a tuple space, and those tuples react to each other—any two tuples can be transformed to a new tuple with the value of their summation—like molecules in a chemical reaction system.

This chapter mainly concentrates on the programming style of the T-Cham programming language. Some of the examples used in this chapter are computation oriented programs, including *the sieve of Eratosthenes*, *the Dutch flag*, *the Fibonacci numbers*, and *the calculation of the value of* π problems, and the others are interactive ones, such as *vending machine*, *producer-consumer*, *sleeping barber*, and *meeting scheduler* problems. In this chapter, we also discuss the effect of non-determinism on programming and program execution.

All the leaf transactions of the examples in this chapter are written in the C programming language except the calculation of the value of π (Section 5.8), whose leaf transactions are written in Java [133] to show the ability of T-Cham to adopt different programming languages.

5.1 The Sieve of Eratosthenes

Example 2 (Eratosthenes Sieve) Eratosthenes Sieve *is one of the oldest methods to find prime numbers. The basic idea is very simple: we put all (in a particular program, some) the natural numbers in a sieve. In the first round, all the multipliers of the number* 2 *are sieved out, and then the multipliers of* 3, 5, $\cdots$. *Finally, only the prime numbers left. The program of Figure 5.1 finds all prime numbers in a segment of natural numbers beginning with the smallest prime number* 2. ∎

```
transaction root

        tuples
                int n;

        initialization
                [i:2..500000]::n=i;

        reactionrules
                n, n leadsto n$2 when (n$1 mod n$2 == 0);

endtrans
```

Figure 5.1: The T-Cham program of Eratosthenes Sieve

The tuple space in this example only has one type of tuples. Each individual tuple has one field of integer. Initially, the tuple space has $(500000 - 1)$ tuples which represent the natural numbers from 2 to $500,000$ according to the initialization section. The reaction rule says that every number (tuple) destroys its multipliers; thus, only prime numbers are left until no reaction rules can be applied.

We have prototype implementation experience (on CM-5) of this example [126].

```
#define R      500000      -- the total number to be sieved
#define M      20000       -- the size of data chunks

transaction root

        tuples
                tuple {                         -- data chunks: 2-M, M+1-2*M ...
                    int A[20000];
                    int gridsieved;  -- sieved by this number
                } num;
                int N;                          -- the number of chunks
                int grid, nextgrid;
                boolean done;

        initialization
                grid=2; nextgrid=3; N=R/M;
                init_num();

        reactionrules
                num, !grid leadsto num, done, nextgrid by sieve
                     when (num$1.gridsieved != grid);
                {done}, nextgrid, grid leadsto grid by {grid$2 = nextgrid;}
                     when (|done|==N);

        termination
                on (|nextgrid|==0 && |done|==N) do out_prime;

        subtransactions
                sieve: |done|==k // i:1-M:: num$2.A[i] = (num$1.A[i]%grid) ?
                                          num$1.A[i]:0 and |done|==k+1

endtrans
```

Figure 5.2: Another T-Cham program of Eratosthenes Sieve

Our experience reveals that although the T-Cham program is very concise and straightforward, its implementation is not so efficient. We observed that sometimes, though implemented on a 64 node CM-5 parallel computer, it is even slower than a sequential program written in the C programming language running on a SUN station. The culprit here is the so-called *non-determinism*. The reaction rule in this example is very simple: *every number destroys its multipliers*. For example, the number 2 destroys the numbers of 4, 6, 8, $\cdots$, and the number 3 destroys the numbers of 6, 9, 12, $\cdots$ etc. The idea is clear, but the order of the process is not given. In a completely chaotic situation, the possibility exists that when the number 2 is destroying the number 4, the number 6 is trying to destroy 12, and 9 is trying 33, while the number 25 just failed to destroy 5. This experience suggests that while the non-determinism is good for algorithm expression, determinism is good for efficiency.

A more efficient and also more difficult to understand program is in Figure 5.2, where the sequence of the natural numbers from 2 to 500,000 is broken down into subsequences (called chunks), e.g., 2–20000, 20001–40000, and so on. We sieve the chunks with `grid` concurrently (reaction rule 1). After every chunk has been sieved by `grid`, which is indicated by the number of `done` tuples in the tuple space, `grid` is replaced by one of `nextgrid` tuples and all `done` tuples vanish at the same time (reaction rule 2), and then the next round of sieving. The program will be easier to understand if you come back after the subsequent sections.

As T-Cham relies on the transactions written in imperative programming languages for its efficiency, automatic program transformation is not our primary concern in this thesis, although it can further increase the efficiency of T-Cham implementation. We refer the interested readers to [84, 48, 44, 85, 169].

5.2 Vending Machine

Example 3 (Vending Machine) *There is a vending machine which sells chocolate bars: a large bar costs \$2, and a small one costs \$1; furthermore, we assume that only these two kinds of coins*[1] *can be used. Two buttons on the vending machine are* `large`

[1]We have \$1 and \$2 coins in Australia.

and `small`*, which are used by a customer to choose a chocolate bar. The program is in Figure 5.3.* ∎

```
transaction root

        tuples
                boolean ausd1, ausd2;           -- $1 and $2 coins
                boolean small, large;           -- the two buttons
                boolean LargeBar, SmallBar;     -- the two kinds of choc bars

        initialization
                [1..M]::LargeBar=TRUE;          -- M large bars
                [1..N]::SmallBar=TRUE;          -- N small bars
                [1..S]::ausd2=TRUE;             -- S number of $2
                [1..T]::ausd1=TRUE;             -- T number of $1
                small=TRUE, large=TURE;         -- the two buttons

        reactionrules
                ausd2, large leadsto LargeBar, large;
                ausd2, small leadsto SmallBar, ausd1, small;
                ausd1, ausd1 leadsto ausd2;
                ausd1, small leadsto SmallBar, small;

endtrans
```

Figure 5.3: The T-Cham Program of Vending Machine

The tuple space of the problem simulates the vending machine. It has `M` large chocolate bars, `N` small ones, and two buttons (`small` and `large`) at the beginning. Coins and the buttons pressed are the stimuli from the outside world. The vending machine responds to the stimuli according to the reaction rules: a \$2 coin with a button `large` pressed gives out a large chocolate bar (reaction rule one), or a small chocolate bar and a \$1 coin change if the button `small` is pressed (reaction rule two). By reaction rule three, two \$1 coins makes one \$2 value, and by rule four, a \$1 coin and pressing button `small` produce a small chocolate bar. In the reaction rule 1, we can see that the tuple `large`, which stands for the button for a large chocolate bar, appears on both side of the reaction rules. It means the tuple is a re-usable resource, and it can be recycled after "being used".

We use the "`!`" operator before the tuples of `small` and `large` because they can never be consumed.

To manage the re-charge of chocolate bars, we can use termination section:

```
termination
    on (|LargeBar|==0 || |SmallBar|==0) do wait_for_recharging;
```

As the transaction `wait_for_recharging` is in the termination section, a re-start of the program is necessary. The arrangement is fine with this particular application. In real world, we do see vending machines being shut down for re-charging. For some other applications, which may have to keep running all the time, we cannot rely on termination section to handle exceptional events. Exception handlers should be introduced into T-Cham. As it is not the core components of T-Cham, we leave it as future work for the time being.

5.3 The Producer-Consumer Problem

Example 4 (The Producer-Consumer Problem, with bounded buffer) *The producer produces one message, a string of at most MAX characters, at one time and the consumer consumes one message at another time. Both producer and consumer are autonomous. The only constraint on them is the capacity of the repository where the messages are temporarily stored. We assume the capacity is N in our example. The producer will continue producing messages as long as the total message number is less then N, and the consumer will consume messages whenever they are available. The T-Cham program is given in Figure 5.4.* ■

The first two lines define two constant `MAX` and `N`. They are substituted by `1024` and `100` respectively before the program is passed to a T-Cham compiler.

There are two kinds of tuples in this program. They are `token` and `msg`. The number of tuple `tokens` denotes the current capacity of the message container in this example. If there are n $(0 < n \leq \texttt{N})$ `tokens` currently in the tuple space, it means that

```
#define MAX      1024         -- the max length of a message
#define N        100          -- the max number of messages

transaction root

        tuples
                boolean token;       -- a place-holder for a message
                char msg[MAX];       -- a message

        initialization
                [i:1..N]::token=1;

        reactionrules
                token leadsto msg by prod;       -- producer's rule
                msg leadsto token by cons;       -- consumer's rule

        subtransaction
                prod: |token|>0 // (|token|'=|token|-1)&&(|msg|'=|msg|+1);
                cons: |msg|>0 // (|token|'=|token|+1)&&(|msg|'=|msg|-1);

endtrans

transaction prod
#language C
#tuplein boolean token;
#tupleout char msg[];

        prod() {
                /* some C code writing messages to the msg tuple */
        }

endtrans

transaction cons
#language C
#tuplein char msg[];
#tupleout boolean token;

        cons() {
                /* some C code reading the content of the msg tuple */
        }

endtrans
```

Figure 5.4: The T-Cham Program of Producer-Consumer Problem

the producer still can produce n messages without any consumption by the consumer. The tuple `msgs` simply hold the messages. At the very beginning, there are `N` tuple `tokens` and no `msg`.

The first reaction rule says that the leaf transaction `prod` consumes (thinking as occupies) one `token` tuple and generates one message `msg`. The reaction cannot happen if there are no `tokens` left. The second reaction rule says that the leaf transaction `cons` consumes one message, and as the result of the consumption, one more `token` is available. Similarly, the consumption reaction could not happen if there is no message currently in the tuple space.

The subtransaction description section gives the pre- and post-conditions of the two leaf transactions: `prod` and `cons`. They actually specify the population of `token` and `msg` in the tuple space before and after the execution of the leaf transactions.

There is no termination section in this example because this is a non-terminating program.

We do not give the details of the C codes of the two leaf transactions. They are not relevant to T-Cham.

Finally, in this example, we do not care about the order of the consumed messages. That any message will be *eventually* consumed is guaranteed by the fairness principle of T-Cham.

5.4 The Dutch Flag Problem

Example 5 (Dutch Flag) *We have an array of n elements, each of which is either* `Red`, `White`, *or* `Blue`. *A program is needed to re-arrange their positions so that all* `Red` *elements come before* `White` *ones, which are in turn before* `Blue` *ones. The n elements are represented by n tuples in the tuple space. Every tuple has the form of (x, y), where x is the sequence number of the element in the original array and y is the color of the element. The T-Cham program is in Figure 5.5.* ■

The idea of the program is very simple. Pick up any two tuples of two different colours. If their relative position indexes are not right, exchange the two indexes, and then put the two new tuples back to the tuple space. Repeat the procedures until no

```
transaction root

    tuples
        tuple {
            int order;
            enum color {r,w,b};
        } strip;                     -- one element

    initialization
        init_element;

    reactionrules
        strip(i,r), strip(j,w) leadsto strip(i,w), strip(j,r)
                               when (strip.i > strip.j);
        strip(i,w), strip(j,b) leadsto strip(i,b), strip(j,w)
                               when (strip.i > strip.j);
        strip(i,r), strip(j,b) leadsto strip(i,b), strip(j,r)
                               when (strip.i > strip.j);

    termination
        on(forall (i,r), (j,w), (k,b) :: i<j<k) do output_strip;

endtrans

transaction init_element
#language C
    init_element() {
        /* put n strip tuples into the tuple space */
    }
endtrans

transaction output_strip
#language C
    output_strip() {
        /* code for strip tuples output */
    }
endstran
```

Figure 5.5: The T-Cham Program of Dutch Flag

such tuples exist.

In this example, we use the internal structures (with some values) of tuples as part of the selection criteria to choose tuples for reactions. The first reaction rule of the program in Figure 5.5 is equivalent to:

```
    strip, strip leadsto strip, strip by index_exchange
                    when (strip$1.color=='r' && strip$2.color=='w' &&
                              strip$1.i > strip$2.j);
```

where the transaction `index_exchange` exchanges the index value of `strip$1` and `strip$2`.

5.5 Sleeping Barber

The Sleeping Barber problem is actually an abstraction of the client/server programming model, which is widely used in programming for computer networks.

Example 6 (Sleeping Barber) [2] *A barber provides hair-cutting service in his shop, where there are two doors—one for entrance and the other for exit—and N chairs for waiting customers. Only one customer can receive the service on the barber's chair at a time. When there are no customers in the shop, the barber will fall asleep on his chair; otherwise, he continuously provides hair-cutting service until no customers are left. The barber spends all his life serving customers or sleeping.*

When a customer arrives and finds the barber sleeping, he wakes up the barber and has his hair cut on the barber's chair. After the service, the customer gets out of the shop by the exit door. If the barber is busy when a customer comes, the customer will take a seat, provided that there is an empty chair, and wait for the barber. If all chairs are occupied, the new customer has to wait until a chair is available. The T-Cham program is in Figure 5.6. ■

The tuple space of this program simulates the barber's shop. The tuples in the tuple space denote the states of each customer, each chair, and the barber. A `pin` in

[2] We simply assume that the barber and all his customers are male for description brevity.

```
transaction root

        tuples
                boolean pin, pwt, pcut, pout;    -- the states of
                                                 -- a customer
                boolean bsp, bwk, bfin;        -- the states of the barber
                boolean chair;

        initialization
                [i:1..N]::chair=TRUE;          -- There are N chairs.
                bsp=TRUE;                      -- The barber is asleep.

        reactionrules
                pin, bsp leadsto pcut, bwk;
                pin, chair leadsto pwt when (!bsp);
                pcut, bwk leadsto pout, bfin by cutting;
                pwt, bfin leadsto pcut, chair, bwk;
                bfin leadsto bsp when (|pwt|==0);

        subtransaction
                cutting: pcut&&bwk // pout&&bfin;

endtrans

transaction cutting
        /* some operations here */
endtrans
```

Figure 5.6: The T-Cham Program of Sleeping Barber

the tuple space means that a new customer is coming, `pwt` a customer is waiting on a chair, `pcut` a customer is sitting in the barber's chair and having his hair cut, and `pout` means a customer leaving the barber's shop. `bsp` denotes that the barber is sleeping, `bwk` the barber is working, and `bfin` the barber has just finished cutting the hair of a customer. A tuple `chair` in the tuple space means that the chair is available for a new customer.

Initially, there are N chairs available in the tuple space (i.e., barber's shop) and the barber is sleeping.

The first reaction rule says that if a customer finds the barber is sleeping when he is coming, he wakes the barber up and has his hair cut, i.e., makes the barber busy, and the second, that if a new customer finds that the barber is busy (or not sleeping) and a chair is available, he will sit down on the chair and wait for the barber. By the reaction rule three, the busy barber will finish his service to the customer who is having his hair cut so that the customer is ready to go. A waiting customer is asked to sit down on the barber's chair to have his hair cut according to reaction rule four; as the consequence, an occupied chair is available again. By the reaction rule five, when there are no waiting customers, the barber is going to sleep on his chair.

This is a typical interactive situation. Unlike the Eratosthenes Sieve example, there is no way to eliminate non-determinism. Other solutions of the problem can be found in [17, pp 266–267] and [11, pp 290–294]. We believe readers will find the program given here has a more natural and intuitive presentation.

5.6 The Meeting Scheduler

Example 7 (Meeting Scheduler) *We are to find the earliest common meeting time for a group of people. For the sake of brevity, we suppose that there are only three people, F, G, and H, in the group. Every person of the group suggests his/her earliest acceptable meeting time. Finally, the earliest common meeting time is reached. Figure 5.7 shows a T-Cham program to solve the problem*[3]. ■

[3]*For a more detailed discussion of the problem and other solutions, we refer readers to Chandy and Misra [43, pp 13–18]*

In the program, the tuple time holds the current suggested time for the meeting. It will be changed by the persons F, G, and H according to their own agenda. The transaction F_time (resp. G_time and H_time) withdraws time and then checks it against to his/her own agenda. If the time is an acceptable time, it remains unchanged and F_changed is set to FALSE; otherwise, a new time is put back into the tuple space and F_changed is set to TRUE. The common meeting time will be reached when F_changed, G_changed, and H_changed are all set to FALSE.

More formally, the transaction F_time executes the function f:

$$f: \ int \rightarrow int.$$

The result of $f(t)$ is the time acceptable for F to have the meeting according to the current suggestion t, i.e., for any t, $f(t) \geq t$, and $f(t)$ is an acceptable time for F while any other time r, $t < r < f(t)$, is not acceptable. g and h are defined accordingly.

In fact, we can have a unique transaction my_time instead of F_time, G_time, and H_time. The transaction executes the function ϕ:

$$\phi : \{F, G, H\} \times int \rightarrow int, \qquad \text{or} \qquad \phi : \{F, G, H\} \rightarrow (int \rightarrow int).$$

If we apply ϕ to F, G, and H, we obtain $\phi(F) \equiv f$, $\phi(G) \equiv g$, and $\phi(H) \equiv h$. We keep the different transactions, F_time, G_time, and H_time, in the thesis to simplify our proof in Chapter 7. The result of program in Figure 5.7 is the time u which satisfies $u = f(u) = g(u) = h(u)$.

5.7 The Fibonacci Numbers

Fibonacci numbers are not a commonly used example for concurrent programming, especially in parallel situation, for the definition of the n^{th} Fibonacci number,

$$f(n) = \begin{cases} f(n-1) + f(n-2) & \text{if } n > 2 \\ 1 & \text{if } n = 1 \text{ or } n = 2, \end{cases}$$

suggests a very limited degree of concurrency, but an alternative definition of the formula exposes a high degree of concurrency.

```
transaction root

  tuples
     date time;                        -- the type of time is date.
     boolean F_changed, G_changed, H_changed;

  initialization
     time=0;
     F_changed=G_changed=H_changed= true;

  reactionrules
     time, F_changed, !G_changed, !H_changed leadsto time, F_changed
        by F_time when (G_changed==TRUE || H_changed==TRUE);
     time, G_changed, !F_changed, !H_changed leadsto time, G_changed
        by G_time when (F_changed==TRUE || H_changed==TRUE);
     time, H_changed, !F_changed, !G_changed leadsto time, H_changed
        by H_time when (F_changed==TRUE || G_changed==TRUE);

  termination
     on(F_changed==false && G_changed==false && H_changed==false)
        do output;

  subtransactions
     F_time: time=r // (time=r) && (F_changed==FALSE) ||
                      (time=f(r)) && (F_changed==TRUE)
     G_time: time=r // (time=r) && (G_changed==FALSE) ||
                      (time=g(r)) && (G_changed==TRUE)
     H_time: time=r // (time=r) && (H_changed==FALSE) ||
                      (time=h(r)) && (H_changed==TRUE)
endtrans

transaction F_time
#language C
#tuplein date time;
#tupleout boolean F_changed, G_changed, H_changed;
     F_time() {
           /* the C code to figure out the earliest available time */
           /* for the person F from his/her event calendar.        */
     }
endtrans

/* the leaf transactions of G_time and H_time, similar to that of F_time */
```

Figure 5.7: The T-Cham Program of Meeting Scheduler

Example 8 (Fibonacci Number) *The n^{th} and the $(n-1)^{th}$ Fibonacci numbers can be defined by the $(n-1)^{th}$ and the $(n-2)^{th}$ numbers, and so on:*

$$\begin{aligned}\begin{pmatrix} f(n) \\ f(n-1) \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} f(n-1) \\ f(n-2) \end{pmatrix} = \cdots \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}\begin{pmatrix} 1 \\ 0 \end{pmatrix}.\end{aligned}$$

Let `coef` *be* $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ *and* `fib` *be* $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, *then the n^{th} and $(n-1)^{th}$ Fibonacci numbers will be the matrix product of $(n-1)$* `coef`*s and one* `fib`*. The T-Cham program is in Figure 5.8.* ∎

The tuple `coef` and the tuple `fib` correspond to the two matrices. At first, i.e., the initial tuple space state of transaction `root`, there are $(n-1)$ copies of tuple `coefs` and one `fib`. The reaction rules of `root` say that two copies of `coefs` can be used to produce one `coef` by the transaction `mc`, which calculates the product of the two `coef` matrices, and one `coef` and one `fib` can be made to one new `fib`, the product of the two tuples (matrices) by `mf`. Whenever there is no `coef` left, the program can terminate and output the result — the n^{th} and the $(n-1)^{th}$ Fibonacci numbers, contained in the remaining `fib` tuple.

The leaf transactions `init_coef` and `mf` are written in the C programming language ("`#language C`" in the transactions). The former reads the number n from keyboard and then generates $(n-1)$ copies of `coef` tuples to the `root`'s tuple space; and the latter calculates the matrix product of a `coef` and a `fib`. The `coef` and the `fib` in the transaction `mf` are not taken as arguments. They are the resources prepared for the transaction before its execution and consumed by it after the execution. Similarly, the tuples generated by a transaction are not the function value returned but the new resources. Finally, "`#language`" specifies the language used to write the transaction (default is T-Cham itself) and the "`#tuplein`" and "`#tupleout`" line is used to provide the type information of a tuple to the compiler.

`p0`, `p1`, `p2`, `q0`, `q1`, and `q2` are the pre-conditions and post-conditions of the three transactions of `init_coef`, `mc`, and `mf`. They are

```
transaction root

        tuples
                tuple {int a,b,c,d;} coef;
                tuple {int x,y;} fib;
        initialization
                init_coef; fib={{1,1}};
        reactionrules
                coef, coef leadsto coef by mc;
                coef, fib leadsto fib by mf;
        termination
                on (|coef|==0) do output_fib;
        subtransactions
                init_coef: p0 // q0;
                mc: p1 // q1;
                mf: p2 // q2;

endtrans

transaction init_coef
#language  C
#tupleout tuple {int a, b, c, d} coef;
        void init_coef() {

                int i,n;

                scanf("%d", &n);
                for (i=1; i++; i<n) coef.a=coef.b=coef.c=1, coef.d=0;
        }
endtrans

transaction mf
#language   C
#tuplein tuple {int a, b, c, d} coef;
#tuplein tuple {int x, y} fib$1;
#tupleout tuple {int x, y} fib$2;
        void mf() {
                fib$2.x=coef.a*fib$1.x,coef.b*fib$1.y;
                fib$2.y=coef.c*fib$1.x,coef.d*fib$1.y;
        }
endtrans

        /* mc is omitted. it is similar to mf */
```

Figure 5.8: The T-Cham Program of Fibonacci Number

$$\begin{aligned}
&\texttt{p0} \equiv \texttt{TRUE},\\
&\texttt{p1} \equiv \texttt{coef\$1} = \begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} \wedge \texttt{coef\$2} = \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix},\\
&\texttt{p2} \equiv \texttt{coef} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \wedge \texttt{fib\$1} = \begin{pmatrix} x \\ y \end{pmatrix},\\
&\texttt{q0} \equiv |\texttt{coef}| = n \wedge \forall \texttt{coef} \in \mathcal{T} : \texttt{coef} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\\
&\quad \wedge |\texttt{fib}| = 1 \wedge \texttt{fib} = \begin{pmatrix} 1 \\ 0 \end{pmatrix},\\
&\texttt{q1} \equiv \texttt{coef\$3} = \begin{pmatrix} a1a2 + b1c2 & a1b2 + b1d2 \\ c1a2 + d1c2 & c1b2 + d1d2 \end{pmatrix},\\
&\texttt{q2} \equiv \texttt{fib\$2} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix},
\end{aligned}$$

where $\mathcal{T}$ denotes the tuple space. These pre-conditions and post-conditions are used for program verification purposes. If a programmer finds it is difficult to provide them for every transaction or is unwilling to do so, just simply have the conditions be trivially `true`.

5.8 The Calculation of the Value of π

Example 9 (The Calculation of the Value of π) *The value of π can be obtained by numerical integration method:*

$$\begin{aligned}
\pi &\approx 4\int_0^1 \frac{1}{1+x^2}dx\\
&= \sum_{i=0}^{n-1} 4\int_{\frac{i}{n}}^{\frac{i+1}{n}} \frac{1}{1+x^2}dx, \quad n \in \mathcal{N},
\end{aligned}$$

where $\mathcal{N}$ is the set of natural numbers. For each integration $\int_p^q f(x)dx$, its numerical value is, approximately,

$$\int_p^q f(x)dx \approx \frac{1}{N}\sum_{i=0}^{N} f(p + \frac{(q-p)*i}{N}), \quad N \in \mathcal{N}.$$

The T-Cham program is in Figure 5.9. ∎

```
transaction root
        tuples
                tuple {double p, q;} dom;    -- a chunk of integration
                double val;                  -- the value of the pi
                int N;              -- density for the numerical integration
        initialization
                N=1000;
                init_dom();         -- active initialization for dom
        reactionrules
                dom, !N leadsto val by integration;
                val, val leadsto val by summation;
        termination
                on (|dom|==0 && |val|==1) do out_pi;
        subtransaction
                init_dom:p0//q0;
                integration:p1//q1;
                summation:p2//q2;
endtrans

transaction init_dom
        /* omitted to save space */
endtrans

transaction integration
#language Java
#tuplein {double p, q;} dom
#tuplein int N
#tupleout double val
        public class integration() {
                double pi, x, step;
                int i;

                step = (dom.q-dom.p)/N;
                x = dom.p;
                pi = 4*(1/(1+x*x));
                for (i=0; i<N; i++) {
                        x += step;
                        pi += 4*(1/(1+x*x));
                } //for
                val = pi/(N+1);
        }
endtrans

transaction summation
        /* omitted to save space */
endtrans
```

Figure 5.9: The T-Cham Program to Calculate the Value of π

This program is quite easy to understand. Initially, in the tuple space, there are certain chunks of the integration:

$$4 \times \int_{\frac{i}{n}}^{\frac{i+1}{n}} \frac{1}{1+x^2} dx,$$

which are put into the tuple space by the leaf transaction `init_dom`. The transaction `integration` turns them to a numerical value `val`. The value π is obtained by adding all those `val`s together.

There can be as many transaction `integration` concurrently as to the limit of the computational resources or the total number of the tuples `dom` left in the tuple space. The amalgamation of the tuple `val` can happen at the same time.

5.9 Discussion

The chapter concentrated on the using of the basic T-Cham programming language to write application programs.

To write a T-Cham program, or to solve a problem with the T-Cham programming language, we first divide the problem into small pieces of sub-problems, and then figure out the reactions among those sub-problems during the process of computation. We use tuples to store the data, including the initial data, intermittent results and the final results, and reaction rules to describe the reactions. Some of the examples in this chapter are interactive programs, which never terminate, while the others are computational intensive ones. All of them only have the main (i.e., the root) transaction and a number of leaf transactions. There is no nested T-Cham transaction. We leave the issue to Chapter 8.

In this chapter, we also discussed the impact of non-determinism on the program execution efficiency and the program itself specification. We believe that non-determinism is good for algorithm expression, but determinism is good for execution efficiency. The T-Cham implementation (in Chapter 6) has little ability to reduce non-determinism. It is necessary for programmers to eliminate any *unnecessary* non-determinism. We suggest that programmers start with the most natural way to write their programs. After the ideas and the logic of the programs have been proven, reduce the unnecessary non-determinism for the efficient execution of the programs.

This program is quite easy to understand. Initially, in the tuple space, there are certain chunks of the integration:

$$4 \times \int_p^q \frac{1}{1+x^2}\,dx$$

which are put into the tuple space by the leaf transaction init_dom. The transaction integration turns them to a numerical value val. The value of π is obtained by adding all those vals together.

There can be as many concurrent integration reactions as possible to the limit of the computational resources or the total number of the tuples dom left in the tuple space. The amalgamation of the tuple val can happen at the same time.

5.9 Discussion

The chapter concentrated on the using of the basic T-Chant programming language to write application programs.

To write a T-Chant program, or to solve a problem with the T-Chant programming language, we first divide the problem into small pieces of sub-problems, and then figure out the reactions among those sub-problems during the process of computation. We use tuples to store the data, including the initial data, intermediate results and the final results, and reaction rules to describe the reactions. Some of the examples in this chapter are interactive programs, which never terminate, while the others are computational intensive ones. All of them only have the main (i.e., the root) transaction and a number of leaf transactions. There is no nested T-Chant transaction. We leave the issue to Chapter 8.

In this chapter, we also discussed the impact of non-determinism on the program execution efficiency and the program itself specification. We believe that non-determinism is good for algorithm expression, but determinism is good for execution efficiency. The T-Chant implementation (in Chapter 6) has little ability to reduce non-determinism. It is necessary for programmers to eliminate any unnecessary non-determinism. We suggest that programmers start with the most natural way to write their programs. After the ideas and the logic of the programs have been proven, reduce the unnecessary non-determinism for the efficient execution of the program.

Chapter 6

T-Cham Implementation

In this chapter, we propose an implementation model for T-Cham programs. It is a generic model, called *T-Cham Machine*, for MIMD computer architectures. We tested this basic idea on the AP1000 multicomputer and received some preliminary performance data.

The chapter first introduces some fundamental issues involved in the implementation of a parallel programming language, and then proposes the T-Cham Machine. It is actually an extension to the master/worker parallel programming model. Because the native master/worker model is subject to communication congestion between the master node and the worker nodes when the number of worker nodes increases, the T-Cham Machine suggests a multi-master structure. Section 6.2 describes our basic methods of managing data integrity, task distribution, and the communication in multiple master environment. Section 6.3.1 gives some preliminary performance data of a T-Cham Machine implementation on the AP1000 multicomputer together with theoretical performance analysis, and Section 6.3.2 studies the performance of the Matrix Multiplication example (Example 12) on the T-Cham Machine. We conclude this chapter with a discussion on some broader issues involved in a full-fledged T-Cham implementation.

6.1 Introduction

The implementation of a parallel programming language is much more complex than that of a sequential programming language. For example, in addition to the transformation from the source code of a programming language to the machine code of a target computer, (i) *parallel task description and partitioning* (how to describe a parallel task, how large is the task, and how to divide and assemble parallel tasks?), (ii) *scheduling* (when and where a task should be started?), (iii) *communication and synchronization* among those tasks, and (iv) *scalability* (will the performance be better if more processor elements are involved?) are all new issues.

In this section, we briefly discuss the implementation issues which are directly related to our T-Cham Machine. For more detailed discussion about parallel programming implementation, we refer readers to other reports, papers, and books [171, 97, 46, 72, 103, 31, 142, 61, 157, 33, 162, 102].

Parallel task description and partitioning are not only the major issues of parallel programming language design but also the issue of its implementation. As we discussed in Section 2.2, there are two kinds, in terms of parallel task description, of parallel programming languages: implicit or explicit. An implicit parallel programming language has no concept of parallelism at the programming language level. Parallel tasks are identified by its implementation. In contrast, an explicit language requires programmers to provide the information of parallelism. When talking about implementation, it is obvious that the parallel tasks inside of the program written in an implicit programming language need to be identified, while for the program which is written in an explicit programming language, there is still some room for parallel task identification because the description of the original program may not be suitable to this particular implementation or this particular run-time environment etc. The size of a parallel task is defined as *granularity* (Section 3.1.4). Ideally, a good implementation has the ability to automatically change the granularity, known as *granularity packing*, of parallel tasks for the best performance. Of course, the more information provided by the source code of a program, the better and the easier the granularity packing.

Parallel task scheduling is the biggest and hardest problem of any parallel language implementation. Given that many processor elements are available in a parallel computer system (Section 2.1), scheduling the parallel tasks among those computers is by no means an easy job. Even if there is no internal connection among the parallel tasks, scheduling them on those computers is still an NP hard problem [94, 114]. In fact, those parallel tasks are from the same program. They cooperate with each other to solve the same problem: each of them contributes a bit to the final result of the problem. Therefore, they have to communicate with each other, and what is more, some tasks may depend on the results of some other tasks. In other words, there have to be some communication and synchronization among those tasks. It makes the scheduling problem even more complex. Worse still, in some implementations, the number of the parallel tasks and the patterns of their communication and synchronization are not static but generated on the fly due to granularity packing.

Scalability over processor elements is another important issue of parallel program implementation. An implementation with good scalability can produce better performance when the number of the processor elements in a parallel computer system goes up, while a poor scalability implementation may not be able to take the advantage of the increasing processor element number and even drop the performance.

Finally, a theoretical performance calculation model is always desirable. There are some proposals of using chemical kinetics calculation in parallel computing [32, 31]. As T-Cham is based on the chemical reaction metaphor, chemical kinetics may be able to serve as a theoretical performance model for the execution of a T-Cham program on a particular parallel computer system.

6.2 The Execution Model of T-Cham

This section mainly concentrates on the abstract execution model of T-Cham, which we call *T-Cham Machine*. The model is independent of any particular computer architecture. It can be easily realized on a MIMD computer.

As the reactions of a T-Cham program rely on large dynamic sets of tuples, the management of tuples and tuple spaces is one of the main issues in a T-Cham implementation. To avoid synchronizations across different computers or computational nodes, which are very expensive (and time-consuming) operations, it is desirable to keep all the *related* tuples in one place. A straightforward implementation therefore uses the master/workers paradigm, where a host node (master) stores tuples and tests the reaction conditions, and a number of worker nodes execute the reaction rules. Each of the worker nodes bids for tasks to execute from the host node, Figure 6.1(a). This kind of implementation is very inefficient when there is heavy communication between the host node (tuple space) and the worker nodes, and it does not scale well.

Our approach is an extension to the basic master/workers structure in that we may have more than one master, Figure 6.1(b). Every master holds a piece of the original tuple space, and the workers bid for tasks from one of those masters. As the duties of the masters are mainly on tuple management while a certain group of tuples symbolizes the generation of a new task, a master is thus called a *task manager*, or simply a *manager*; accordingly, a worker is a *task executor*, or just an *executor*. The managers hold all the tuples of a T-Cham program, and the executors bid for tasks from those managers (by the action we call *task bid*). The execution of a reaction rule under T-Cham Machine starts from an executor which bids for a task from one of the managers. After the executor receives the task, it executes the operations of this reaction and then returns the tuples generated. An executor can only communicate with one of the managers to bid a task or send the newly generated tuples back. It cannot communicate with another executor. A manager can communicate with those executors as well as other managers to maintain tuple space integrity, including managing *tuple migration*, or pass a newly received bid, *bid handling*, if this manager has no task ready to go out for execution. The number of task managers varies with the problem to be solved and the machine on which this program is executed.

With the co-existence of multiple masters, there is more than one place where a tuple can go and a task can be from. This makes task scheduling more complicated than the simple master/worker structure. To cope with the tuple space partition and data integrity among the multiple masters, task distribution, and the communication between the masters and the workers etc. problems in this new environment, we suggest four basic algorithms, *Tuple Space Partition*, *Tuple Migration*, *Task Bid Handling*, and *Task Bidding and Receiving* in the subsequent sections. Those four algorithms are the

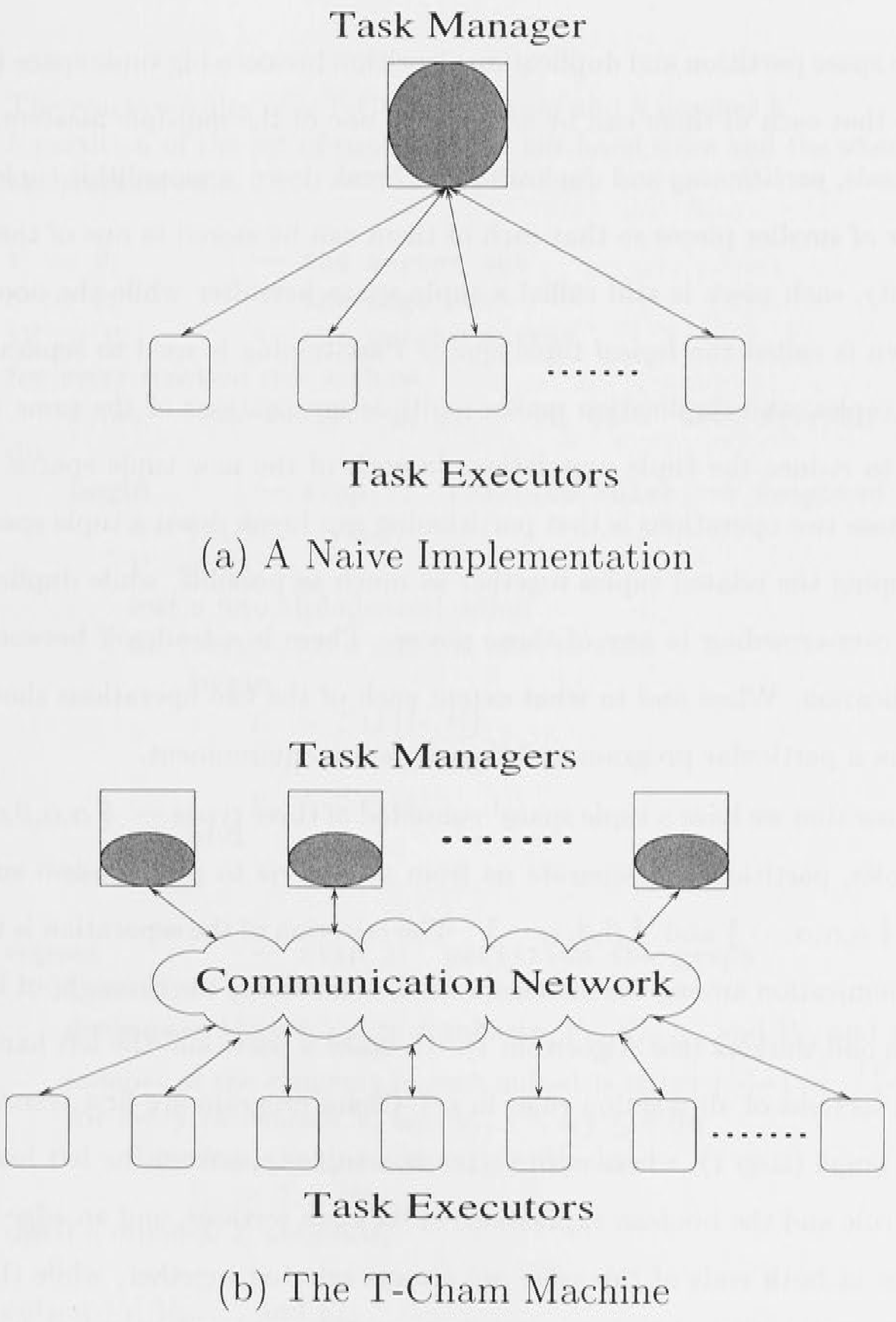


Figure 6.1: The Implementation of T-Cham

central part of the T-Cham Machine.

6.2.1 Tuple Space Partition and Duplication

Tuple space partition and duplication algorithm breaks a big tuple space into smaller pieces so that each of them can be assigned to one of the multiple masters. There are two methods, *partitioning* and *duplication*, to break down a monolithic tuple space into a number of smaller pieces so that each of them can be stored in one of the managers. For brevity, each piece is still called a tuple space hereafter while the one before the breakdown is called the *logical* tuple space. Partitioning is used to separate different types of tuples, and duplication makes multiple incarnations of the same tuple space in order to reduce the tuple populations in each of the new tuple spaces. The basic idea of those two operations is that partitioning can break down a tuple space to pieces while keeping the related tuples together as much as possible, while duplication helps to avoid over-crowding in any of those pieces. There is a trade-off between partition and duplication. When and to what extent each of the two operations should be used depend on a particular program and the execution environment.

Suppose that we have a tuple space[1] consisted of three types — $\{\!|\ \alpha,\alpha,\beta,\alpha,\gamma,\beta,\alpha,\cdots\ |\!\}$ — of tuples, partition can separate αs from βs and γs to produce two smaller tuple spaces: $\{\!|\ \alpha,\alpha,\alpha,\cdots\ |\!\}$ and $\{\!|\ \beta,\beta,\gamma,\cdots\ |\!\}$. The criterion of the separation is to *minimize* the communication among the managers while *maximizing* the throughput between the managers and workers (see Algorithm 1). To make a partition, the left hand sides and the `when` sections of all reaction rules in a T-Cham program are first transformed to a *weighted graph* (step 1), where each vertex is a tuple type from the left hand side of a reaction rule and the boolean expressions of its `when` sections, and an edge means that the tuples at both ends of this edge are always selected together, while the weight w of an edge means that the edge comes from w different reaction rules. In step 2, the graph is then partitioned into k pieces: the number of vertices in each piece is either $\lfloor \frac{n}{k} \rfloor$ or $\lfloor \frac{n}{k} \rfloor + 1$, where n is the number of vertices in the original graph. The weight summation (which corresponds to the communication cost) of cutting edges has the

[1]We use $\{\!|$ and $|\!\}$ to denote a tuple space. For example, $\{\!|\ \alpha\ |\!\}$ means a tuple space which has one tuple α in it.

Algorithm 1 (Tuple Space Partition)

INPUT: The reaction rules of a T-Cham program and a number k

OUTPUT: k partition of the set of tuples on the left hand sides and the `when` parts of the reaction rules.

METHOD:

```
$\mathcal{V}$ := $\emptyset$;            -- the vertex set
$\mathcal{E}$ := $\emptyset$;            -- the edge set
$\mathcal{W}$ := 0;            -- the weight matrix
for every reaction rule such as
    $x_1, x_2, \cdots, x_n$ leadsto $y_1, y_2, \cdots, y_m$ by $T$ when $f(x_1, x_2, \cdots, x_n)$
do
    begin          -- step 1:  reaction rules --> weighted graph
        $v = \{x_1, x_2, \cdots, x_n\}$;
        $\mathcal{V} := \mathcal{V} \cup v$;
        sort $v$ into alphabetical order;
        for every $s$, $t \in v$ and $t$ is next to $s$ do
            begin
                $\mathcal{E} := \mathcal{E} \cup \{(s,t)\}$;
                $\mathcal{W}_{st} := \mathcal{W}_{st} + 1$;
                $v := v - \{s\}$;
            end
    end

repeat             -- step 2:  partition the graph
    $\Sigma := 0$;
    decompose the set $\mathcal{V}$ into $k$ subsets: $\mathcal{V}_1$, $\mathcal{V}_2$,..., and $\mathcal{V}_k$, and the
    number of the elements in each subset is either $\lfloor \frac{|\mathcal{V}|}{k} \rfloor$ or $\lfloor \frac{|\mathcal{V}|}{k} \rfloor + 1$;
    for every two subset $\mathcal{V}_i$ and $\mathcal{V}_j$, $1 \le i, j \le k$ do
        for every $s \in \mathcal{V}_i$ and every $t \in \mathcal{V}_j$ do
            $\Sigma := \Sigma + \mathcal{W}_{st}$;
until a minimal $\Sigma$ obtained;

output $\mathcal{V}_1$, $\mathcal{V}_2$,..., and $\mathcal{V}_k$;
```

Figure 6.2: The Tuple Space Partition Algorithm

minimum value among all possible partitions.

Taking the Vending Machine problem (Example 5.3) for example: the graph obtained from the reaction rules and its partition are in Figure 6.3.

As graph partition algorithms are NP hard, most of them use heuristic rules to reduce the computation time. The graphs obtained from the reaction rules of T-Cham programs so far have reasonably small number of vertices (no more than 100); thus, we partition them by brute-force. A better algorithm may be used, but it does not affect the idea discussed in this thesis.

If the population of the tuples in one of those tuple spaces, say, the one with the α type tuples, is still very large, the tuple space therefore needs to be continuously broken down into even smaller pieces (in terms of tuple population instead of tuple types) to alleviate communication congestion. The tuple space (not the tuples themselves) is then duplicated. In other words, if the population of people in a building is too high, we build another (or another n) building exactly the same as this building, and move half (or $\frac{1}{n}$ population) of the population to the new building. For example, the α type tuple space can be made into, for example, three incarnations: each of them holds one third of the original tuple population.

6.2.2 The Communications Between Task Managers

There are two kinds of communications between any two task managers. The first is that one manager asks tuples from another (*tuple migration*), and the second is task bid receiving and passing (*bid handling*).

We adopt a distributed *two phase locking* (2PL) protocol [28, pp. 77–78] for tuple migration. When a task manager, known as *asking site* in Algorithm 2, needs some tuples which are not available in the local tuple space at the moment, it immigrates those kinds of tuples from remote managers. To keep the integrity of the tuple space, the local manager first acquires all the necessary locks on the tuples from the remote managers, known as *asked sites* in Algorithm 2, and then withdraws those tuples to its local tuple space. Finally, the locks are released after the tuples have been immigrated. The asking site follows this sequence: (i) send out messages to the corresponding sites asking for locks, (ii) wait for the replies from each site, and (iii) if all agree, i.e., the

```
reactionrules

        ausd2, large leadsto LargeBar;

        ausd2, small leadsto SmallBar, ausd1;

        ausd1, ausd1 leadsto ausd2;

        ausd1, small leadsto SmallBar;
```

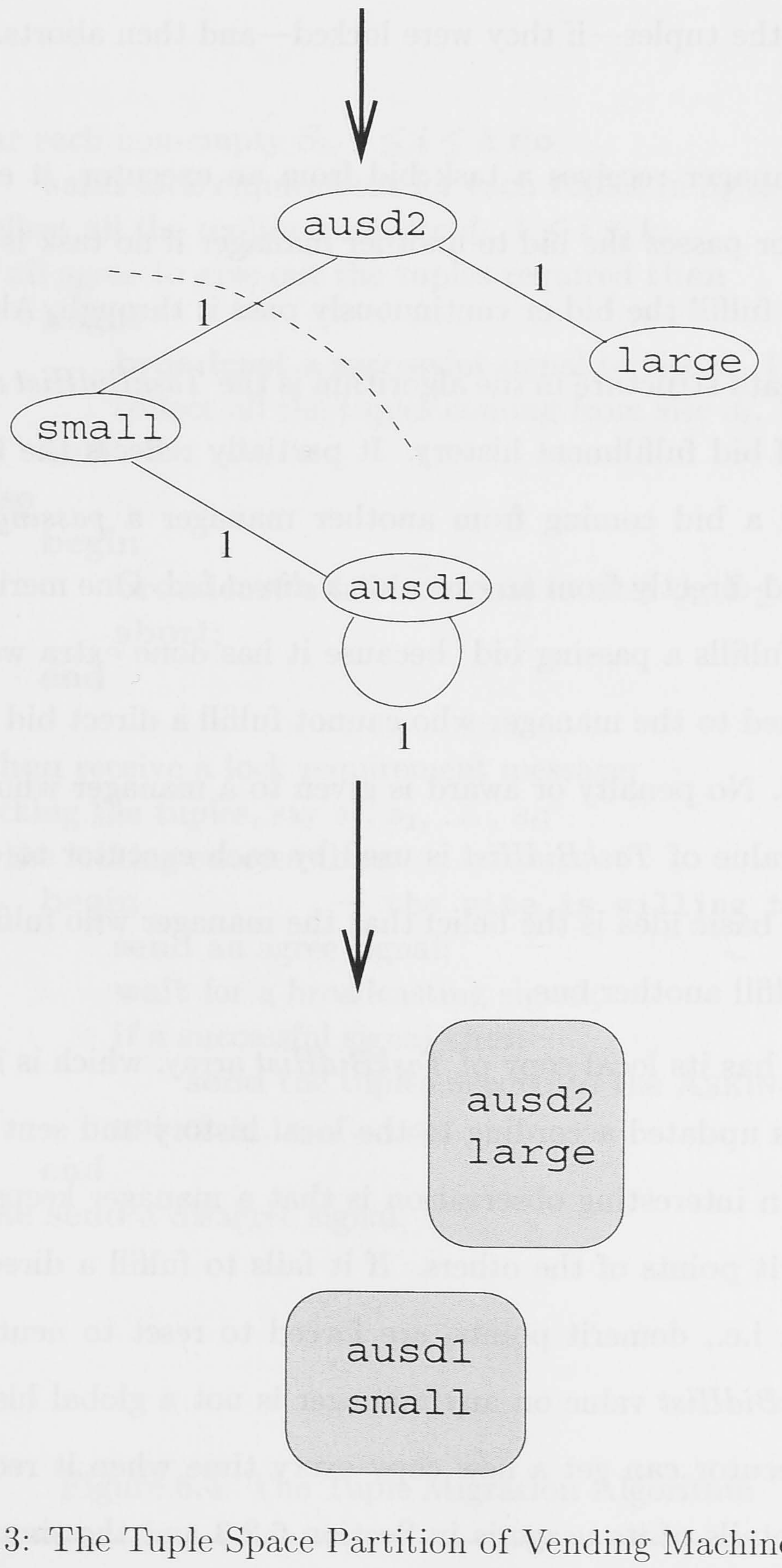


Figure 6.3: The Tuple Space Partition of Vending Machine Problem

locking on every site is successful, broadcast a successful signal to all related sites, and wait for the tuples come out from those sites; otherwise, broadcast a failure signal and then abort. In an *asked site*, activities proceed like this: (i) upon receiving a lock requirement, the site tries to lock the tuples specified; if successful, replies an agree signal and waits for further replies; otherwise, a disagree signal and then abort, (ii) on receiving a successful signal, i.e., the locks on every site are successful, this site sends out the tuples needed by the asking site and then unlocks the tuples if necessary; otherwise, unlocks the tuples—if they were locked—and then aborts. See Algorithm 2 for details.

Whenever a manager receives a task bid from an executor, it either sends out a task for execution or passes the bid to another manager if no task is ready to go. The other manager will fulfill the bid or continuously pass it through, Algorithm 3.

An important data structure in the algorithm is the *TaskBidHist* array, which keeps the local version of bid fulfillment history. It partially reflects the task loads of each manager. We call a bid coming from another manager a *passing bid* to the local manager while a bid directly from an executor a *direct bid*. One merit point is given to the manager who fulfills a passing bid because it has done extra work; meanwhile, a demerit point is fined to the manager who cannot fulfill a direct bid because it cannot do its own job well. No penalty or award is given to a manager who passes through a passing bid. The value of *TaskBidHist* is used by each executor to choose a manager for a task bid. The basic idea is the belief that the manager who fulfilled one extra bid is most likely to fulfill another one.

Every manager has its local copy of *TaskBidHist* array, which is initialized to 0s at the beginning. It is updated according to the local history and sent out with tasks to a task executor. An interesting observation is that a manager keeps the merit points of itself and demerit points of the others. If it fails to fulfill a direct bid, its records of other managers, i.e., demerit points, are forced to reset to neutral, i.e., 0 (zero). Although the *TaskBidHist* value on any manager is not a global history, it causes no problem, as an executor can get a new copy every time when it receives a task from a manager. The details of its usage is in Section 6.2.3 and the simulation result is in Section 6.3.

Algorithm 2 (Tuple Migration) *To generate a task T of the reaction rule:*

$x_1, x_2, \cdots, x_n$ `leadsto`... `by` T `when` $f(x_1, x_2, \cdots, x_n)$,

a manager needs the tuples of $\{\!|x_1, x_2, \cdots, x_n|\!\}$. Grouping the tuples according to where they belong, we have S_i and d_i, $1 \leq i \leq k$: the tuples in S_i can be got from the manager d_i.

```
ASKING SITE: for each non-empty $S_i$, $1 \leq i \leq k$ do
                 send lock requirement for each tuples in $S_i$ to site $d_i$;
             collect all the replies from site $d_i$, $1 \leq i \leq k$;
             if all agree to give out the tuples required then
                 begin
                     broadcast a successful signal to site $d_i$, $1 \leq i \leq k$;
                     collect all the tuples coming from site $d_i$, $1 \leq i \leq k$;
                 end
             else
                 begin
                     broadcast a failure signal to sites $d_i$, $1 \leq i \leq k$;
                     abort;
                 end

ASKED SITE:  when receive a lock requirement message;
             locking the tuples, say $s_1$, $s_2$, ..., $s_t$;
             if the locking success then
                 begin      -- the site is willing to give out the tuples
                     send an agree signal;
                     wait for a broadcasting signal;
                     if a successful signal then
                         send the tuples needed to the ASKING SITE;
                     unlock $s_1$, $s_2$, ..., $s_t$;
                 end
             else send a disagree signal;
```

Figure 6.4: The Tuple Migration Algorithm

Algorithm 3 (Task Bid Handling) *The algorithm executes on every manager.*

```
    TaskBidHist := 0;
    self := the identity of the node running this algorithm;
    when receive a message;
        case: a task bid message from an executor E;
            if there is at least one task ready to go then
                send a task with the current value of TaskBidHist to E;
            else
                begin
                    send the bid to its neighbor manager M;
                    TaskBidHist[self] := −1;   -- unable to do its job well
                    for all x, x ≠self do
                        if TaskBidHist[x]< 0
                            then TaskBidHist[x] := 0;
                                  -- reset the demerit records of the other managers
                end;
            break;

        case: a task bid message, originally from E, passed from another manager M′;
            if there is at least one task ready to go then
                begin
                    TaskBidHist[self] := TaskBidHist[self]+1;
                    TaskBidHist[M′] := TaskBidHist[M′]−1;
                    send a task with the current value of TaskBidHist to E;
                end
            else
                send the bid to its neighbor manager M;
            break;

        others: other actions;
```

Figure 6.5: The Bid Handling Algorithm

Algorithm 4 (Task Bidding and Receiving) *The algorithm is executed on every Executor.*

```
bid_target := F(TaskBidHist);                  -- bid a task
send a bid message to task manager bid_target;
wait for a task and the new TaskBidHist value;
update the TaskBidHist array;
execute the task;
for every generated tuple $x$ do               -- send back the tuples generated
    if $x$ has only one place to go, say $M$
        then send $x$ to $M$
        else begin
            tuple_target := G(TaskBidHist);
            send $x$ to tuple_target;
        end
```

Figure 6.6: The Task Bidding and Receiving Algorithm

6.2.3 The Communications Between Task Managers and Executors

The load balance is an important problem in this proposed T-Cham Machine execution model, although it is not a problem in the original master/workers architecture, because there is only *one* master. As each node, a manager or an executor, knows nothing about the states of the others in our approach, task distribution is a difficult issue. Generally speaking, there are two types of task distributions: *manager-based* or *executor-based*. By manager-based distribution, whenever there is a task ready in a manager at a time, the task is sent out to an executor for execution. The destination (executor) is determined by a *choice function*. If that executor is busy, the task will be passed to another executor like the bid passing among managers. The other way is when there is a task ready, it waits in the manager for an executor to bid, i.e., executor-based distribution. An unloaded executor chooses one manager to bid for a task. The bid will be passed to another manager if it fails in this one. The choice of a manager is made by a choice function, too. Our implementation experience is based on the second approach.

Every executor works on a permanent loop (until a termination condition is satisfied) of three steps: bidding for a task, executing the task, and returning the generated tuples back to where they belong. The executor chooses a manager to send its bid message and a manager, if necessary (i.e., if the generated tuple has more than one "tuple space" to go), to return a tuple. Two choice functions, *bid-choice function* F and *return-choice function* G, are responsible for the choice. We call them F function and G function for brevity. Both of them are based on the *TaskBidHist* array, which is originally set to 0s, posing no discrimination to any manager, and then every coming task updates it to the value of the *TaskBidHist* array in the manager who fulfills this task bid.

There are many ways to define the functions F and G, but the basic philosophy is the same: the manager who fulfills an extra bid passed from the other managers is most likely to fulfill another one and the one who failed to fulfill a bid badly needs new tuples to generate more tasks; in other words, F favors the manager who gives more tasks while G favors the one of less tasks.

6.3 The Basic Performance Measurements: T-Cham Machine Implementation Case Study

This section gives some basic performance measurements of a prototype T-Cham implementation, which is conducted on the AP1000 multicomputer. The AP1000 multicomputer is a distributed memory MIMD machine, and the communications between the processing nodes are performed by point-to-point message passing or group broadcasting. We currently have 128 processing nodes installed at ANU. We refer the readers to Appendix B for more details about the structure of the AP1000 multicomputer.

In this section, we use an interactive program, the Sleeping Barber problem, to illustrate the communication patterns among masters and workers, the calculation of the *TaskBidHist* array, and the impact of this array to execution efficiency by calculating the *bid hit rate*. We will use the Matrix Multiplication problem as the example for the execution efficiency study.

6.3.1 The Basic Performance Experiment

Three kinds of experiments have been conducted on AP1000 using its message passing library. They are (i) the communication overhead for a reaction, (ii) the accuracy of the *TaskBidHist* array, and (iii) the bid hit rate (or the bid failure rate).

The execution of every reaction consists of three steps: bidding a task, receiving and executing the task, and sending back the tuples generated:

$$T_{reac} = T_{bid} + T_{recv} + T_{exec} + T_{ret} \tag{6.1}$$

where T_{reac} is the total time needed for the execution of a reaction, T_{bid} the time used to send a bid, T_{recv} the time to receive a task, T_{exec} the time used to execute the operations in the reaction, and finally, T_{ret} is the time used to return generated tuples to where they belong. The time needed for bid passing, if the bid cannot be fulfilled in the first place, is omitted for brevity. The communication overhead (T_{comm}) for a reaction is

$$T_{comm} = T_{bid} + T_{recv} + T_{ret} \tag{6.2}$$

The time used for any communication can be roughly divided into two parts: the time for synchronization (including hand-shaking etc.) and that for real data transmission, i.e.,

$$T = T_{sync} + T_{trans} \tag{6.3}$$

where T_{sync} is almost a constant for a particular communication function call, and T_{trans} is proportional to the size of data to be transmitted[2]. Formula 6.2 is thus rewritten as

$$T_{comm} = 3 \times T_{sync} + \frac{\text{total size of data transmitted}}{\text{communication bandwidth}} \tag{6.4}$$

The real time of communication overhead for a reaction with different amount of data transmission involved is in Figure 6.7. The details of AP1000 communication mechanism and performance data can be found in [156].

[2]The communication over AP1000 network is almost distance independent, and the message passing bandwidth is constant. We ignore network contention, which is rarely observed in this kind of programs [100].

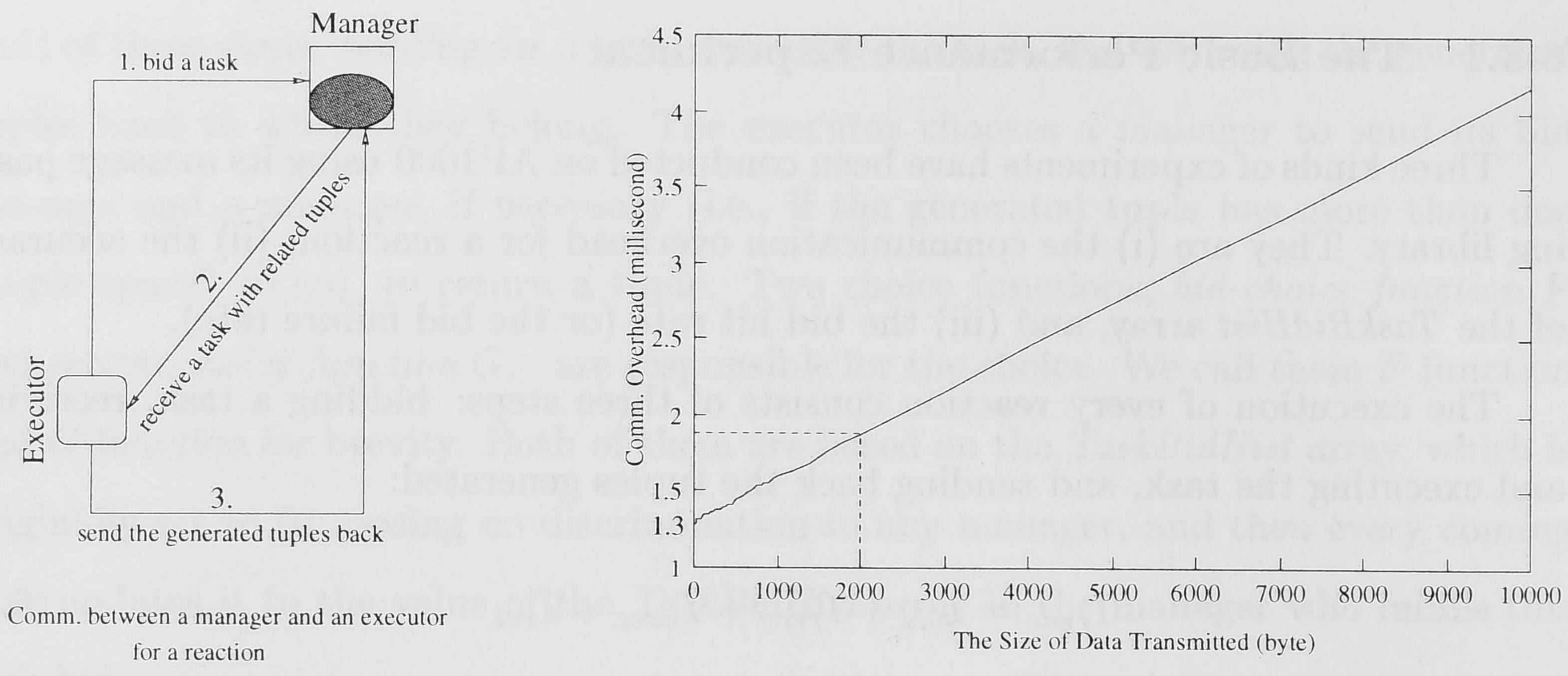


Figure 6.7: The Communication Overhead for a Reaction

The isoefficiency concept [97, pp. 125–129] is often used to analyze the scalability of a parallel program or algorithm. The efficiency (E) of a parallel program on a given parallel computer is:

$$E = \frac{w(s)}{w(s) + h(s,n)}, \tag{6.5}$$

where s is the problem size, n the computer size, say, the number of computational nodes, $w(s)$ the useful computations which normally grows in the order $\mathcal{O}(s)$, and $h(s,n)$ are overhead attributed to synchronizations and data communications.

The *bid hit rate*, defined as the ratio of the number of direct bids to the number of total bids, is another factor affecting good performance. The higher the rate means the less the bid passing, and hence, a better performance. The value of the *TaskBidHist* array and the bid hit rate are two very close things. Taking the Sleeping Barber problem as an example, the logical tuple space of the problem can be decomposed into four pieces (therefore, four task managers): the first one contains the tuple types of `pin`, `bsp`, and `chair`, the second one `pcut` and `bwk`, the third one `pwt` and `bfin`, and the fourth one `pout`. In this example, we have four task executors, which bid for tasks for execution from those managers, Figure 6.8(a). The example gives a clear illustration on tuple space partitioning and the run time communication patterns. As only one barber is available in the original problem (Example 6), there could not be any concurrent tasks. To be more general, we adapt the problem to more than one barber sharing a common barber shop so that the barbers can work concurrently. Figure 6.8(b) is the so-called

Space-Time Diagram [108] of the first several rounds of an execution of Sleeping Barber (with 5 barbers) on the AP1000 computer. The computational nodes (called "cells" in AP1000) 0 to 3, `cell 0` to `cell 3`, work as the four task managers and nodes 4 to 7, `cell 4` to `cell 7`, the four task executors. From the diagram, we can see the communications needed for any reaction: task bidding, bid passing, task receiving and tuple returning,and the time spent on the communications and a real computation. The calculation of the *TaskBidHist* array and hit rate are shown in Figure 6.9.

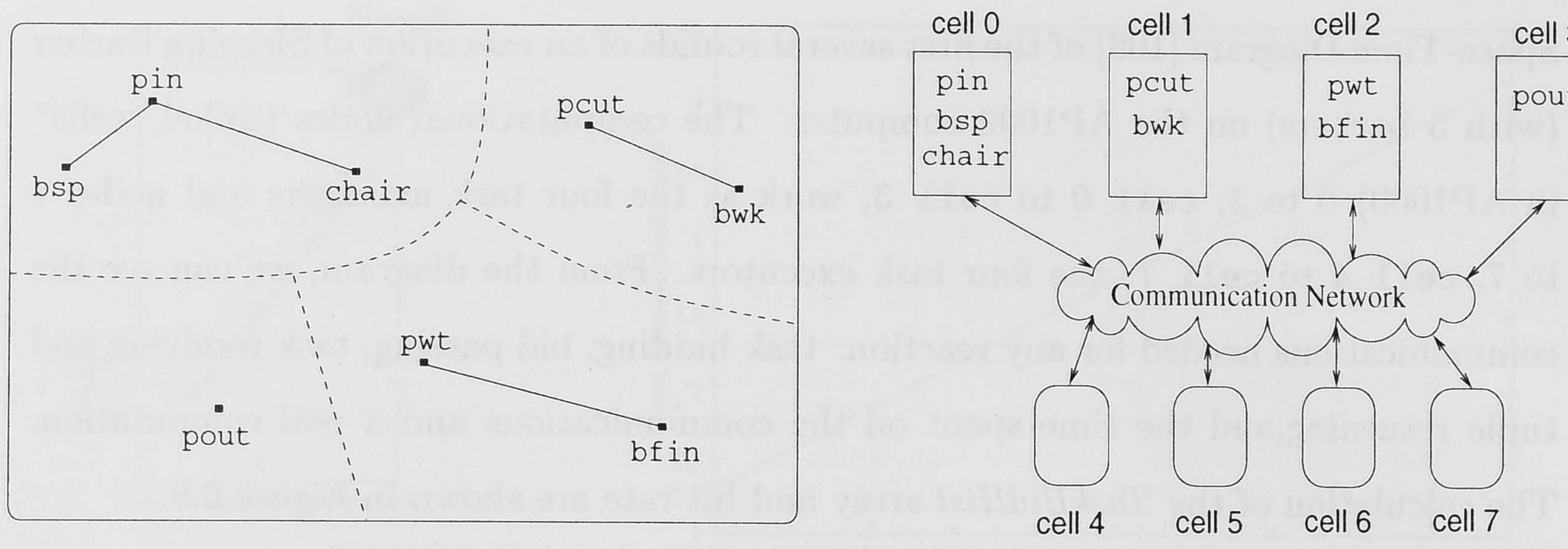
The functions F and G take the *TaskBidHist* array as input and return the task manager chosen. A simplest definition ignores the value of *TaskBidHist* and gives the same chance to each of the task managers:

$$F = \text{random}(x) \ \textbf{mod} \ 4, \tag{6.6}$$

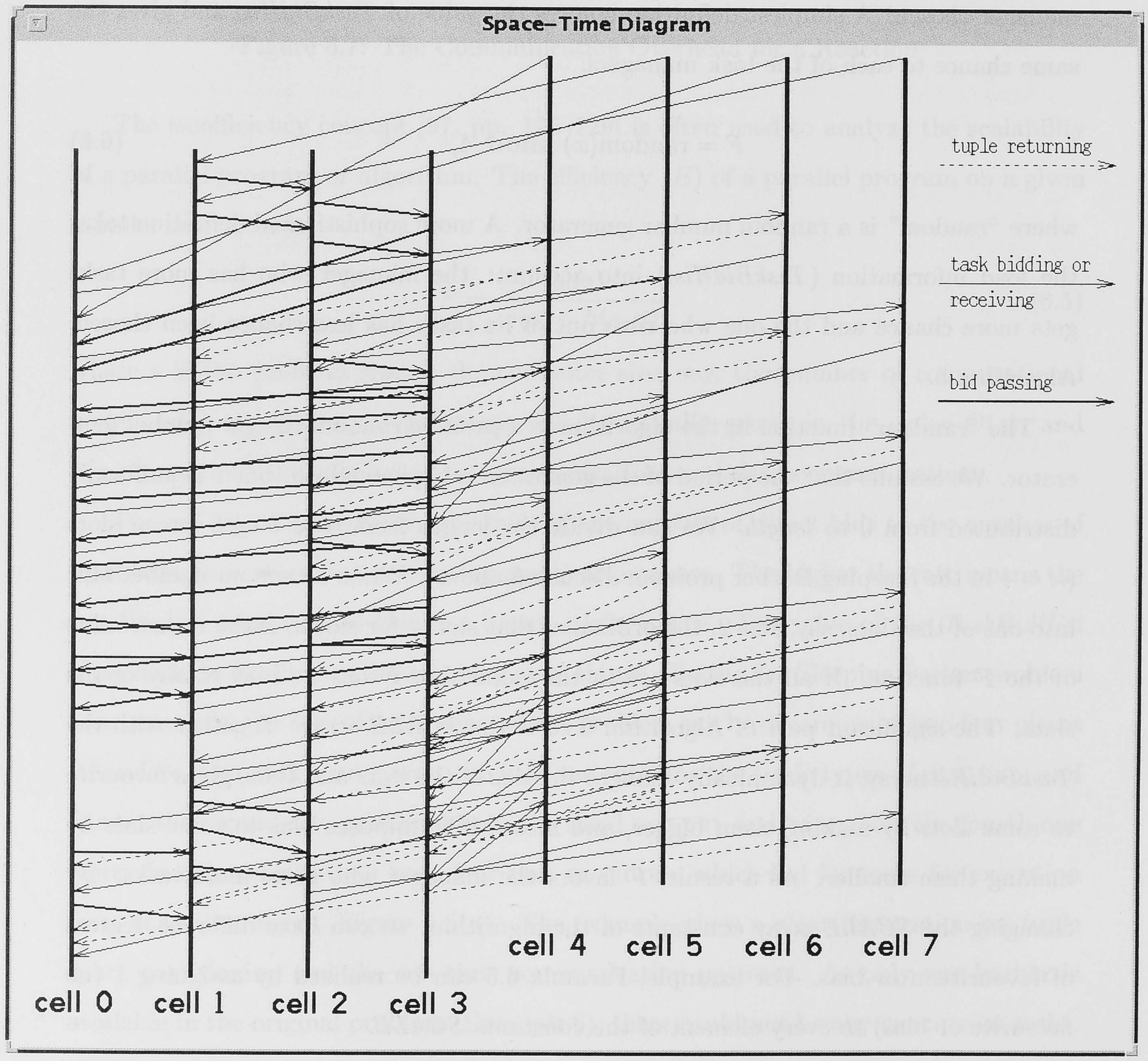
where "random" is a random number generator. A more sophisticated definition takes the load information (*TaskBidHist*) into account: the manager who has more tasks gets more chance and the one who runs out of its tasks has less chance been chosen, Algorithm 5.

The "random" function in the algorithm is a positive random integer number generator. We assume that the period of the generator is big enough so that x is uniformly distributed from 0 to *length*. We can divide the length from 0 to *length* into n slots ($n = 4$ in the Sleeping Barber problem discussed above). When a random number falls into one of the slots, say, slot 2, the ordinal of that slot, 2 for slot 2, is the return value of the F function. If all the slots are in the same size, F has no bias to any of the slots. The significant part of Algorithm 6.10 is the $SCALE$ array. Together with the *TaskBidHist* array, it dynamically changes the size of the slots and thus, gives favourite to some slots by making them bigger, and meanwhile, imposes bias to some slots by making them smaller. As a result, F favors the manager who gives more tasks. By changing the $SCALE$ array constants of the algorithm, we can have different degrees of favouritism or bias. For example, Formula 6.6 can be realized by assigning 1 (no favourite or bias) to every element of the constants $SCALE$.

G can be defined in a similar way. The only difference is that when two managers host a same tuple type, the one who has less value of *TaskBidHist* has a better chance



(a) Tuple Space and the T-Cham Machine



(b) Space-Time Diagram

Figure 6.8: Tuple Space Partition and Communication Patterns of Sleeping Barber

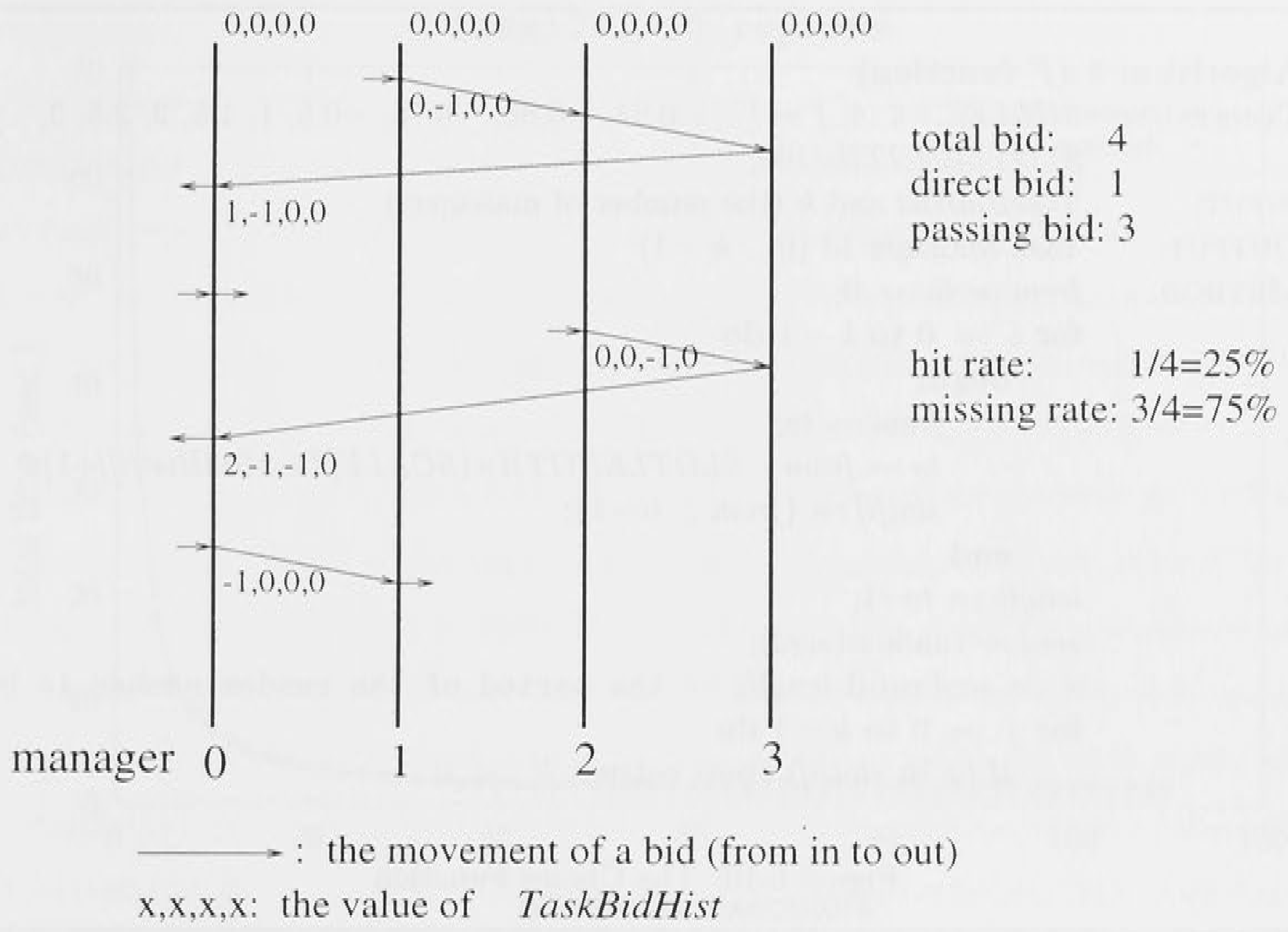


Figure 6.9: The Calculation of *TaskBidHist* and Hit Rate

to be chosen.

The hit rate for the first several rounds of an execution of Sleeping Barber is 53% with Formula 6.6, and 79% under Algorithm 5. Figure 6.8(b) is its Space-Time diagram under Formula 6.6.

6.3.2 The Performance Measurements of Matrix Multiplication

The T-Cham program of Matrix Multiplication is in Section 8.4. For simplified description, imagine that each row of the first matrix and each column of the second matrix are represented as tuples in a tuple space. We call the first one x and the second one y. A reaction rule withdraws one x and one y to produce one product element z.

As the tuple space of the problem has only three types of tuples, there is no need for tuple space partition, but duplication is needed for a good performance when the computer nodes used for the execution scale up. The performance curves, Figure 6.11, suggest that in the case of 128x128 matrix multiplication, one manager is good for every 30–40 executors. From Figure 6.11(b), which is an enlargement of performance

```
Algorithm 5 ($F$ function)
CONSTANT:  $SCALE[..-4..4..] = \{.., -0.94, -0.88, -0.75, -0.5, 1, 1.5, 2, 2.5, 3, ..\}$;
           $SLOTLENGTH$=100;
INPUT:     $TaskBidHist$ and $k$ (the number of managers)
OUTPUT:    Task Manager Id (0 .. $k-1$)
METHOD:    $from := to :=\ 0$;
           for $i\ :=\ 0$ to $k-1$ do
               begin
                   $from := to$;
                   $to := from+\ SLOTLENGTH\times(SCALE[TaskBidHist[i]]+1)$;
                   $slot[i] := \{from\ ..\ to-1\}$;
               end
           $length := to-1$;
           $seed :=$ random($seed$);
           $x\ := seed$ mod $length$; -- the period of the random number is big enough
           for $j\ :=\ 0$ to $k-1$ do
               if ($x$ in $slot[i]$) then return $F\ :=\ i$;
```

Figure 6.10: The Choice Function

curve segment from the 40th executors to the 110th executors in Figure 6.11(a), readers can easily figure out the effect of the second task manager.

This result can also be obtained by a simple calculation from the basic performance measurement data in Section 6.3.1. The volume of data involved in every execution of `multiply` is approximately (8 byte per `float` number in AP1000):

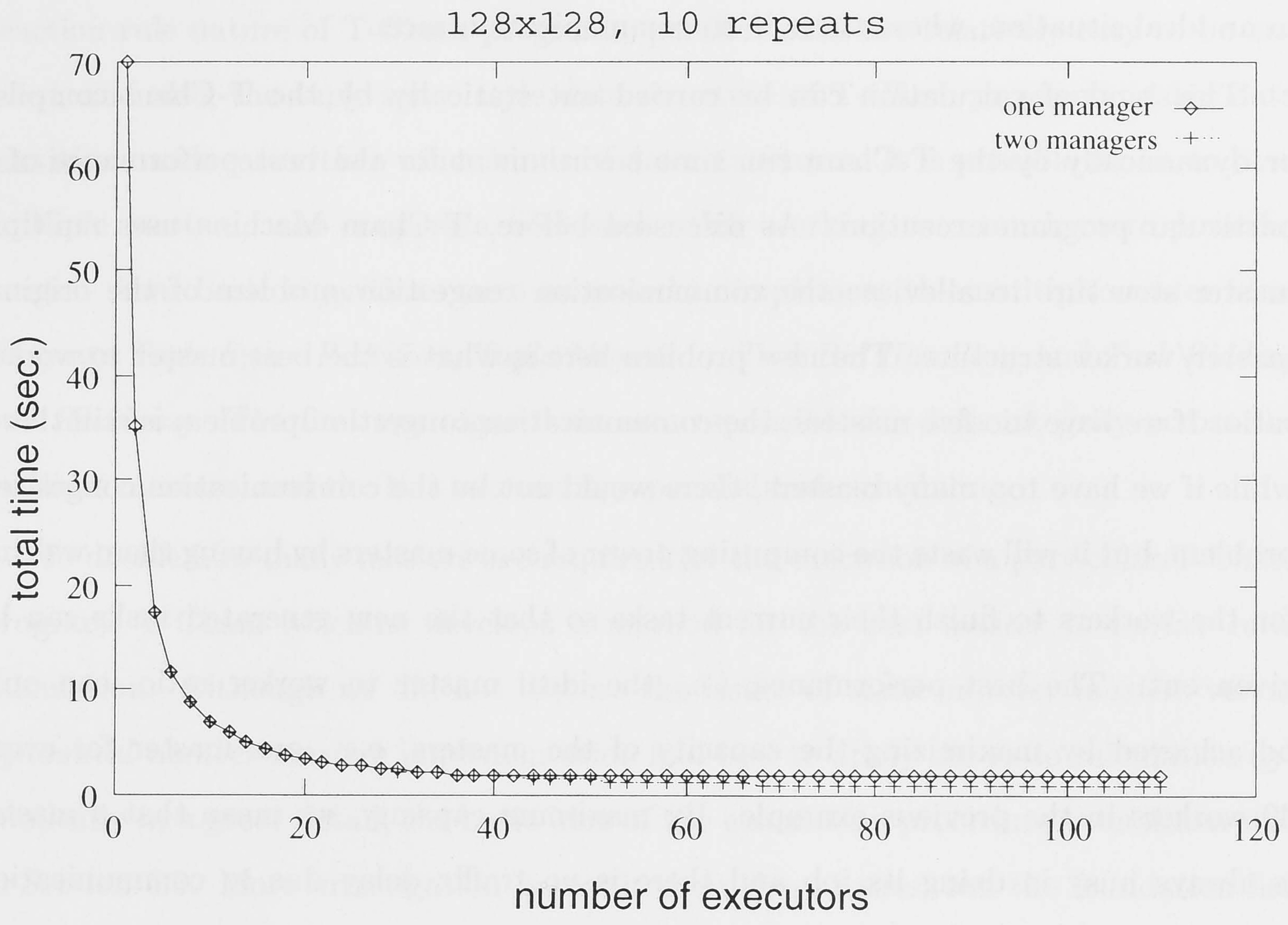
$$128 \times 8 \text{ (tuple } \mathtt{x}) + 128 \times 8 \text{ (tuple } \mathtt{y}) + \text{others}(\textit{tupleheadetc.}) \approx 2000 \text{ (bytes)} \quad (6.7)$$

From Figure 6.7, the time cost of a communication is the time of data transmission plus three time synchronizations (Formula 6.4); thus the communication time needed for every `multiply` execution (two synchronizations) is:

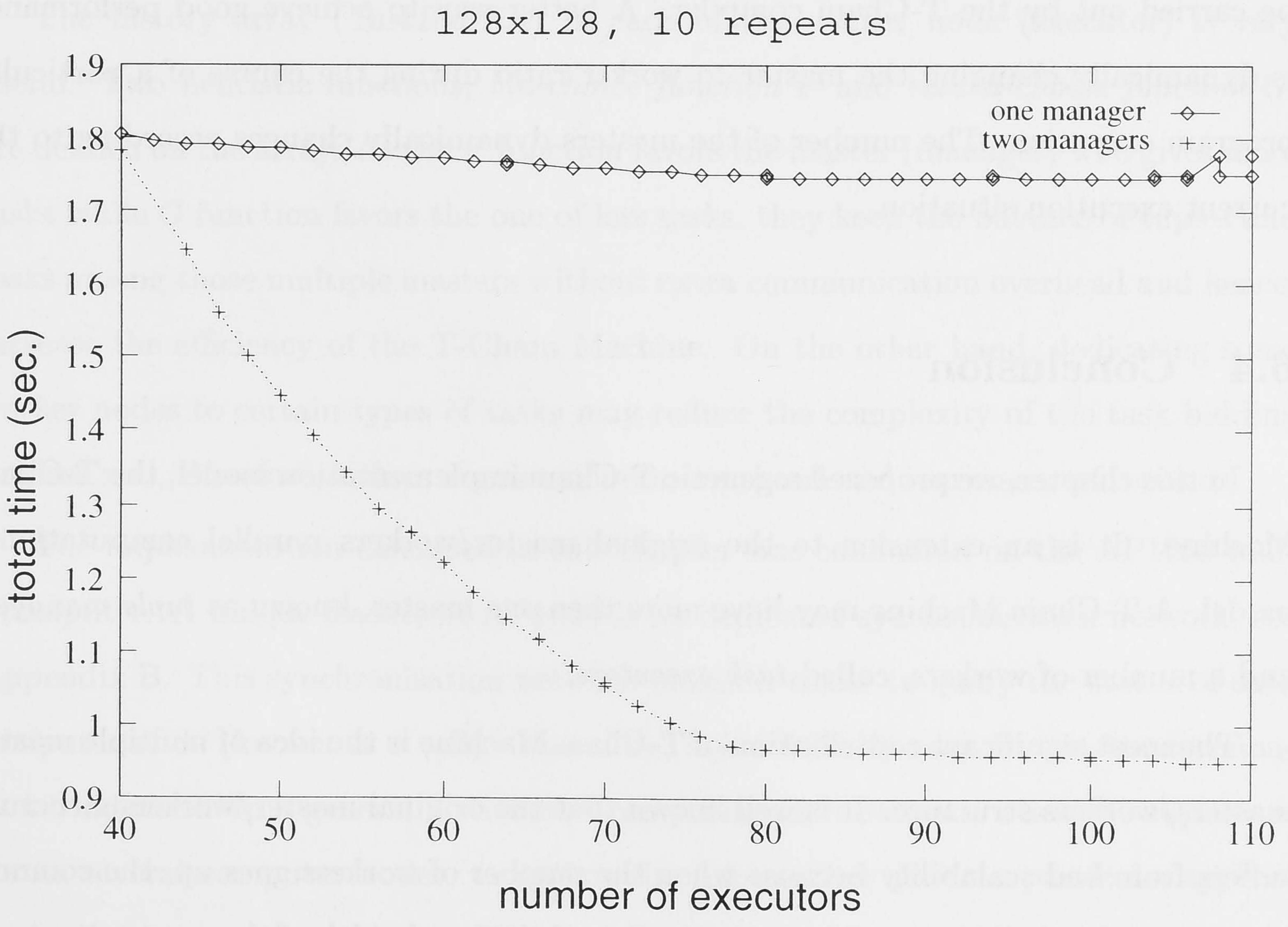
$$T_{comm} = T_{2000} - \frac{T_{sync}}{3} = 1.8 - \frac{1.3}{3} = 1.4 \text{ (millisecond)} \quad (6.8)$$

The average calculation time for an execution of x×y is 54.4 (millisecond). The ideal performance can be obtained if a task manager always keeps busy while no bid waits for communication, i.e., the maximum capacity of a manager is

$$\frac{54.4}{1.4} \approx 39 \text{ (executor)}, \quad (6.9)$$



(a) The Number of Executors from 1 to 110



(b) The Number of Executors from 40 to 110

Figure 6.11: Performance of Matrix Multiplication (128x128)

in an ideal situation, where there is no communication race.

This kind of calculation can be carried out statically by the T-Cham compiler or dynamically by the T-Cham run time environment for the best performance of a particular program execution. As discussed before, T-Cham Machine uses multiple master structure to alleviate the communication congestion problem of the original master/worker structure. The new problem here is what is the best master to worker ratio. If we have too few masters, the communication congestion problem is still there, while if we have too many masters, there would not be the communication congestion problem, but it will waste the computing power of some masters by having them waiting for the workers to finish their current tasks so that the new generated tasks can be given out. The best performance, i.e., the ideal master to worker ratio, can only be achieved by maximizing the capacity of the masters, e.g., one master for every 39 workers in the previous example. By maximum capacity, we mean that a master is always busy in doing its job and there is no traffic delay due to communication congestion. If the number of the masters is fixed (static), the ratio calculation can be carried out by the T-Cham compiler. A better way to achieve good performance is dynamically changing the master to worker ratio during the course of a particular program execution. The number of the masters dynamically changes according to the current execution situation.

6.4 Conclusion

In this chapter, we proposed a generic T-Cham implementation model, the T-Cham Machine. It is an extension to the original master/workers parallel computational model. A T-Cham Machine may have more then one master, known as *tuple manager*, and a number of workers, called *task executors*.

The most significant contribution of T-Cham Machine is the idea of multiple master masters/workers structure. It is well known that the original master/workers structure suffers from bad scalability because when the number of workers goes up, the communication to the single master goes up as well, and the bandwidth of the master becomes the bottleneck and restricts the maximum number of workers it can have. Given the reaction rule nature of T-Cham programs, parallel tasks are relatively easy to be isolated, and also, the tuple space style data structure of T-Cham programs makes data partition much easier then other kinds of data structures. It is possible for us to have multiple masters, who cooperate with each other working as a much more powerful master. Four basic algorithms were proposed to implement this multiple master idea. They are *Tuple Space Partition*, *Tuple Migration*, *Task Bid Handling*, and *Task Bidding and Receiving*. The first two algorithms are responsible for data integrity across the multiple masters, and the last two manage parallel task scheduling.

To decide how many masters are required for the execution of a particular T-Cham program, T-Cham Machine develops a method for the best *master to worker ratio* calculation. Although we did not discuss the issues of *static number masters* versus *dynamical number masters* implementation and how to apply the ratio calculation dynamically in a great detail, the basic idea of the calculation procedures was illustrated in Section 6.3. More investigation is required on those issues, but the foundation has been laid.

The history array (*TaskBidHist*) in each of the worker node (executor) is very useful. Two heuristic functions, *bid-choice function* F and *return-choice function* G, are defined on the array. As the F function favors the master (manager) who gives more tasks while G function favors the one of less tasks, they keep the balance of tuples and tasks among those multiple masters without extra communication overhead and hence, increase the efficiency of the T-Cham Machine. On the other hand, dedicating some worker nodes to certain types of tasks may reduce the complexity of the task bidding process and, therefore, reduce communication overhead and increase performance.

The implementation discussed in this chapter was conducted on the AP1000 multicomputer. A unique feature of AP1000 is its dedicated synchronisation network, see Appendix B. This synchronisation network makes it easier to query the status of each computer node (known as a cell). Therefore, it is relatively easy to test the termination conditions. In a general distributed computing environment, when the tuples are across several computer nodes, to gather the information about a particular type of tuples is a very costly operation. The simplest way to gather the information is to send out a halt request to every node involved and then ask for the information about the tuples. This is very inefficient. Another approach is to lock only the relevant tuples. A distributed *two phase locking* (2PL) protocol [28, pp. 77–78] can be used to do the locking. In addition, a deadlock-preventing algorithm should also be applied to avoid any potential deadlock. For detailed discussion in this area, we refer readers to [39].

Finally, BSP [164, 89, 37] is another encouraging model for T-Cham implementation. In BSP, the execution of a program is achieved by a number of sequential steps of so-called *supersteps*. Each superstep consists of three phases. They are the local computation in every processing node of the underlying computer, communication amongst them, and a barrier synchronisation, which waits for each of the local computation to finish and then moves forward to the next superstep. The superstep of BSP well matches the concurrent reactions of the same reaction rule of T-Cham, but the implementation of the global accessible tuple space on BSP seems not so trivial and requires more investigation.

Chapter 7

Towards the Temporal Logic Proof System of T-Cham

In this chapter, we concentrate on the *application* of a temporal logic system to T-Cham program verification. As it is about the application of temporal logic, we take the theoretic results of temporal logic for granted and do not repeat the proof procedures. We do not always stick to the rigid temporal logic regime either[1], but we will refer interested readers to the related references.

We first briefly introduce the temporal logic system, which is based on Manna-Pnueli temporal logic [130, 129], and then, in Section 7.2, we adapt the temporal logic system to T-Cham. We also develop a number of definitions and transformations by means of which we can translate a T-Cham program into a set of temporal logic formulae. In Section 7.3, we study the impact of racing between the reaction rules of a T-Cham program on its temporal logic proof system. Section 7.4 gives some examples on how to use the temporal logic system to prove some important properties of T-Cham programs.

[1]There are many proposed formal proof systems using different mathematical tools to prove the correctness of programs, but most of them primarily emphasize on the mathematical theories themselves. We believe this is the reason why they are not widely accepted by programmers, who are actually supposed to use those systems. Programmers, whenever possible, tend to think more intuitively, for example, the Venn Diagrams [53] in set theory.

The impact of T-Cham termination conditions on the temporal logic proof system is discussed in Section 7.5. Finally, we conclude the chapter with a short summary in Section 7.6.

7.1 Temporal Logic

Temporal logics [148, 83, 111, 130] are very useful tools to reason about temporal properties and especially suitable for modelling and reasoning about parallel and distributed applications. They are widely used in program specification and verification, even as programming languages on their own right [141].

Temporal logic is a kind of modal logic [38, 45], the logic of *necessity* and *possibility*. It is about time-dependent properties, such as causality, historical necessity, and the notions of events and actions. In temporal logic, the value of a formula depends on time. It could be `TRUE` at a given instant of time, but `FALSE` at another instant.

Temporal logics are also used to study the *temporal properties*, which depend on time instants during computation, of programs. They can express the properties, such as safety, liveness, precedence, and response, in a natural and succinct way. There are many kinds of temporal logics: linear, branching, or interval time, discrete time or dense time, etc. Gotzhein's paper [83] is a good introduction to temporal logics.

Reynolds [149] applied Lamport's Temporal Logic of Actions (TLA) [112] to GAMMA programs and developed their temporal logic proof systems. The GAMMA programs studied by Reynolds have some sorts of ordering operators on their reaction rules. They are more like the refined GAMMA programs of Chaudron and Jong [44], rather than the original GAMMA programs proposed by J.-P. Banâtre et al. [21, 22, 23].

In this thesis, we adopt Manna-Pnueli temporal logic [130], which is linear, discrete, and based on non-negative time with an original time point 0. In other words, the time domain is modeled by the set of natural numbers with its usual ordering relation, $<$ (less than).

The temporal logic is based on the first-order (predicate) logic [56, 130] with the extension of some temporal operators. Formulae from predicate logic are called *state formulae*, and so are *state predicate*, *state term* and so on. A *temporal formula* is a state formula governed by *temporal operators*. We use the following temporal operators:

$$\Box,\ \Diamond,\ \mathcal{U},\ \mathcal{W}.$$

Their informal meanings are

1. $\Box$ is *always* or *henceforth*. $\Box p$ says that p is TRUE from now on;
2. $\Diamond q$ means q will be TRUE *eventually*;
3. $p \mathcal{U} q$ means p is TRUE and holds *until* q eventually becomes TRUE; and
4. $\mathcal{W}$ is called *weak until* — $p \mathcal{W} q$ means p is TRUE and holds *until* q eventually becomes TRUE or p holds forever if q cannot become TRUE.

We do not use the next ($\bigcirc$) operator in our logic system, because it is hard to define the exact meaning of the next state as the reaction rules are all autonomous, and the execution order of the reaction rules and the time spent on a reaction rule can be arbitrary. In addition, the next operator destroys the compositionality of a logic system [109].

Example 10 (Temporal Logic Formulae) *Let p and q are formulae, we can get temporal logic formulae: $\Box p$, $\Diamond q$, $\Box\Diamond p$, $\Box p \vee \Diamond q$, $\Box(p \vee q)$, $p \mathcal{U} q$, and $p \mathcal{W} q$.* ■

A more illustrative explanation of temporal logic formulae is in Figure 7.1, where we suppose that time begins from A.D. (*Anno Domini*) year 0. The English statements above the arrow lines are state formulae; $\Diamond$ and $\Box$ are temporal logic operators. The arrow lines indicate the projection of the truth of the underlying English statements on the time axis. The thick lines or the short vertical bars mean truth; otherwise, false.

The formal semantics of a temporal formula is defined on a model σ, which is a sequence of states $\sigma = s_0 s_1 s_2 s_3 \cdots$ along the time axis, where $s_i (i \geq 0)$ denotes the state at time instant i. We use $\mathcal{M}$ to stand for the set of models, i.e., all possible sequences of states. While a state formula can be evaluated at any individual state of a sequence, a temporal formula should be evaluated on the sequences. Each state

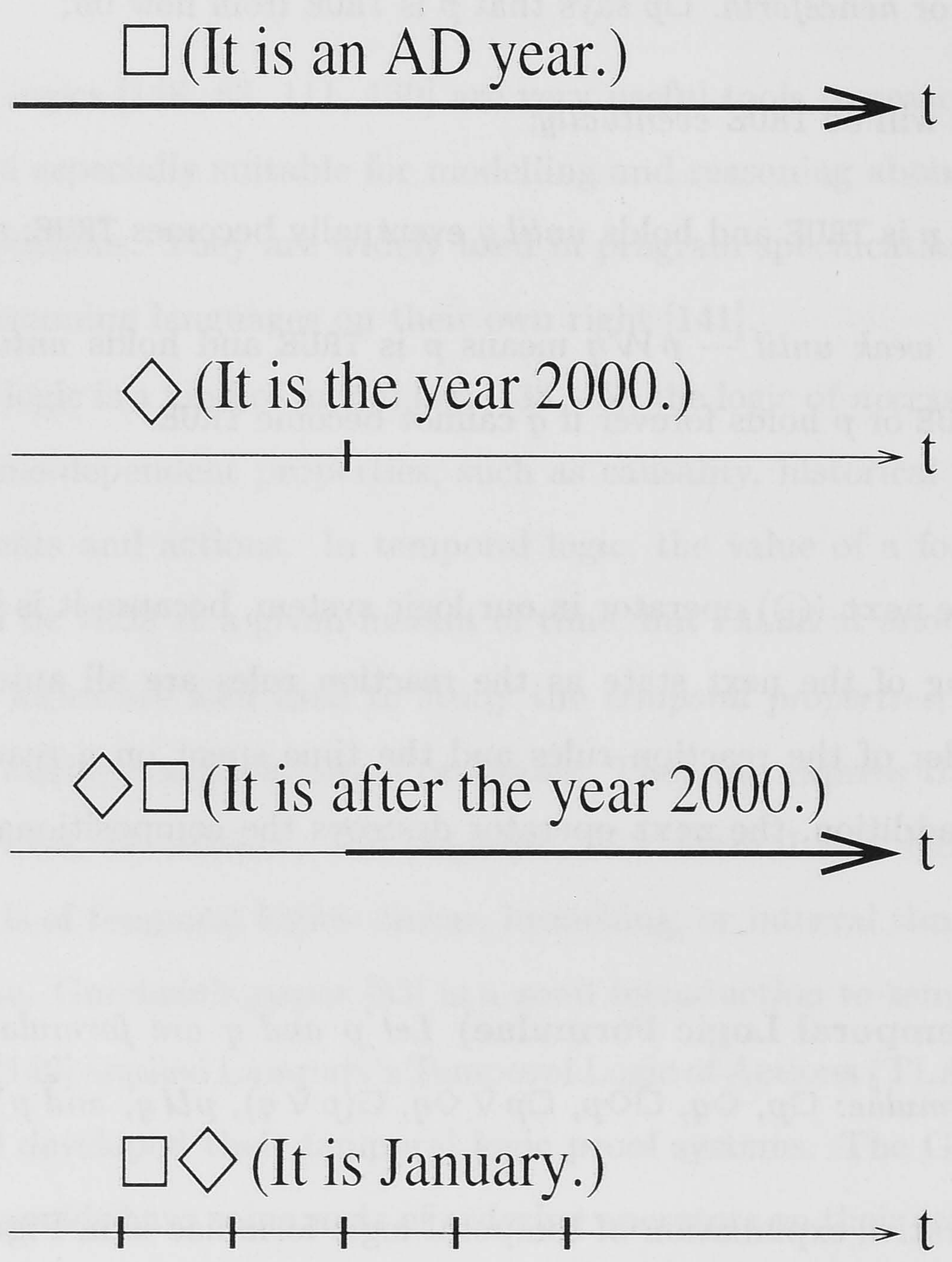


Figure 7.1: Illustrative Examples of Temporal Logical Operators

s_i can be considered as a collection of predicates which have the value TRUE at s_i, or equivalently, as a mapping between variables and boolean values.

If $(\sigma, j) \models p$, we say the model $\sigma \in \mathcal{M}$ *satisfies p at position j*; if a formula p holds at position 0 of a model σ, i.e. $(\sigma, 0) \models p$, we write $\sigma \models p$ and say that the model σ *satisfies* the formula p; and if a formula p is satisfied in every model $\sigma \in \mathcal{M}$, it is *valid* and we write $\mathcal{M} \models p$, or $\models p$ for short.

Formally, supposing $\sigma = s_0 s_1 s_2 s_3 \cdots \in \mathcal{M}$, the meaning of temporal formulae is defined as follows (read *iff* as "if and only if"):

- $(\sigma, j) \models p$ *iff* p is TRUE in s_j if p is a state formula;
- $(\sigma, j) \models \neg p$ *iff* it is not the case that $(\sigma, j) \models p$;
- $(\sigma, j) \models p \wedge q$ *iff* $(\sigma, j) \models p$ and $(\sigma, j) \models q$;
- $(\sigma, j) \models \Box p$ *iff* $(\sigma, k) \models p$ for every $k \geq j$;
- $(\sigma, j) \models \Diamond p$ *iff* $(\sigma, k) \models p$ for some $k \geq j$;
- $(\sigma, j) \models p \,\mathcal{U}\, q$ *iff* there is a $k \geq j$, such that $(\sigma, k) \models q$, and for every i, $j \leq i < k$, $(\sigma, i) \models p$;
- $(\sigma, j) \models p \,\mathcal{W}\, q$ *iff* $(\sigma, j) \models p \,\mathcal{U}\, q$ or $(\sigma, j) \models \Box p$.

The operators of $\Box$, $\Diamond$, $\mathcal{U}$, and $\mathcal{W}$ are not all independent. Actually, we can use $\mathcal{W}$ as the only primitive operator and define:

$$\Box p \equiv p \,\mathcal{W}\, \mathtt{FALSE} \qquad \Diamond p \equiv \neg \Box \neg p \qquad p \,\mathcal{U}\, q \equiv p \,\mathcal{W}\, q \wedge \Diamond q$$

A deductive system consists of a set of axioms, say $\mathcal{A}$, and a set of rewriting rules, called *inference rules*, which govern the deductive process. In other words, a deductive process is defined in purely syntactical terms. We do not deal with the issue of the *completeness* and *soundness* of a deductive system in this thesis. We refer readers to the related papers [113, 130, 148, 105]. For a temporal formula p, if we can find a sequence of rewritings from the axioms by a limited number of the applications of inference rules which leads to p (reasoning by syntax), it is called a *theorem*, and, we can also be sure that it is valid provided that the deductive system is sound, i.e., it is based on sound inference rules. We write $\mathcal{A} \vdash p$, or simply $\vdash p$, to mean that p can be proven in our deductive system from a given set of $\mathcal{A}$.

Here we list some of temporal logic axioms and inference rules. For more details, we refer readers to the corresponding papers [130, 83]. Supposing p and q are temporal logic formulae, we write:

$$p \Leftrightarrow q \quad \text{for} \quad \Box[(p \rightarrow q) \wedge (q \rightarrow p)]$$

and

$$p \Rightarrow q \quad \text{for} \quad \Box(p \rightarrow q)$$

where $p \rightarrow q$ means $\neg p \vee q$. $p \Rightarrow q$ is actually a stronger version of logic implication, which is known as *entailment*.

Some of general axioms[2] of temporal logic are:

A0	axioms and tautologies of underlying logic
A1	$\Box p \Rightarrow p$
A2	$p \Rightarrow \Diamond p$
A3	$\Box\Box p \Leftrightarrow \Box p$
A4	$\Diamond\Diamond p \Leftrightarrow \Diamond p$
A5	$\Diamond\Box\Diamond p \Leftrightarrow \Box\Diamond p$
A6	$\Box\Diamond\Box p \Leftrightarrow \Diamond\Box p$
A7	$\Box p \rightarrow p$
A8	$\Box(p \rightarrow q) \Rightarrow (\Box p \rightarrow \Box q)$
A9	$\Box p \Rightarrow p\mathcal{W}q$

Inference rules are:

1. **Generalization**(GEN): for a state formula p which does not have any temporal logic operators (i.e., it is satisfied in every state),

$$\frac{p}{\Box p}$$

[2]Some of them are theorems, which can be derived from a smaller set of axioms. For brevity, we treat them all as axioms in this thesis. The same method is applied to basic inference rules and derived inference rules.

2. **Specialization**(SPEC): for a state formula p,

$$\frac{\Box p}{p}$$

3. **Modus ponens**(MP): for any formula $p_1, \cdots, p_n$ and q,

$$\frac{(p_1 \wedge \cdots \wedge p_n) \rightarrow q,\ p_1, \cdots, p_n}{q}$$

4. **Entailment modus ponens**(EMP):for any formula $p_1, \cdots, p_n$ and q,

$$\frac{(p_1 \wedge \cdots \wedge p_n) \Rightarrow q,\ \Box p_1, \cdots, \Box p_n}{\Box q}$$

5. **Entailment transitivity**(ET): for any formula p, q, and r,

$$\frac{p \Rightarrow q, q \Rightarrow r}{p \Rightarrow r}$$

6. $\Diamond$**T**: for any formula p, q, and r,

$$\frac{p \Rightarrow \Diamond q, q \Rightarrow \Diamond r}{p \Rightarrow \Diamond r}$$

7.2 The Temporal Logic Model of T-Cham

7.2.1 From a T-Cham Program to its Temporal Logic Formulae

The logic proof system for a program consists of three parts: *the uninterpreted logic part*, *the domain part*, and *the program part* [129]. The uninterpreted logic part is the general underlying logic system, which is the temporal logic discussed in the previous section. The domain part restricts the proof system to the related domains, for example, the axioms and theorems about integers, strings, trees, etc. The program part of the proof system further restricts the proof system to the acceptable computation sequences of this program.

To represent the program part of the T-Cham proof system in temporal logic notations, we recast T-Cham programs to a 4-tuple: $P = (T, S, R, I)$[3], where

1. T is the set of all tuples (data) possibly appearing in the tuple space, for example, {`token`, `msg`} in the Producer-Consumer problem (Example 4). It is specified in the `tuples` section of a T-Cham program.

2. S is the set of all possible tuple space states. Its purpose is twofold. On the one hand, an element of S can assign values to tuples or the variables contained in the tuples. In other words, it maps variables to their domains. On the other hand, the element designates the tuples currently in some tuple space. For any tuple $t \in T$, the characteristic function $C(t) = \texttt{TRUE}$ if t is currently present in the tuple space; otherwise $C(t) = \texttt{FALSE}$. For simplicity, we write just t itself for $C(t)$ whenever there is no ambiguity. A possible tuple space state of the Producer-Consumer problem (Example 4) might be "$C(\texttt{msg}) = \texttt{TRUE} \wedge \texttt{msg} = $ `'This is a message'`".

3. R ($R \subseteq S \times S$) is the set of reaction rules. The elements of R come from `reactionrules` section.

4. I ($I \in S$): the initial state of the tuple space, which is specified by `initialization` section of a T-Cham program, for example, $I =$ {`token, token, token`}.

The logic model of a T-Cham program, written as $\mathcal{P}$, is the set of the sequences of tuple space states, which are the execution paths of the program, for any $\sigma, \sigma \in \mathcal{P}$:

$$\sigma = (s_0 s_1 s_2 s_3 \cdots).$$

A T-Cham deductive system consists of a set of axioms and a set of inference rules.

Axioms come from the general axioms of the uninterpreted temporal logic, domain axioms (including fairness properties etc.), and the axioms from a given T-Cham program [130, pp. 255–258][129].

For any reaction rule, say,

$$x_1, x_2, \cdots, x_n \;\texttt{leadsto}\; y_1, y_2, \cdots, y_m \texttt{by}\; T \;\texttt{when}\; f(x_1, x_2, \cdots, x_n), \tag{7.1}$$

[3]In order to avoid attacking all complex issues in a single step, we ignore the impact of race among T-Cham reactions and T-Cham termination conditions on the proof system in this section and will discuss them later.

supposing the pre-condition and post-condition of T are p and q, i.e., $\{p\}T\{q\}$, and there are no repetitive elements among $x_1, x_2, \cdots, x_n, y_1, y_2, \cdots$, and y_m, we have:

$$[[\bigwedge_{i=1}^{n}(|x_i| \geq 1)] \wedge f(x_1, x_2, \cdots, x_n) \wedge p] \Rightarrow \Diamond[(\bigwedge_{i=1}^{n}|x_i|^{-1}) \wedge (\bigwedge_{j=1}^{m}|y_j|^{+1}) \wedge q], \quad (7.2)$$

where $|x|$ denotes the number of x tuples currently in the tuple space, and $|x|^{-t}$ means the number of tuple x has decreased by t in comparison to the former state (i.e., the state corresponding to the left hand side of the reaction rules), while $|x|^{+t}$ means that the population of tuple x is increased by t. If there is no multiplicity of occurrences of a same type tuple and *no confusion* based on the context, Formula 7.2 can be written simply as:

$$[\bigwedge_{i=1}^{n} x_i \wedge f(x_1, x_2, \cdots, x_n) \wedge p] \Rightarrow \Diamond[\bigwedge_{i=1}^{n} \neg x_i \wedge \bigwedge_{j=1}^{m} y_j \wedge q], \quad (7.3)$$

where x_i or y_j means the tuple is current in tuple space, while $\neg x_i$ or $\neg y_j$ means it is not in the tuple space.

To deal with repetitive elements, we define x-type tuples and y-type tuples.

Definition 1 (x-type tuples, x-type multiset, y-type tuples, and y-type multiset) *In a reaction rule of T-Cham such as*

$x_1 + x_2 + \cdots + x_n$ `leadsto` $y_1 + y_2 + \cdots + y_m$ `by` T `when` $f(x_1, x_2, \cdots, x_n)$,

the tuples which appear on the left hand side of `leadsto` *are called the* x-type tuples *of the reaction rule. They may repetitively appear on the right hand side of* `leadsto`. *The multiset* $\{x_1, x_2, \cdots, x_n\}$ *is called the* x-type multiset *of the rule, because it is composed by x-type tuples. The tuples which appear on the right hand side of a reaction rule and do not appear on the left hand side are called the* y-type tuples *of the rule. The multiset* $\{y_1, y_2, \cdots, y_m\} - \{x_1, x_2, \cdots, x_n\}$ *is called the* y-type multiset *of the rule.* ∎

Definition 2 (Representative Set) *A* representative set *of a multiset* $\mathbf{M}$ *is the set obtained from* $\mathbf{M}$ *by eliminating repetitive elements. We write* $\tilde{\mathbf{M}}$ *as the representative set of* $\mathbf{M}$. ∎

Supposing $\tilde{x}_1$, $\tilde{x}_2$, $\cdots$, and $\tilde{x}_{\tilde{n}}$ are the elements of the representative set of x-type multiset; $\tilde{y}_1$, $\tilde{y}_2$, $\cdots$, and $\tilde{y}_{\tilde{m}}$ are of y-type's; and the repetition number of $\tilde{x}_i$ in the x-type multiset is r_i and $\tilde{y}_i$ is s_i. The more general form of the formula is:

$$[[\bigwedge_{i=1}^{\tilde{n}}(|\tilde{x}_i| \geq r_i)] \wedge f(x_1, x_2, \cdots, x_n) \wedge p] \Rightarrow \Diamond[(\bigwedge_{i=1}^{\tilde{n}}|\tilde{x}_i|^{-r_i+t_i}) \wedge (\bigwedge_{j=1}^{\tilde{m}}|\tilde{y}_j|^{+s_j}) \wedge q], \quad (7.4)$$

where t_i is the number of x-type tuples which appear on the right hand side of the reaction rule. For example, if we have `x+x+x leadsto x+y`, then $r_x = 3$ and $t_x = 1$.

7.2.2 Stuttering

If there is a reaction rule r which makes the state s_i transmit to the state s_j, i.e.,

$$s_i \xrightarrow{r} s_j,$$

only the values and populations of the x-type tuples and the y-type tuples of the reaction rule are changed from s_i to s_j. All the other tuples remain unaffected. To express this property, Formula 7.4 has to be rewritten.

Supposing $\mathbf{S}$ is the multiset of tuples in the current tuple space, and $\mathbf{X}$ is the multiset of the x-type tuples in a reaction rule.

$$\mathbf{Z} = \mathbf{S} - \mathbf{X}$$

is the multiset of the tuples which are not affected by the reaction rule. Similarly, the representative set of $\mathbf{Z}$ is $\tilde{\mathbf{Z}}$. If there are $\tilde{u}$ elements in $\tilde{\mathbf{Z}}$, we use $\tilde{z}_1$, $\tilde{z}_2$, $\cdots$, and $\tilde{z}_{\tilde{u}}$ to stand for each element. Formula 7.4 now is:

$$[[\bigwedge_{i=1}^{\tilde{n}}(|\tilde{x}_i| \geq r_i)] \wedge f(x_1, x_2, \cdots, x_n) \wedge p \wedge [\bigwedge_{k=1}^{\tilde{u}}(|\tilde{z}_k| = u_k)]] \Rightarrow$$
$$\Diamond[(\bigwedge_{i=1}^{\tilde{n}}|\tilde{x}_i|^{-r_i+t_i}) \wedge (\bigwedge_{j=1}^{\tilde{m}}|\tilde{y}_j|^{+s_j}) \wedge q \wedge [\bigwedge_{k=1}^{\tilde{u}}(|\tilde{z}_k| = u_k)]],$$

This formula schema affects all the translations from T-Cham programs to their corresponding temporal logic formulas, but the $\tilde{z}_k$ part, i.e., the unchanging tuples part, is omitted in the rest of the chapter whenever there is no confusion.

If p holds at s_i, and q holds at s_j, we have:

$$p \Rightarrow \Diamond q.$$

The transition r may consist of some sub-transitions r_1, r_2, $\cdots$, and r_n:

$$s_i \xrightarrow{r_1} s_{i_1} \xrightarrow{r_2} s_{i_2} \xrightarrow{r_3} \cdots \xrightarrow{r_n} s_j.$$

We do not have to worry about the impact of r_1, r_2, $\cdots$, and r_n because the transition r is closed under stuttering [109]. In other words, if s_i and s_j are in the execution sequence (σ), we will always have

$$p \Rightarrow \Diamond q.$$

In addition, if a termination state, say s_n, is reached, e.g., in a termination program whose termination condition is met, we simply assume that the state s_n will be transmitted back to itself indefinitely, i.e.,

$$\sigma = (s_0 s_1 s_2 \cdots s_{n-1} s_n s_n s_n \cdots).$$

7.3 The Impact of Race between Reaction Rules

7.3.1 The Problem

If a reaction rule deprives the execution of other rule(s), a race happens, for example (we refer the program as **P** in the subsequent paragraphs),

```
initialization
        x, y, z;
reactionrules
        x, y leadsto a;
        y, z leadsto b;
```

The program **P** may end up with either $\{\!|$ a, z $|\!\}$ or $\{\!|$ b, x $|\!\}$, if the initial tuple space is $\{\!|$ x, y, z $|\!\}$, depending on which reaction rule is fired. Some other possible execution paths are shown in Figure 7.2.

We say there is a race between two reaction rules, if at any stage, one reaction rule can *permanently* prevent the other from firing. In this program, whenever the tuple

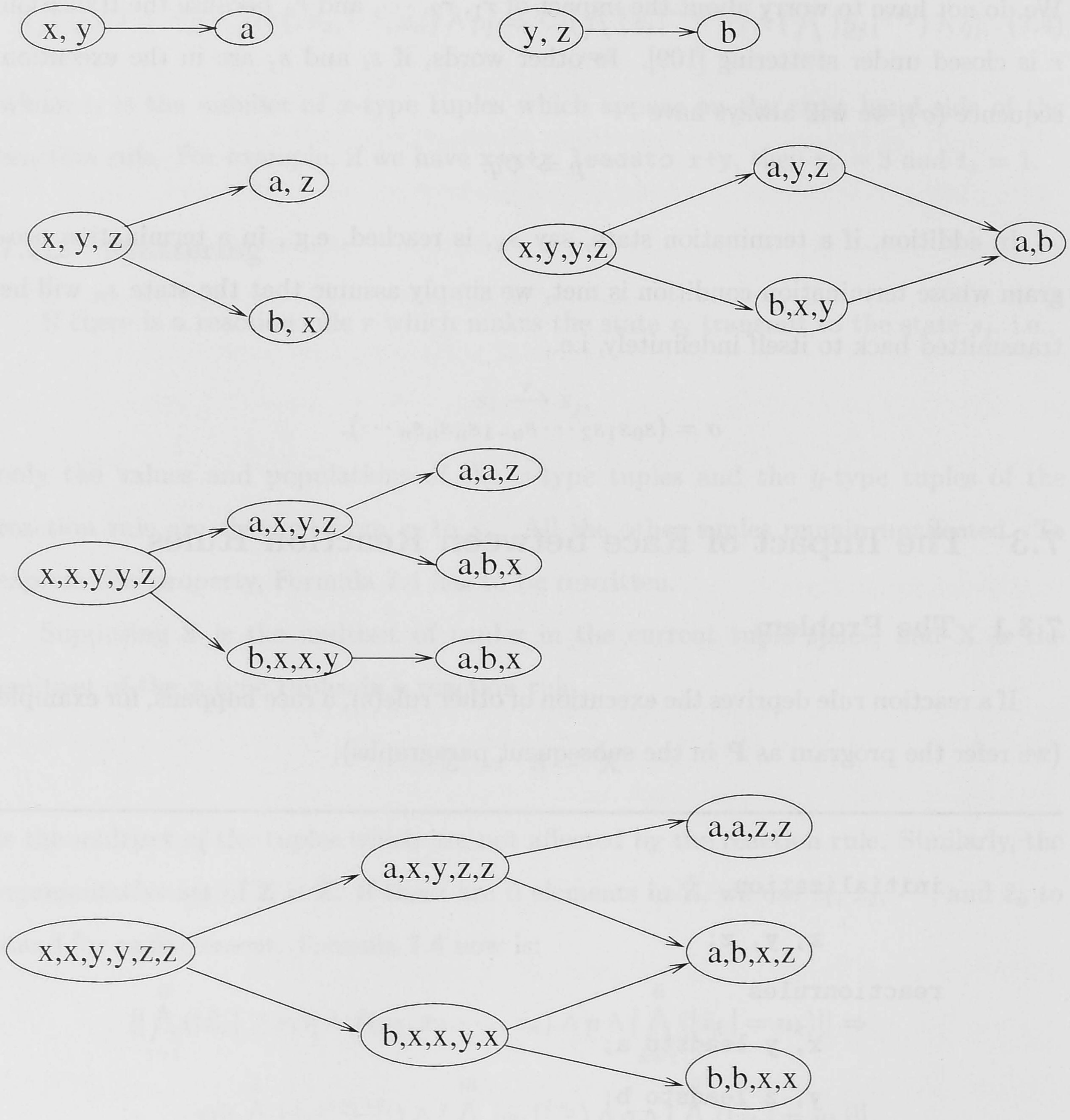


Figure 7.2: Different Execution Paths

space goes into $\{\!|\ \mathsf{x},\ \mathsf{y},\ \mathsf{z}\ |\!\}$ state, either of the two reaction rules of **P** can fire, but not both. After the firing, the program reaches its termination point, because no reaction rule can fire according to the new tuple space state, either $\{\!|\ \mathsf{a},\ \mathsf{z}\ |\!\}$ or $\{\!|\ \mathsf{b},\ \mathsf{x}\ |\!\}$.

If we translate the program **P** into temporal logic formulae by the method developed in Section 7.2.1, we will have:

$$|\mathsf{x}| \geq 1 \wedge |\mathsf{y}| \geq 1 \ \Rightarrow\ \diamond(|\mathsf{a}|^{+1} \wedge |\mathsf{x}|^{-1} \wedge |\mathsf{y}|^{-1}) \tag{7.5}$$

$$|\mathsf{y}| \geq 1 \wedge |\mathsf{z}| \geq 1 \ \Rightarrow\ \diamond(|\mathsf{b}|^{+1} \wedge |\mathsf{y}|^{-1} \wedge |\mathsf{z}|^{-1}) \tag{7.6}$$

If the initial tuple space, written as $\mathcal{I}$, is $\{\!|\ \mathsf{x},\ \mathsf{y},\ \mathsf{z}\ |\!\}$, we will have:

$$\diamond(|\mathsf{a}| = 1 \wedge |\mathsf{b}| = 1)$$

which is obviously wrong. The culprit here is the race between the two reaction rules of the program **P**: any of those two reaction rules, if fired, can prevent the other from firing. In other words, if one rule is fired, the other will never have a chance. It is equivalent to say, under this particular circumstance, that the reaction rule, which does not have the chance to fire, does not exist in the program **P**. But on the other hand, the temporal logic formulae (7.5) and (7.6) are true under all possible circumstances, regardless of initial tuple space states or the execution paths. We cannot say that formula (7.5) holds under certain initial tuple space states and execution paths, while formula (7.6) holds under some other circumstances. In short, the two temporal logic formulae do not correspond to the program **P**.

Before we proceed any further to find out a remedy to the problem, let's have a careful look at the program **P** and its execution paths under different initial tuple space states as well as their temporal logic significance. As temporal logic is about the safety and liveness properties of programs [109, 110], what we can (and can only) expect by formalising a program with temporal logic is to prove that:

> *Something bad will never happen (the safety properties), and*
> *something good will eventually happen (the liveness properties).*

Both the safety and liveness properties should be valid under *any initial conditions* and *any execution paths*. In this program **P**, if we have the initial tuple space of

$\{\!|$ x, y, z $|\!\}$, the program will stop at either $\{\!|$ a, z $|\!\}$ or $\{\!|$ b, x $|\!\}$. If $\{\!|$ a, z $|\!\}$ is the "something good", we can never guarantee it will happen; neither does $\{\!|$ b, x $|\!\}$. The same observation is true under different initial tuple space states, Figure 7.2. The only sensible property which is true under all possible initial tuple space states and execution paths of **P** is:

$$\Diamond\Box(|\mathtt{y}| = 0), \text{ i.e., tuple } \mathtt{y} \text{ will eventually be used up.}$$

It might be "something good" and also can be guaranteed to happen, but we really do not think it provides much useful information about the program **P**.

Due to the unpredictable outcomes of the program **P**, temporal logic fails to prove any useful properties out of it. Although, in this thesis, we are not going to argue if the program **P** (and all the other possible programs of its class) is sensible, the existence of this kind of programs does raise a problem for us. We have to find out to what extent the temporal logic proof system developed in Section 7.2.1 can apply.

As we said before, the reason for the failure of the temporal logic formulae (7.5) and (7.6) is due to the race between the two reaction rules of the T-Cham program **P**. If we investigate a bit further into the two reaction rules, we find out that it is the tuple y which causes the race. The number of tuple y in the program **P**, no matter what it is in the initial tuple space state, is limited. Either of the reaction rules, when firing, reduces the number of tuple y by one. Eventually, it will reach the critical state that only one tuple y is in the tuple space. Whenever it comes into that state, the two reaction rules have to race against each other for tuple y, but only one reaction rule can succeed. This is the point where the temporal logic formulae (7.5) and (7.6) fail, because the two formulae have to be true over every possible execution path according to temporal logic semantics. If the number of tuple y were unlimited, i.e., tuple y would never be used up, the two reaction rules would not have to race for it—there always are enough tuple y ready for the two the reaction rules to use. In reality, there is no way to include the unlimited number of tuple y into the initial tuple space of any T-Cham program, but two possibilities exist which make tuple y never be used up. It is just equivalent to say that the number of tuple y in the tuple space is unlimited. The first possibility is that the tuple y can always be reproduced after being consumed:

```
reactionrules
        x, y leadsto a, y;
        y, z leadsto b, y;
```

The other possibility is that some other reaction rules in the program will indirectly reproduce the tuple y: whenever a tuple y is consumed, another tuple y will be re-generated, for example, if the T-Cham program **P** has another two reaction rules (the new program is referred as $\mathbf{P}'$)[4]:

```
reactionrules
        x, y leadsto a;
        y, z leadsto b;
        a leadsto y;
        b leadsto y;
```

The temporal logic formulae (7.5) and (7.6) hold under every possible initial tuple space state and execution path of the program $\mathbf{P}'$. In addition, the other two temporal logic formulae:

$$|\mathtt{a}| \geq 1 \;\Rightarrow\; \Diamond(|\mathtt{y}|^{+1} \wedge |\mathtt{a}|^{-1}) \tag{7.7}$$

$$|\mathtt{b}| \geq 1 \;\Rightarrow\; \Diamond(|\mathtt{y}|^{+1} \wedge |\mathtt{b}|^{-1}) \tag{7.8}$$

also hold under every possible initial tuple space state and execution path of the program $\mathbf{P}'$. The four temporal logic formulae (7.5), (7.6), (7.7), and (7.8) are the proper temporal logic formulation translation of the reaction rules of program $\mathbf{P}'$.

In short, if a T-Cham program is not under a race condition, the temporal logic system developed in Section 7.2.1 applies; if the program is under a race condition, but all those tuples which cause the race can always be re-produced after being consumed, the temporal logic system can still be applied; otherwise, the temporal logic system does not apply.

[4]We are not talking about the sensible meaning of the program. This artificial program only serves to find out when we still can apply the temporal logic formula translation methods developed in Section 7.2.1 upon the existence of race condition. Later on in this section, we discuss the Dining Philosophers problem, which is a meaningful program.

7.3.2 The Solution

Following the previous observation on the race problems of the two T-Cham program examples and their impact on the temporal logic proof system, here we discuss, in a more general sense, how to find out if a T-Cham program, which is under race conditions, still can enjoy a proper temporal logic proof system.

Definition 3 (Maximum Common Tuple Group, MCTG) *If the left hand sides of any two reaction rules of a T-Cham program have some tuples in common, we call any of those tuples as a* common tuple *of the two reaction rules. We call a number of the common tuples together* common tuple group, *and all the common tuples of the two reaction rules* maximum common tuple group *or* MCTG. ∎

For example, if we have two reaction rules:

$$\alpha,\ \mathtt{x},\ \beta\ \ \mathtt{leadsto}\ \ A; \qquad (7.9)$$

$$\beta,\ \mathtt{x},\ \gamma\ \ \mathtt{leadsto}\ \ B; \qquad (7.10)$$

where α, β, and γ consist of the tuples from a T-Cham program, and there is no single common tuple between α and γ. The tuple `x` is one of the common tuples of the two reaction rules, while tuple group β or any part of β is common tuple group. As there is no single common tuple between α and γ, tuple `x` and β together is the maximum common tuple group (or MCTG) of the two reaction rules.

Definition 4 (Race Condition) *If any two reaction rules of a T-Cham program have at least one common tuple, we say the two reaction rules are under* race condition. *If any two reaction rules of a T-Cham program are under race condition, we say the program itself is also under* race condition. ∎

If two reaction rules of a T-Cham program are under race condition, they may not necessarily race against each other for the common tuples. Only when it reaches at a state where the number of common tuples is not big enough—at least, more than two copies of MCTG—to meet the requirement of the both reaction rules, a race happens: either of the reaction rules, if fired, will prevent the other from firing.

Definition 5 (Race) *A* race *between two reaction rules of a T-Cham program happens if at any stage along any execution path of the program, one of the reaction rule, if fired,* permanently *prevents the other from firing. If a race between any two reaction rules of a T-Cham program happens, we say the* race *happens in program, or the program has* race *action.* ■

Race condition is different from race. A program is under race condition does not mean the program will absolutely develop into race actions. Both of the program **P** and **P**′ discussed before are under race condition. Program **P** has race actions, but program **P**′ doesn't.

Definition 6 (No Race Under Race Condition, NRURC) *If two reaction rules of a T-Cham program are under race condition, but under no circumstance does a race happens between the two reaction rules, we call the situation as* no race under race condition *or* NRURC. *Similarly, if a T-Cham program is under race condition but no race actions, it is called* no race under race condition *or* NRURC *for the program.* ■

If a T-Cham program is not under any race condition, it is obvious that the temporal logic proof system developed in Section 7.2.1 can be applied to the program without any problem. On the other hand, if a T-Cham program is under some race conditions, as long as the program keeps in NRURC situation, the proof system developed in Section 7.2.1 still applies, because NRURC means that no reaction rule will be deprived from firing by any other reaction rules. In other words, at any stage of any execution path, whenever the condition for a raction rule to fire is true, the reaction rule will fire, and the tuples at its left hand side will be transformed into the tuples at its right hand side. This is just what the temporal logic formula requires. Finally, if a T-Cham program does have race actions, and it will race, the temporal logic proof system does not apply.

In summary, if a T-Cham program is under race condition but keeps in NRURC situation, the temporal logic proof system developed in Section 7.2.1 can still be used to reason the temporal properties of the program.

The conclusion brings us two new questions: how to decide if a program under race condition is NRURC and is there any useful program in this NRURC group?

7.3.3 How to Decide No Race Under Race Condition

If two reaction rules are under race condition, a race happens when the tuple space has only one copy of MCTG and no more MCTG after that MCTG is consumed by either of the reaction rules. No race under race condition (NRURC) is possible only if the number of every tuple in MCTG is unlimited. There are two possibilities where the number of a tuple can be considered as unlimited. The first possibility is that the tuples are guarded by the operator "!", i.e., they won't be consumed by the reaction rule. The other possibility is that the tuples will be re-produced after being consumed by the reaction rule.

To guarantee that the tuples will always be successfully re-produced, some conditions have to be met. Suppose we have a T-Cham program **R**, and **r** is one of its reaction rules. Tuple `t` is one of the tuples in the left hand side of reaction rule **r**, and it belongs to a MCTG, i.e., reaction rule **r** is under race condition with some other reaction rules. To check if tuple `t` can be re-produced under any circumstance, we set up a tuple space with exactly the tuples required by the left hand side of the reaction rule **r** as the initial tuple space of program **R**, and then simulate the execution of the program. During the simulation, any reaction rule which is under race condition is excluded. If the simulation finally produces a copy of tuple `t`, we will know that tuple `t` can always be re-produced. The simulation procedure can be terminated within a finite number of steps if we keep the tracks of execution paths and check any possible reaction loops, i.e., the current tuple space state is the same as the other previous tuple space state along this execution path. When a reaction loop happens, we stop simulating the reactions on this loop. The algorithm for the simulation is very simple. Manually, we can draw a graph to do the simulation. In the graph, tuples are written as they are except that the one we want to check if it can be re-produced is in a circle. A bar means a reaction. The arrows from some tuples to a bar indicate that those tuples will be consumed by the reaction, while the arrows from a bar to some tuples indicate that those tuples will be generated by the reaction. Some example graphs are in Figure 7.3.

To decide if a tuple can be guaranteed to be re-produced after being consumed,

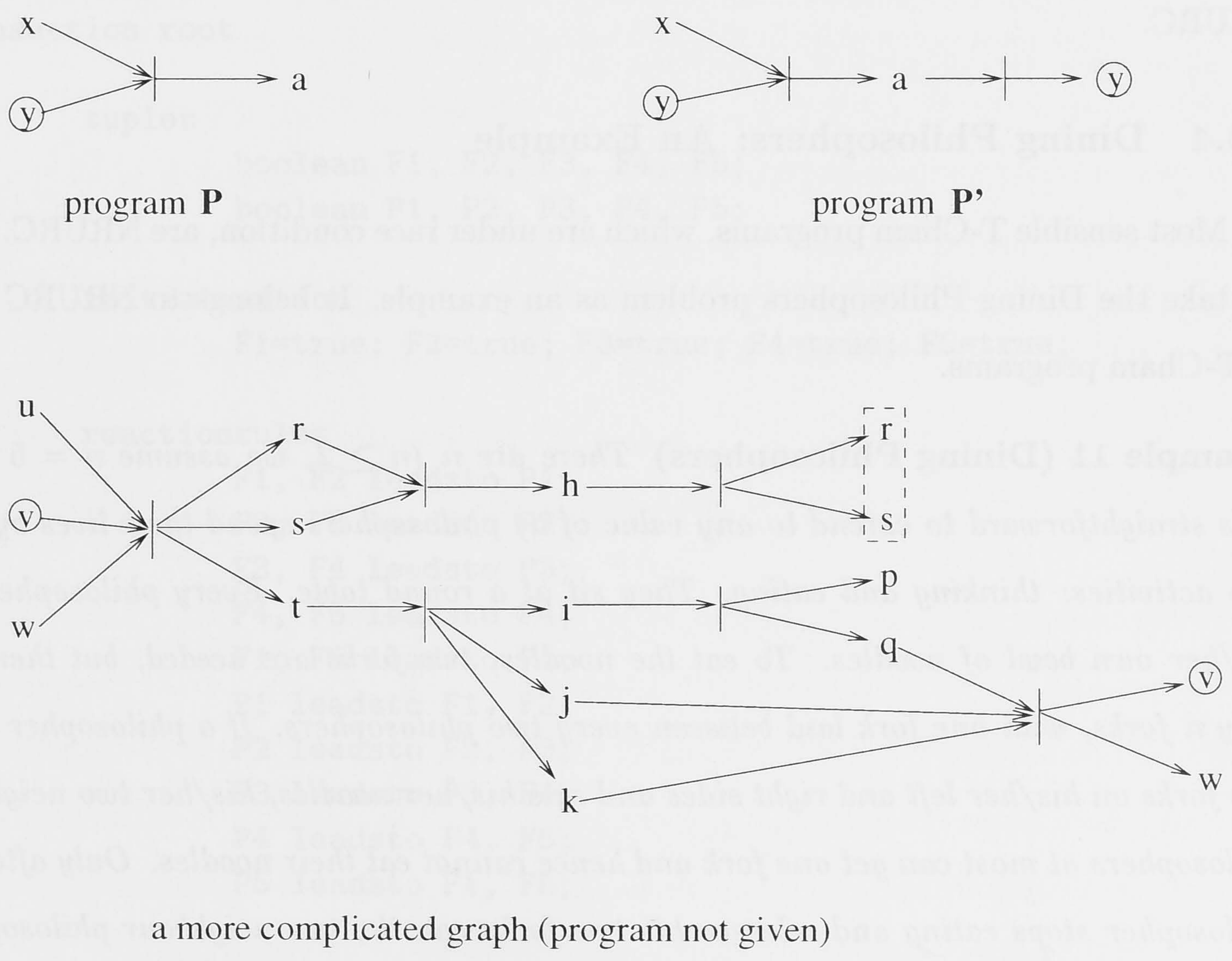


Figure 7.3: The Graphs for Tuple Re-producing Checking

every reaction rule which has the tuple at its left hand side has to be checked by the simulation procedure discussed before. The tuple y in the T-Cham program **P** of the previous section cannot be guaranteed to be re-produced, but the tuple y in the program **P′** can.

If a T-Cham program is under some race conditions, but every single tuple of any MCTG between any two reaction rules—which are under a race condition—of the program can be guaranteed to be re-produced after being consumed, the program is NRURC.

7.3.4 Dining Philosophers: An Example

Most sensible T-Cham programs, which are under race condition, are NRURC. Here we take the Dining Philosophers problem as an example. It belongs to NRURC class of T-Cham programs.

Example 11 (Dining Philosophers) *There are n ($n \geq 2$, we assume $n = 5$ here. It is straightforward to extend to any value of n) philosophers spend their lives by only two activities: thinking and eating. They sit at a round table. Every philosopher has his/her own bowl of noodles. To eat the noodles, two forks are needed, but there are only n forks, with one fork laid between every two philosophers. If a philosopher takes two forks on his/her left and right sides and eats his/her noodles, his/her two neighbour philosophers at most can get one fork and hence cannot eat their noodles. Only after the philosopher stops eating and releases his two forks can the two neighbour philosophers have the possibility to eat.* ∎

In the T-Cham program (Figure 7.4), we use tuples `F1` to `F5` to stand for the five forks and tuple `P1` to `P5` for the five philosophers. We assume that forks in tuple space means they are available, and philosophers in the tuple space mean they are eating.

It is very straightforward to verify that the T-Cham program is NRURC. According to Section 7.2.1, the first reaction rule of the program can be translated into:

$$|\mathtt{F1}| \geq 1 \wedge |\mathtt{F2}| \geq 1 \Rightarrow \Diamond(|\mathtt{P1}|^{+1} \wedge |\mathtt{F1}|^{-1} \wedge |\mathtt{F2}|^{-1})$$

Given the initial tuple space state and the reaction rules, the number of any tuples in the tuple space is either 1 or 0 (it is very easy to prove). In addition, the temporal

```
transaction root

        tuples
                boolean F1, F2, F3, F4, F5;
                boolean P1, P2, P3, P4, P5;

        initialization
                F1=true; F2=true; F3=true; F4=true; F5=true;

        reactionrules
                F1, F2 leadsto P1;
                F2, F3 leadsto P2;
                F3, F4 leadsto P3;
                F4, F5 leadsto P4;
                F1, F5 leadsto P4;
                P1 leadsto F1, F2;
                P2 leadsto F2, F3;
                P3 leadsto F3, F4;
                P4 leadsto F4, F5;
                P5 leadsto F1, F5;

endtrans
```

Figure 7.4: The T-Cham program of Dining Philosophers

logic formulae for the first five reaction rules are very similar, and so are the second five. We write down the temporal logic formulae for the program in a compact way[5]:

$$|\mathtt{F}_i| = 1 \ \wedge \ |\mathtt{F}_{i\oplus 1}| = 1 \quad \Rightarrow \quad \Diamond(|\mathtt{F}_i| = 0 \ \wedge \ |\mathtt{F}_{i\oplus 1}| = 0 \ \wedge \ |\mathtt{P}_i| = 1) \qquad \text{(ph.f)}$$

$$|\mathtt{P}_i| = 1 \quad \Rightarrow \quad \Diamond(|\mathtt{F}_i| = 1 \ \wedge \ |\mathtt{F}_{i\oplus 1}| = 1 \ \wedge \ |\mathtt{P}_i| = 0) \qquad \text{(ph.p)}$$

There is no deadlock in this program, because the condition testing is atomic. One liveness property of the problem is *free of starvation*, which says that a philosopher who is thinking (not eating, and hence will get hungry) will eventually gets the forks and eats, and any philosopher who is eating will eventually stop eating to think. In temporal logic, the properties are written as:

$$|\mathtt{P}_i| = 0 \Rightarrow \Diamond(|\mathtt{P}_i| = 1) \quad \text{and} \quad |\mathtt{P}_i| = 1 \Rightarrow \Diamond(|\mathtt{P}_i| = 0)$$

The proof is trivial given the temporal logic formula translation—Formuale (ph.f) and (ph.p) —of the program (Figure 7.4) and the fairness assumption of T-Cham.

7.4 T-Cham Program Verification

Safety and liveness are two fundamental temporal properties [109]. Safety ensures that something bad never happens ($\Box p$ in temporal logic formula), while liveness guarantees that something good will eventually happen ($\Diamond q$ or $p \rightarrow \Diamond q$). A specification (or program) disciplines the behavior of a computation. The behavior is actually a safety property of a computation. In other words, safety properties are expressed by T-Cham programs themselves, which will be translated into a set of temporal logic formulae. Liveness is not always given out by a specification. Temporal logic is very useful to prove liveness properties from the corresponding safety properties [111].

7.4.1 The Producer-Consumer Problem

To prove the correctness of Producer-Consumer (Example 4), we first convert its T-Cham program in Figure 5.4 to temporal logic formulae, which will be used as additional axioms of the "producer-consumer" deductive system. The axioms, referred as $\mathcal{A}_{pc}$, are:

[5] $1 \leq i \leq 5$ and $i \oplus 1$ means $(i+1)$ *mod* 5.

$$(\sigma, 0) \models (|\mathtt{token}| = n) \wedge (|\mathtt{msg}| = 0)$$
$$|\mathtt{token}| > 0 \Rightarrow \Diamond(|\mathtt{token}|^{-1} \wedge |\mathtt{msg}|^{+1})$$
$$|\mathtt{msg}| > 0 \Rightarrow \Diamond(|\mathtt{msg}|^{-1} \wedge |\mathtt{token}|^{+1})$$

For the sake of brevity, we user variable *prod* to denote the producing action and *cons* to denote the consuming action. Thus, $\Diamond prod$ assert there will be a producing action, i.e., transaction `prod` will be committed, and $\Diamond cons$ stands for a consuming action in the future.

Two main properties of the producer-consumer problem are *reactivity*, which means there always are producing or consuming actions, $\Box(\Diamond prod \vee \Diamond cons)$, and *progress* — every produced message (by transaction `prod`) will be eventually consumed (by transaction `cons`), that is, $prod \Rightarrow \Diamond cons$. The two properties can certainly be represented by assertions on tuples `token` and `msg`, but it is natural to reason on the actions of *prod* and *cons*.

$$|\mathtt{token}| > 0 \Rightarrow \Diamond(|\mathtt{token}|^{-1} \wedge |\mathtt{msg}|^{+1})$$

thus is written as:

$$|\mathtt{token}| > 0 \Rightarrow \Diamond prod \text{ and } prod \Rightarrow \Diamond(|\mathtt{token}|^{-1} \wedge |\mathtt{msg}|^{+1})$$

Besides, from $\mathcal{A}_{pc}$, we can also have "*the total number of* `tokens` *and* `msgs` *is n*," i.e.,

$$\mathcal{A}_{pc} \vdash \Box(|\mathtt{Token}| + |\mathtt{Msg}| = n),$$

which is also called an *axiom* in the following proof for brevity. Thus, we have the new set of axioms[6]:

$$\Box(|\mathtt{Token}| + |\mathtt{Msg}| = n) \qquad (\text{pc}.1)$$
$$(|\mathtt{Token}| + |\mathtt{Msg}| = n) \Rightarrow \Diamond prod \vee \Diamond cons \qquad (\text{pc}.2)$$
$$prod \Rightarrow \Diamond(|\mathtt{Msg}| > 0) \qquad (\text{pc}.3)$$
$$cons \Rightarrow \Diamond(|\mathtt{Token}| > 0) \qquad (\text{pc}.4)$$
$$(|\mathtt{Msg}| > 0) \Rightarrow \Diamond cons \qquad (\text{pc}.5)$$
$$(|\mathtt{Token}| > 0) \Rightarrow \Diamond prod \qquad (\text{pc}.6)$$

[6]We can obtain (pc.2) by a simple calculation:

$$(|\mathtt{Token}| + |\mathtt{Msg}| = n) \Rightarrow (|\mathtt{Token}| > 0 \vee |\mathtt{Msg}| > 0) \Rightarrow \Diamond prod \vee \Diamond cons.$$

Theorem 1 (reactivity of producer-consumer) *There will always be prod or cons actions, i.e.,* $\Box(\Diamond prod \vee \Diamond cons)$.

Proof:

1	$\Box(\lvert\texttt{Token}\rvert + \lvert\texttt{Msg}\rvert = n)$	(pc.1)
2	$(\lvert\texttt{Token}\rvert + \lvert\texttt{Msg}\rvert = n) \Rightarrow \Diamond prod \vee \Diamond cons$	(pc.2)
3	$\Box(\Diamond prod \vee \Diamond cons)$	EMP, 1, 2

■

Theorem 2 (liveness of producer-consumer) *Every produced message will be eventually consumed, or in other words, every prod action will be followed by a cons action.* $prod \Rightarrow \Diamond cons$.

Proof:

1	$prod \Rightarrow \Diamond(\lvert\texttt{Msg}\rvert > 0)$	(pc.3)
2	$(\lvert\texttt{Msg}\rvert > 0) \Rightarrow \Diamond cons$	(pc.5)
3	$prod \Rightarrow \Diamond cons$	$\Diamond$T, 1, 2

■

7.4.2 The Dutch Flag Problem

The Dutch flag problem of Example 5 shows the properties of a terminating program and the use of universal quantifier ($\forall$).

The temporal logic formulae of the program have the so-called race problem, but we still can verify the program by the simple solution discussed in Section 7.3. As we do not have to rely on $p \rightarrow p \vee p$ or $p \rightarrow p \wedge p$ to prove any theorem (see Section 7.3 for detailed discussion), this particular race problem has no impact on the underlying temporal logic system.

Recall the rules of T-Cham program in Figure 5.5, the temporal logic formula we get has the form of[7]:

$$\forall x, y, 1 \leq x, y \leq \texttt{n}: \quad (x, \texttt{r}) \wedge (y, \texttt{w}) \wedge x > y \Rightarrow \Diamond[(x, \texttt{r}) \wedge (y, \texttt{w}) \wedge x < y],$$

Please note the x and y in both sides of the formula may have different values, because in the underlying temporal logic, values of terms are not required to be *rigid*. The

[7]For brevity, we write $(x,\texttt{r})$ instead of $\texttt{strip}(x,\texttt{r})$.

new values, on the right hand side, are obtained by swapping the old values of x and y. So we have "$x > y$" on the left hand side of the formula while "$x < y$" on the right. Following the tradition and also for brevity, we drop the universal quantifier $\forall$ whenever there is no confusion. Thus the above formula becomes:

$$(x,\texttt{r}) \land (y,\texttt{w}) \land x > y \Rightarrow \Diamond[(x,\texttt{r}) \land (y,\texttt{w}) \land x < y] \qquad \text{(df.1)}$$

Theorem 3 (Dutch Flag) *When the program terminates (no rules are applicable any further), we expect that all* `Red` *elements come first, then* `White` *ones, and* `Blue` *ones come last. The properties can be described by the following temporal logic formulae:*

1. $\Box\Diamond[(x,\texttt{r}) \land (y,\texttt{w}) \rightarrow x < y]$;

2. $\Box\Diamond[(x,\texttt{w}) \land (y,\texttt{b}) \rightarrow x < y]$;

3. $\Box\Diamond[(x,\texttt{r}) \land (y,\texttt{b}) \rightarrow x < y]$.

Proof:

Consider the first property, $\Box\Diamond[(x,\texttt{r}) \land (y,\texttt{w}) \rightarrow x < y]$. It can be proved by case analysis, for any two elements of $(x,\texttt{r})$ and $(y,\texttt{w})$:

- $x < y$: $\Box\Diamond[(x,\texttt{r}) \land (y,\texttt{w}) \rightarrow x < y]$ holds trivially;

1	$(x,\texttt{r}) \land (y,\texttt{w}) \land x > y$	given;
2	$(x,\texttt{r}) \land (y,\texttt{w}) \land x > y \Rightarrow \Diamond[(x,\texttt{r}) \land (y,\texttt{w}) \land x < y]$	df.1
3	$\Box[(x,\texttt{r}) \land (y,\texttt{w}) \land x > y \rightarrow \Diamond[(x,\texttt{r}) \land (y,\texttt{w}) \land x < y]]$	def. of $\Rightarrow$, 2

- $x > y$:

4	$(x,\texttt{r}) \land (y,\texttt{w}) \land x > y \rightarrow \Diamond[(x,\texttt{r}) \land (y,\texttt{w}) \land x < y]$	SPEC, 3
5	$\Diamond[(x,\texttt{r}) \land (y,\texttt{w}) \land x < y]$	MP, 1, 4
6	$\Box\Diamond[(x,\texttt{r}) \land (y,\texttt{w}) \land x < y]$	GEN, 5
7	$\Box\Diamond[(x,\texttt{r}) \land (y,\texttt{w}) \rightarrow x < y]$	weakening, 6

The other two properties can be proved in a similar fashion. ∎

7.4.3 The Meeting Scheduler Problem

Meeting Scheduler, Example 7, is to find the minimum u such that $u = f(u) = g(u) = h(u)$. After $\texttt{time} = u$, the tuples `F_changed`, `G_changed`, and `H_changed` are all set to `FALSE` and the program terminates. Let `F_changed` denotes to $\texttt{F\_changed} = \texttt{TRUE}$ and $\neg$`F_changed` to $\texttt{F\_changed} = \texttt{FALSE}$, so do $(\neg)$`G_changed` and $(\neg)$`H_changed`. From the program (Figure 5.7), we get:

$$(\sigma, 0) \models \texttt{time} = 0 \wedge \texttt{F\_changed} = \texttt{TRUE} \wedge \texttt{F\_changed} = \texttt{TRUE} \wedge \texttt{F\_changed} = \texttt{TRUE} \qquad (\text{ms.0})$$

$$\texttt{time} = r \wedge (\texttt{G\_changed} \vee \texttt{H\_changed}) \Rightarrow \Diamond \begin{bmatrix} (\texttt{time} = r \wedge \neg\texttt{F\_changed}) \\ \vee \\ (\texttt{time} = f(r) \wedge \texttt{F\_changed}) \end{bmatrix} \qquad (\text{ms.1})$$

$$\texttt{time} = r \wedge (\texttt{F\_changed} \vee \texttt{H\_changed}) \Rightarrow \Diamond \begin{bmatrix} (\texttt{time} = r \wedge \neg\texttt{G\_changed}) \\ \vee \\ (\texttt{time} = g(r) \wedge \texttt{G\_changed}) \end{bmatrix} \qquad (\text{ms.2})$$

$$\texttt{time} = r \wedge (\texttt{F\_changed} \vee \texttt{G\_changed}) \Rightarrow \Diamond \begin{bmatrix} (\texttt{time} = r \wedge \neg\texttt{H\_changed}) \\ \vee \\ (\texttt{time} = h(r) \wedge \texttt{H\_changed}) \end{bmatrix} \qquad (\text{ms.3})$$

Let TC denotes the termination condition of the program,

$$TC =_{def} \neg\texttt{F\_changed} \wedge \neg\texttt{G\_changed} \wedge \neg\texttt{H\_changed},$$

the correctness of the program relies on (i) the program will terminate, i.e., $\Diamond TC$, and (ii) the value of `time` will reach the value of u.

In the proofs of the following results, we just show the main steps and omit some of the trivial derivations.

Lemma 1 (time non-decreasing) *The value of* `time` *is non-decreasing:* $(\texttt{time} = r) \Rightarrow \Diamond(\texttt{time} \geq r)$.

Proof:

1. From (ms.1) – (ms.3), we can get:

$$\left(\begin{array}{c} \texttt{time} = r \wedge (\texttt{G\_changed} \vee \texttt{H\_changed}) \\ \vee \\ \texttt{time} = r \wedge (\texttt{F\_changed} \vee \texttt{H\_changed}) \\ \vee \\ \texttt{time} = r \wedge (\texttt{F\_changed} \vee \texttt{G\_changed}) \end{array}\right) \Rightarrow \tag{7.11}$$
$$\Diamond \left(\begin{array}{c} (\texttt{time} = r \wedge \neg\texttt{F\_changed}) \vee (\texttt{time} = f(r) \wedge \texttt{F\_changed}) \\ \vee \\ (\texttt{time} = r \wedge \neg\texttt{G\_changed}) \vee (\texttt{time} = g(r) \wedge \texttt{G\_changed}) \\ \vee \\ (\texttt{time} = r \wedge \neg\texttt{H\_changed}) \vee (\texttt{time} = h(r) \wedge \texttt{H\_changed}) \end{array}\right);$$

2. The left side of Formula 7.11 can be reduced to

$$(\texttt{time} = r) \wedge (\texttt{F\_changed} \vee \texttt{G\_changed} \vee \texttt{H\_changed});$$

3. From the right side of Formula 7.11 and the tautology of $p \wedge q \rightarrow p$, we get

$$(\texttt{time} = r) \vee (\texttt{time} = f(r) \vee \texttt{time} = g(r) \vee \texttt{time} = h(r)).$$

   According to the definition of the functions f, g, and h, we have $f(r) > r$, $g(r) > r$, and $h(r) > r$. Thus, the new right side of Formula 7.11 is

$$(\texttt{time} \geq r);$$

4. Formula 7.11 becomes

$$[(\texttt{time} = r) \wedge (\texttt{F\_changed} \vee \texttt{G\_changed} \vee \texttt{H\_changed})] \Rightarrow \Diamond(\texttt{time} \geq r); \tag{7.12}$$

5. Before time u is reached, i.e., $\texttt{time} = r$, $0 \leq r \leq u$, at least one of `F_changed`, `G_changed`, and `H_changed` will be true:

$$(\texttt{time} = r) \Rightarrow \Diamond(\texttt{F\_changed} \vee \texttt{G\_changed} \vee \texttt{H\_changed}); \tag{7.13}$$

6. Applying tautology $(p \to q) \leftrightarrow (p \to p \wedge q)$ to Formula 7.13, and with the definition of "$\Rightarrow$",

$$(\mathtt{time} = r) \Rightarrow \Diamond[(\mathtt{time} = r) \wedge (\mathtt{F\_changed} \vee \mathtt{G\_changed} \vee \mathtt{H\_changed})]; \quad (7.14)$$

7. Applying $\Diamond T$ to Formula 7.12 and Formula 7.14, we can get

$$(\mathtt{time} = r) \Rightarrow \Diamond(\mathtt{time} \geq r).$$

∎

Theorem 4 (u reached) *The value of* time *will eventually reach the value of u:* $\Diamond(\mathtt{time} = u)$.

Proof:

1. $(\sigma, 0) \models \mathtt{time} = 0$, (given, ms.0);
2. $(\mathtt{time} = r) \Rightarrow \Diamond(\mathtt{time} \geq r)$, (Lemma 1), it is the same as

$$(\sigma, i) \models \mathtt{time} = r \;\; \textit{iff} \;\; (\sigma, j) \models \mathtt{time} \geq r, \text{ for some } j, j \geq i;$$

3. time is monotonic and increasing, while u is limited. A position k can be found, such that

$$(\sigma, k) \models (\mathtt{time} = u);$$

4. From *A2* ($p \Rightarrow \Diamond p$), we have $\Diamond(\mathtt{time} = u)$.

∎

Theorem 5 (termination) *The termination condition will be eventually reached:* $\Diamond TC$.

Proof:

1. $(\mathtt{time} = r) \wedge (\mathtt{G\_changed} \vee \mathtt{H\_changed}) \Rightarrow$
$\Diamond[(\mathtt{time} = r \wedge \neg\mathtt{F\_changed}) \vee (\mathtt{time} = f(r) \wedge \mathtt{F\_changed})]$ (given, ms.1);

2. Referring the reasoning in the proof of Theorem 4, we get

$$(\texttt{time} = r) \Rightarrow \Diamond[(\texttt{time} = r \wedge \neg\texttt{F\_changed}) \vee (\texttt{time} = f(r) \wedge \texttt{F\_changed})];$$

3. Before u has been reached, i.e., $0 \leq r < u$, from Lemma 1:

$$(\texttt{time} = r) \Rightarrow \Diamond(\texttt{time} = f(r) \wedge \texttt{F\_changed});$$

4. When $\texttt{time} = u$,

$$(\texttt{time} = u) \Rightarrow \Diamond(\texttt{time} = r \wedge \neg\texttt{F\_changed}), \quad \text{i.e.,} \quad (\texttt{time} = u) \Rightarrow \Diamond\neg\texttt{F\_changed};$$

5. The same reasoning can give us

$$(\texttt{time} = u) \Rightarrow \Diamond\neg\texttt{G\_changed} \quad \text{and} \quad (\texttt{time} = u) \Rightarrow \Diamond\neg\texttt{H\_changed}$$

6. Put the three formulae together, we get

$$(\texttt{time} = u) \Rightarrow \Diamond(\neg\texttt{F\_changed} \wedge \neg\texttt{G\_changed} \wedge \neg\texttt{H\_changed});$$

7. From Theorem 5 and the rule of $\Diamond T$, we have

$$\Diamond(\neg\texttt{F\_changed} \wedge \neg\texttt{G\_changed} \wedge \neg\texttt{H\_changed}).$$

■

7.5 The Impact of T-Cham Termination Conditions

In order to study a concurrent system, where many events may happen at the same time instant, with the linear temporal logic system, *event interleaving* and *fairness* are two basic and essential assumptions. Event interleaving makes the events happen in a pseudo-linear order on the temporal logic time axis. Fairness guarantees that an event will happen if it is continuously ready to happen. Without the fairness assumption, a T-Cham program and its proof system may not be able to deliver the expected result. Take the Producer-Consumer problem (Example 4) as an example, under the fairness assumption, both producing and consuming actions are fairly chosen; without the assumption, either producing or consuming action may continuously be chosen while the other one never has a chance. In the latter situation, the program will stop after a certain number of steps, which is not what it is intended to. In the terms of the sequences of tuple space states, some sequences are not valid under the fairness assumption.

There is a potential conflict between the fairness assumption and T-Cham termination conditions. The fairness assumption says that any event *will* happen if it is continuously ready to happen, while the termination conditions of a T-Cham program say that whenever any of the conditions is true, the program terminates even if there are pending "continuously ready to happen" events.

The conflict between the fairness assumption and T-Cham termination conditions is not so severe. The termination conditions by no means reject the fairness assumption but just cut a valid sequence short. For example, a T-Cham program may have an execution path (i.e., tuple space state sequence):

$$\sigma = (s_0 s_1 s_2 \cdots s_{k-1} s_k s_{k+1} \cdots s_{n-1} s_n s_n s_n \cdots),$$

without the influence of any termination conditions. The state s_n means no more event could happen. If the program has some termination conditions, and one of them, say, t, is true at the state s_k, according to the semantics of T-Cham termination conditions, the program stops at the state s_k, i.e.,

$$\sigma = (s_0 s_1 s_2 \cdots s_{k-1} s_k s_k s_k \cdots).$$

The sudden stop, although it violates the fairness assumption, is justified because the programmer of the program believes that at the state s_k, where the termination condition t is satisfied, the program has already delivered the expected results. It is, therefore, not necessary for the program to go any further.

The termination conditions have some impact on the temporal logic proof system. A temporal logic formula, which is a theorem on the model

$$\sigma = (s_0 s_1 s_2 \cdots s_{k-1} s_k s_{k+1} \cdots s_{n-1} s_n s_n s_n \cdots),$$

may not be able to be proven on the cut-short model

$$\sigma = (s_0 s_1 s_2 \cdots s_{k-1} s_k s_k s_k \cdots).$$

```
transaction root
          ......
          initialization
                    a=true; c=true;
          reactionrules
                    a leadsto b;
                    c leadsto d;
          ......
endtrans
```

Figure 7.5: A small T-Cham program

Let's take a very small artificial T-Cham program, Figure 7.5, as an example. As discussed in the previous section, the temporal logic model of the program is the set of all possible tuple space state sequences and written as $\mathcal{P}$. If there is no termination condition, under the initial tuple space state of {a, c}, we have:

$$\mathcal{P},\ \{\mathtt{a},\ \mathtt{c}\}\ \models\ \Diamond(\mathtt{b} \wedge \mathtt{d}). \tag{7.15}$$

If the termination condition is b, there are two possible sequences:

$$\{\mathtt{a},\ \mathtt{c}\} \rightarrow \{\mathtt{b},\ \mathtt{c}\} \rightarrow \{\mathtt{b},\ \mathtt{c}\} \rightarrow \{\mathtt{b},\ \mathtt{c}\} \rightarrow \cdots, \qquad \text{or}$$
$$\{\mathtt{a},\ \mathtt{c}\} \rightarrow \{\mathtt{a},\ \mathtt{d}\} \rightarrow \{\mathtt{b},\ \mathtt{d}\} \rightarrow \{\mathtt{b},\ \mathtt{d}\} \rightarrow \{\mathtt{b},\ \mathtt{d}\} \rightarrow \cdots.$$

In the first sequence, only "$\Diamond\mathtt{b}$" can be proven, but in the second sequence, "$\Diamond(\mathtt{b} \wedge \mathtt{d})$" can be proven. Putting them together, we have:

$$\mathcal{P},\ \{\mathtt{a},\ \mathtt{c}\}\ \models\ \Diamond\mathtt{b}. \tag{7.16}$$

If the termination condition is d, we have a similar result.

The Formula 7.16 is acceptable under the termination condition b because this is what the programmer of this program wants. Actually, Formula 7.16 is the weakening form of Formula 7.15 as b is the weakening form of $(\mathtt{b} \wedge \mathtt{d})$[8]. In other words, if Formula 7.15 can be satisfied, Formula 7.16 can.

[8] $\mathtt{b} \wedge \mathtt{d} \rightarrow \mathtt{b}$

Generally speaking, for a T-Cham program, we can prove:

$$\mathcal{P} \models \Diamond P, \tag{7.17}$$

where P is the temporal logic property of the program, i.e., P is a theorem under the model $\mathcal{P}$. If we take the initial tuple space state I into account, we will have

$$\mathcal{P}, I \models \Diamond P'. \tag{7.18}$$

$P' \rightarrow P$ because P has to be satisfied on all possible tuple space sequences starting with any initial tuple space states, including I. If the program has termination condition T, $T = t_1 \vee t_2 \vee \cdots \vee t_n$, either P or P' becomes T, i.e.,

$$\mathcal{P} \models \Diamond T, \quad \text{or} \quad \mathcal{P}, I \models \Diamond T. \tag{7.19}$$

7.6 Conclusion

In this chapter, we adopted a temporal logic proof system [130, 129] to T-Cham program verification. First, we briefly introduced the theory of the temporal logic. We did not put much strength on the temporal logic theory itself. We treated it from the point of view of its application instead of pure theoretical research. In other words, we apply the temporal logic to the verification of T-Cham programs. The major contribution of this chapter is on how to translate a T-Cham program into its corresponding temporal logic proof system and then verify the temporal properties of the program. Several examples are used to illustrate the verification of T-Cham programs.

The impact of the race condition problem (Section 7.3) was carefully studied. Generally speaking, the temporal logic proof system cannot handle the T-Cham programs which are under race conditions. Fortunately, if a program is in NRURC class, the proof system is still valid. We believe that NRURC class covers a large scope of sensible T-Cham programs, including the Dining Philosophers problem. We also notice that linear logic [79, 78, 166, 10] can easily delineate the resource-like variables. How to apply linear logic to T-Cham is a very interesting research topic and needs much more further study.

In this chapter, we also discussed the impact of T-Cham termination conditions on the temporal logic proof systems. We found out that the termination conditions reject some temporal properties of a T-Cham program but still keep those properties which are the same as or implied by the termination conditions. We believe it is justified because the purpose of termination conditions is to tell a T-Cham program not to go any further.

Chapter 8

Advanced Notations: Hierarchical Tuple Spaces and Tuple Mapping

The experience of program development suggests that a large program should be constructed from a number of smaller components. In addition, some large data may have their internal structures. T-Cham provides *hierarchical tuple space structure* and *tuple mapping* mechanisms to decompose a large tuple into a number of smaller subtuples at different levels of tuple spaces. These mechanisms apply hierarchical views to T-Cham tuples and provide a means of constructing modular (or structured) T-Cham programs. A transaction of a tuple space may consist of a number of transactions (sub-transactions). The relationships between the tuples in those two different layers of tuple spaces are maintained by the tuple mapping mechanisms. Each of those transactions is isolated from the others. When put together, they constitute the whole reaction system, while changes to a transaction (or subtransaction) are transparent to the others.

In this chapter, we first study the internal structures of tuples, the need for hierarchical tuple spaces, tuple mappings, and their impact on transaction granularities (Section 8.1). Section 8.2 then explains the technical details of tuple mappings and introduces the concept of mapping masks. Section 8.3 concentrates on a special type of masks—*regular masks.* A matrix multiplication example is given in Section 8.4. In Section 8.5, we briefly discuss the implementation issues which are raised by those advanced notations. Finally, we conclude this chapter with a discussion in Section 8.6.

8.1 Tuple Structures, Hierarchical Tuple Spaces and Transaction Granularities

8.1.1 Case Study

In the previous chapters, we studied the basic T-Cham notations and also illustrated the style of T-Cham programming by a number of examples, but we have not yet investigated the internal structures of tuples, especially large tuples.

Some large tuples do have internal structures, just like data structures in the imperative programming paradigm. Let's take matrix summation as an example to explain the internal structures of tuples. For brevity, we assume they are one dimensional vectors. Suppose we are asked to calculate:

$$\mathbf{C} = \mathbf{A} + \mathbf{B}$$

where **C**, **A**, and **B** are n element vectors[1]. A T-Cham program is given in Figure 8.1. Inside of the transaction `sum_vectors`, it is like:

```
for (i=0; i<n; i++) C[i] = A[i]+B[i];
```

A pictorial description of the program is given in Figure 8.2. In the figure, an oval encapsulates a tuple. The grid rectangle inside of an oval denotes the value of the tuple.

We do not regard the program as a good program. It fails to reveal the inherent parallelism of the original problem. If the T-Cham compiler cannot figure out this inherent parallelism, the program is virtually a sequential program, because all the calculations are carried out by the transaction `sum_vectors`, which relies on a loop. A T-Cham compiler, which cannot have the domain knowledge of all problems, may not be able to detect the maximum possible parallelism of problems. In other words,

[1]In Section 2.2.3, the same problem is used to show the programming style of Linda.

```
transaction root
        tuples
                float A[N], B[N], C[N];
        initialization
                init_A(); init_B();
        reactionrules
                A, B leadsto C by sum_vectors;
        termination
                on (|A|==0 && |B|==0) do output_C();
endtrans
```

Figure 8.1: The T-Cham program of Vector Summation, the First Approach

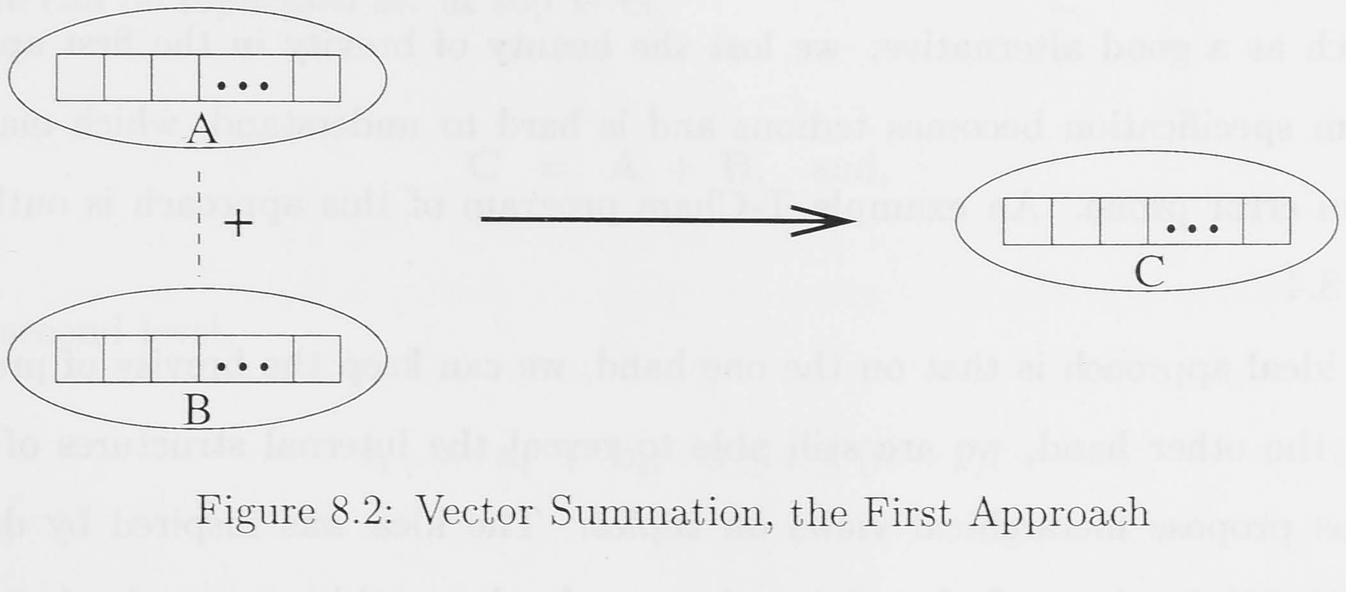


Figure 8.2: Vector Summation, the First Approach

we cannot solely rely on compilers to reveal the inherent parallelism of a problem. It would be very unfortunate if we, as programmers, already know some degree of inherent parallelism, but failed to implement it.

The culprit here is that we take an vector as a whole without looking into its internal structures. If, instead of saying

$$\mathbf{C} \;=\; \mathbf{A} + \mathbf{B},$$

we say

$$\mathbf{c}_i \;=\; \mathbf{a}_i + \mathbf{b}_i, \quad 0 \leq i \leq (n-1),$$

we will be able to take the advantage of the inherent parallelism, Figure 8.3.

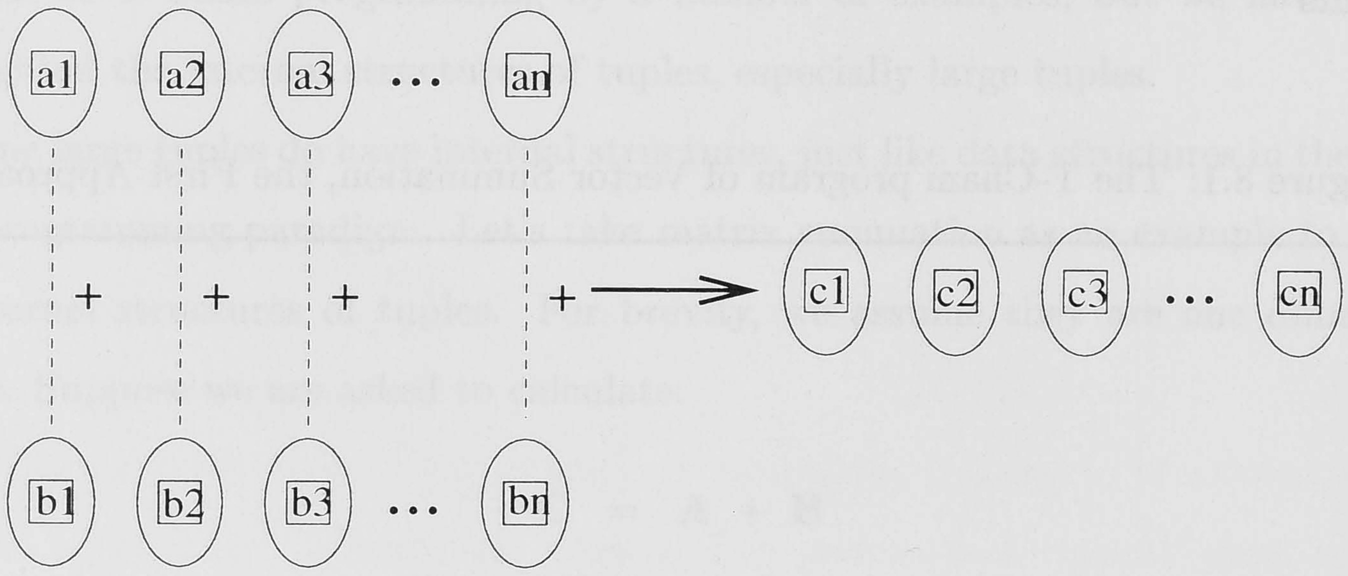


Figure 8.3: Vector Summation, the Second Approach

Although we achieved a high degree of parallelism this time, we cannot regard the approach as a good alternative: we lost the beauty of brevity in the first approach. Program specification becomes tedious and is hard to understand, which makes the program error prone. An example T-Cham program of this approach is outlined in Figure 8.4

An ideal approach is that on the one hand, we can keep the brevity of programs, but on the other hand, we are still able to reveal the internal structures of tuples. We thus propose hierarchical views on tuples. The idea was inspired by database views [58, 59], but instead of applying views on database tables, we apply views on the internal structures of tuples. A tuple, for example, a vector in the previous examples, may be taken as a whole or broken into smaller pieces. We can operate on the vector

```
transaction root
          tuples
                    float a1, a2, ..., an, b1, b2, ..., bn, c1, c2, ..., cn;
          initialization
                    a1=2.1; a2=2.2; ...; b1=3.1; b2=3.2; ...;
          reactionrules
                    a1, b1 leadsto c1 by { c1=a1+b1; }
                    a2, b2 leadsto c2 by { c2=a2+b2; }
                    ......
                    an, bn leadsto cn by { cn=an+bn; }
          termination
                    on (|a1|==0 && |a2|==0 && ... |an|==0 &&
                         |b1|==0 && |b2|==0 && ... |bn|==0) do output();
endtrans
```

Figure 8.4: The T-Cham program outline of Vector Summation, the Second Approach

itself or zoom in and operate on each individual element in parallel. To avoid any possible confusion among different views of the same instance of a tuple, we propose a hierarchical tuple space structure. Different views of a tuple are presented on different levels of tuple spaces.

With the help of multiple tuple views and hierarchical tuple spaces, the former problem can be rephrased as: at top level,

$$\mathbf{C} \;=\; \mathbf{A} + \mathbf{B}, \quad \text{and,}$$

at the second level,

$$\mathbf{c}_i \;=\; \mathbf{a}_i + \mathbf{b}_i, \quad 0 \leq i \leq (n-1).$$

Figure 8.5 is the pictorial description of the approach. On the one hand, the program is as simple as $\mathbf{C} \;=\; \mathbf{A} + \mathbf{B}$, but on the other hand, if we dig a little bit further, we will see that the vectors are broken into pieces and evaluated in parallel.

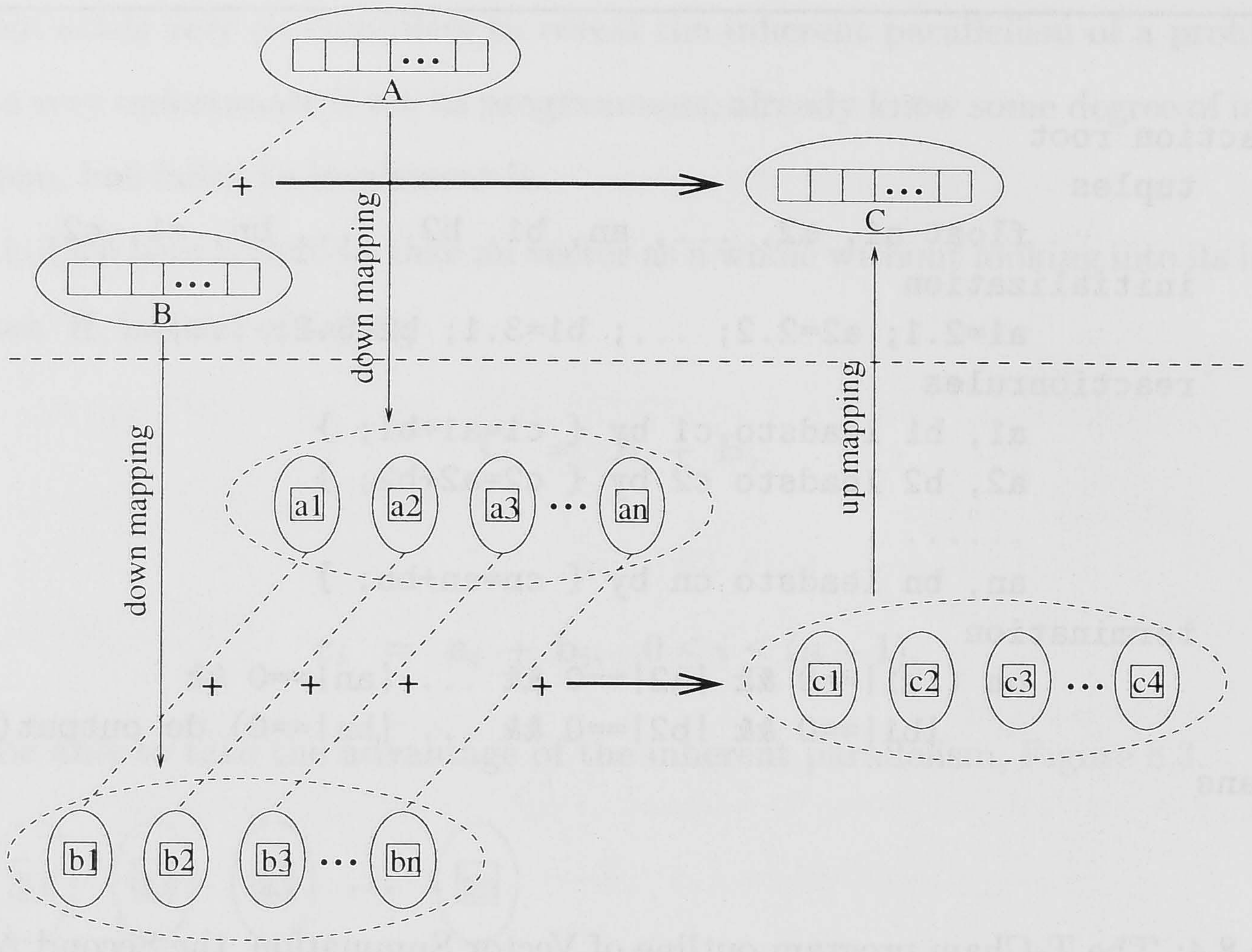


Figure 8.5: Vector Summation, the Third Approach

8.1.2 New Concepts: Hierarchical Tuple Spaces, Tuple Mapping and Transaction Granularities

From the discussion of the previous section, we find out that the best solution for the vector summation problem is to have two layers of tuple spaces (hierarchical tuple space structure), Figure 8.5. On the top layer, we take a vector as a whole no matter how big the vector is, and thus, we have an abstract view of the program: $\mathbf{C} = \mathbf{A} + \mathbf{B}$. A single transaction is responsible for this calculation. On the bottom layer, the detailed structures of vectors **A**, **B**, and **C** are revealed: they are broken into individual elements. The transactions on this layer operate on $\mathbf{a}_i$ and $\mathbf{b}_i$ $(0 \leq i \leq (n-1))$, where $\mathbf{a}_i$ is the i^{th} element of vector **A**, and $\mathbf{b}_i$ is the i^{th} element of vector **B**. Therefore, we have $\mathbf{c}_i = \mathbf{a}_i + \mathbf{b}_i$. There are n transactions. From the point of view of the top layer tuple space, they can be called sub-transactions. Each of them is responsible for the calculation of $\mathbf{a}_i + \mathbf{b}_i$, where i takes any integer value between 0 and $(n-1)$ inclusively, i.e., $i \in [0..(n-1)]$.

The two layers of tuple spaces of Figure 8.5 are not independent to each other. They have their internal relations. For example, vector **A** is decomposed into $\mathbf{a}_i$, $0 \leq$

$i \leq (n-1)$. Vector $\mathbf{B}$ is decomposed into $\mathbf{b}_i$, $0 \leq i \leq (n-1)$. The i^{th} element of $\mathbf{A}$ (i.e., $\mathbf{a}_i$) plus the i^{th} element of $\mathbf{B}$ (i.e., $\mathbf{b}_i$) makes the i^{th} element of $\mathbf{C}$ (i.e., $\mathbf{c}_i$). And finally, $\mathbf{c}_i$ $(0 \leq i \leq (n-1))$ together compose vector $\mathbf{C}$. To preserve the relationship between the tuple spaces and among the tuples themselves, we propose some tuple mapping mechanisms. With the mechanisms, we can specify the relations between the tuples in different layers of tuple spaces. We also introduce a special operator `@`. When a large tuple is broken into small pieces, the operator can keep the track of the order of those small pieces. The order is essential in measuring from which part of the large tuple a particular small piece comes, for example, $\mathbf{b}_i$, so that right calculation can be carried out. In addition, the order also plays a very important role in assembling small tuples to a big one, e.g., from $\mathbf{c}_i$, $0 \leq i \leq (n-1)$ to $\mathbf{C}$.

To be more general, the mapping between tuples is under masks. A mask is a window on a tuple or tuples. Through the window, only part of the underlying tuple(s) can be seen. Different patterns of the window give different views of the underlying tuple(s). Masks bring much more flexibility into tuple mappings. They provide multiple views on tuples.

Another benefit of the hierarchical tuple space structure is that transaction granularity adjustment (Section 3.1.4) will be much easier. The root transaction has the largest granularity. Down the tuple space hierarchy, granularities become smaller and smaller. Take the matrix summation problem for example again, if they are m dimensional matrices, the problem can be solved by:

- simply adding the two matrices together, or
- m instances of two $(m-1)$ matrix summation, or
- $m \times m$ instances of two $(m-2)$ matrix summation, or
- m^3 instances of two $(m-3)$ matrix summation, or
- $\cdots$
- m^m instances of element summation, or

- the mix of some above choices, for example, one instance of two $(m-1)$ matrix summation and $(m-1) \times m$ instances of two $(m-2)$ matrix summation.

There are some other Linda based multiple tuple space approaches [76, 131], but they are different from our proposal. Those approaches are mainly concerned with creating and reclaiming tuple spaces. They do not have much effort on tuple mapping. In our approach, tuple spaces are automatically created and reclaimed whenever necessary. Tuple mapping between tuple spaces is our main concern.

8.2 Tuple Mappings and Plain Masks

The mapping from one tuple to another (or others) provides different views on a large tuple and different granularities to the transaction which operates on this tuple. This mapping is achieved by a *mask* (or a window) on the tuple.

A mask is a group of mask elements. An element of the mask has its value in the form of "*coef*∗*vis*∗*type*" or just "*vis*∗*type*", where *coef* is a coefficient ($coef \in \mathcal{N}$), *vis* is the visibility flag ($vis \in \mathcal{I}$), and *type* the type of the hole ($type \in \{$`char`, `int`, `float`, $\cdots\}$, i.e., the type system of T-Cham). $\mathcal{N}$ is the set of natural numbers, and $\mathcal{I}$ is the set of integer numbers. The default value of *coef* is 1, which can be omitted; otherwise, it means *coef* instances of "*vis*∗*type*". "$vis = 0$" means the underlying data cannot be seen, while "$vis = 1$" is a hole on the mask so that the underlying data can be read out through it. The value of *type* tells the mask how large the hole is, or what kind of data type can be seen through the hole. A mask whose *vis*es only contain the 0s and 1s as discussed in this section is called a *plain mask*.

Suppose we have a tuple `x` and a mask `m`,

```
tuple {                                  mask {
    char name[]="data";                      {0,0,1,1}*char;
    int i=15;                                1*int;
    float a[20]={76.8,3.5,4,...};            {0,1,18*0}*float;
} x;                                     } m;
```

a new tuple, `("ta",15,3.5)`, is obtained by viewing the tuple `x` through the mask `m`, written as "`x\m`" and read as "*tuple* `x` *under mask* `m`".

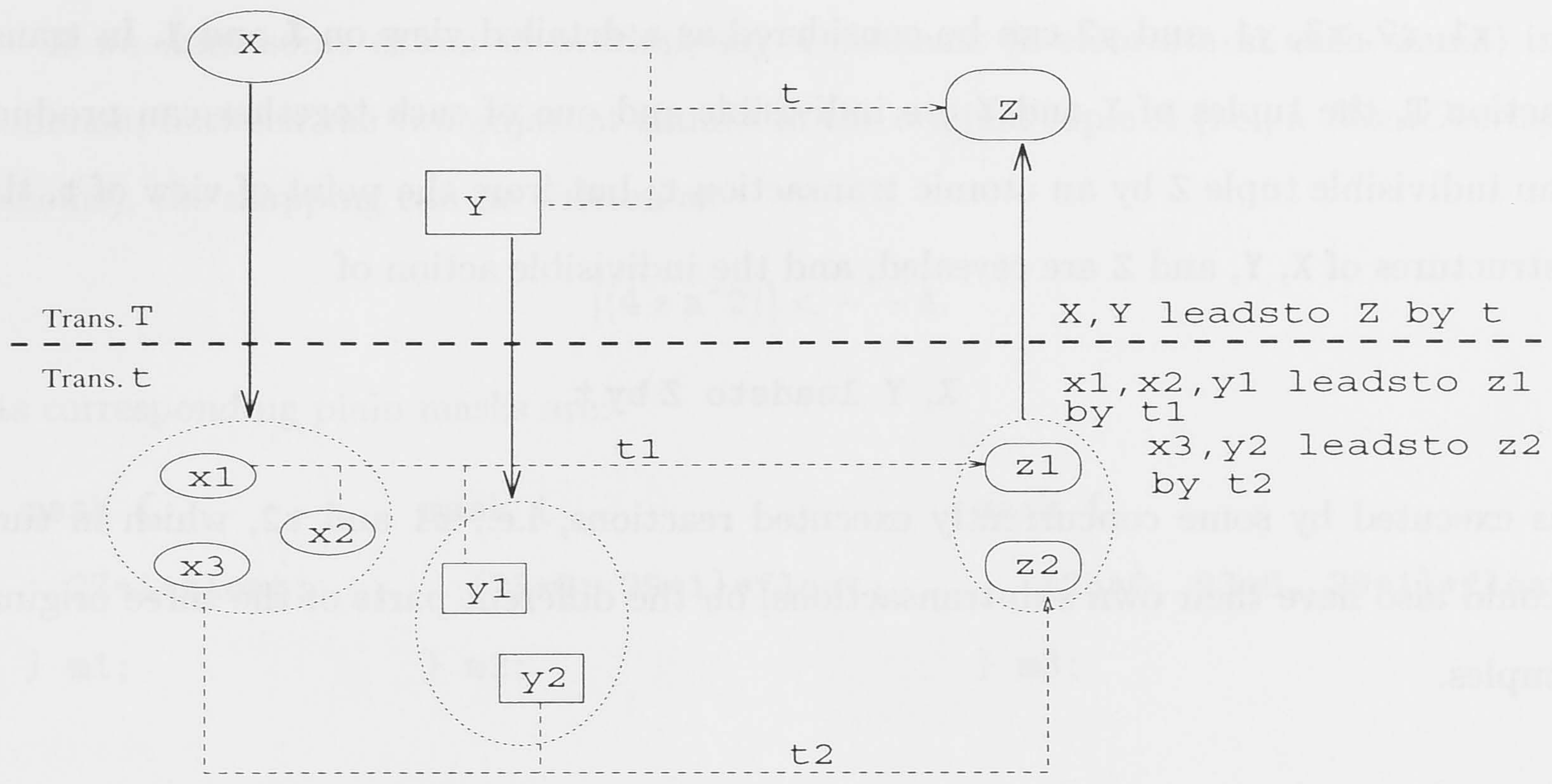


Figure 8.6: The Vertical Mappings of Tuples

The mappings among tuples establish the relations between the tuples on a tuple space and its parent's or child's tuple space, Figure 8.6. They can be used to initialize the child's tuple space, the *down mapping* (represented by `<--`), or return the result tuples to the parent's tuple space, the *up mapping* (`-->`).

Supposing in transaction `T`, we have a reaction rule of:

```
X,Y leadsto Z by t;
```

while in the `initialization` section of transaction `t`, we have

```
[(x1\m1, x2\m2, x3\m3)] <-- X;  [(y1\n1, y2\n2)] <-- Y;
```

and in the `termination` section of the transaction,

`on (`*term_cond*`) do [(z1\s1, z2\s2)] --> Z;`

To execute the action, a T-Cham sub-transaction `t` is invoked, and `X` and `Y` are mapped (decomposed) into `x1`, `x2`, `x3`, `y1`, and `y2` on a new tuple space according to the specified relations between `X`, `Y` and `x1`, `x2`, `x3`, `y1`, and `y2`. After the execution of `t`, i.e., the reactions on `x1`, `x2`, `x3`, `y1`, and `y2` yielding `z1` and `z2`, a new `Z` is generated. Transaction `T`'s tuple space could have tuples other than `X`, `Y`, and `Z`, but in `t`, only those three can be seen. Of course, `t` may have its own private tuples on its tuple space.

x1, x2, x3, y1, and y2 can be considered as a detailed view on X and Y. In transaction T, the tuples of X and Y are indivisible and one of each together can produce an indivisible tuple Z by an atomic transaction t; but from the point of view of t, the structures of X, Y, and Z are revealed, and the indivisible action of

```
X, Y leadsto Z by t
```

is executed by some concurrently executed reactions, i.e., t1 and t2, which in turn could also have their own sub-transactions, on the different parts of the three original tuples.

8.3 Regular Masks

A *regular mask* is a kind of *shorthand* used in the decomposition and assembling of array-like regular data structure. It realizes the *distributed data* structure in the DINO programming language approach [154]. For example, we can have *Block* and *BlockOverlap* masks for one dimensional arrays and *BlockRow*, *BlockRowOverlap* (see Section 2.2.7) for two dimensional arrays etc. If we decompose a one dimension array of $A[100]$ (float type) to 4 one dimension array of $a[25]$'s (*Block*), we can write

$$[(4 * \mathtt{a})] <-- \mathtt{A}.$$

If without the shorthand, we have to define 4 masks for each of individual tuple a:

```
mask {                mask {                     mask {
  25*1*float;           {25*0, 25*1}*float;        {25*0, 25*0, 25*1}*float;
} m1;                 } m2;                      } m3;


mask {
  {25*0, 25*0, 25*0, 25*1}*float;
} m4;
```

where m1 picks up the first 25 elements of $A[100]$, m2 the second 25 elements, m3 the third 25 elements, and m4 the last 25 elements.

If we want some extent of overlap, say 4 elements (2 elements in each chunk) in common, between the two adjacent chunks in the original tuple A (i.e., a *BlockOverlap* scheme), the mapping can be written as

$$[(4 * \mathtt{a\~{}2})] <-- \mathtt{A}.$$

Its corresponding plain masks are:

```
mask {              mask {                       mask {
  27*1*float;         {23*0, 29*1}*float;          {25*0, 23*0, 29*1}*float;
} m1;               } m2;                        } m3;

mask {
  {25*0, 25*0, 23*0, 27*1}*float;
} m4;
```

Higher dimensional array masks can be defined accordingly.

A large array can be broken into smaller pieces for parallel calculations. The parallel calculations may happen in any order. We have to preserve the positions of those small pieces in the original array; otherwise, we cannot assemble them back. A special operator `@` is defined to keep the track of the order. For example, `a@A`= i ($i = 1$ if the `a` is the first chunk of `A`, $i = 2$ for the second chunk, and so on) specifies that `a` corresponds to the i^{th} piece of `A`. This can be written as just `a@` if there is no confusion on where `a` comes from. A dot prefix notation is used if the order is non-linear, for example, `a@T`= 1.2.3, where 1 means the first chunk of `T`, 2 means the second piece of the chunk, and 3 means the third elements of the piece, see Figure 8.7.

8.4 Matrix Multiplication: an Example of Using Masks

Example 12 (Matrix Multiplication) *Supposing we have two $N*N$ matrices $A[N][N]$ and $B[N][N]$, we calculate their product $C[N][N]$. A T-Cham program is given in Figure 8.8.* ∎

The program has two layers in its transaction structure. The `root` transaction has one sub-transaction `multi_matrix`, where the down and up mappings with regular

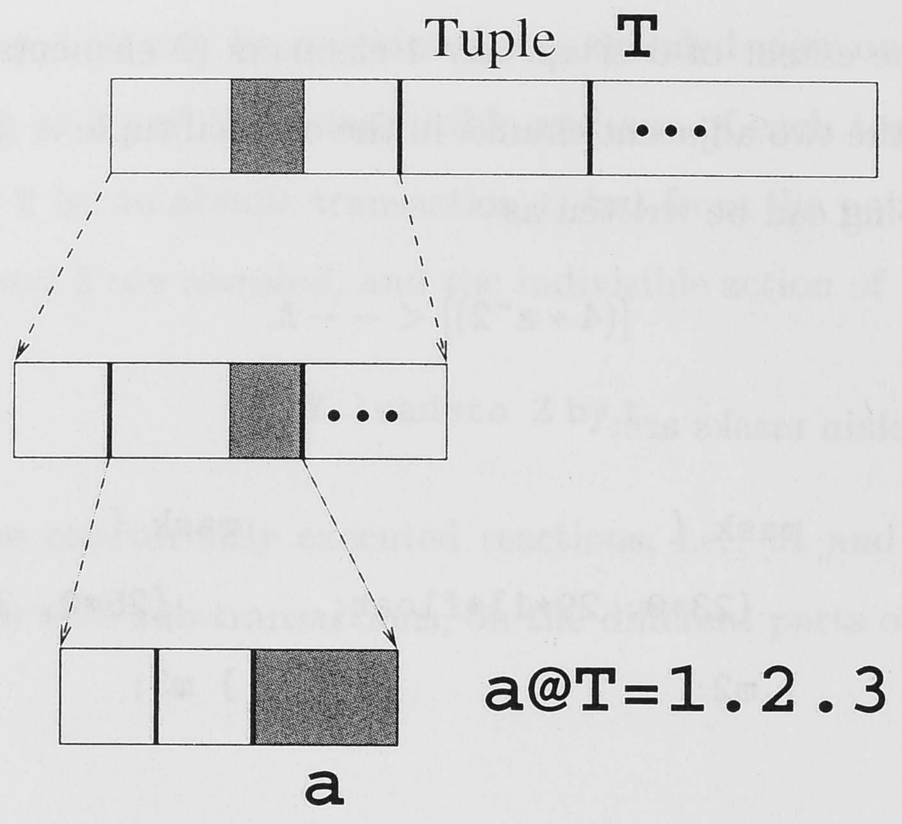


Figure 8.7: The Ordering Operator

masks are used to initialize the sub-transaction tuple space and bring the values back respectively. On the sub-transaction's tuple space, tuple `A` is decomposed into N tuples of `x`, each of which is one of the N rows of matrix A. `p1` and `p2` are pre-conditions (the values of original matrices), and `q1` and `q2` post-conditions:

$$\texttt{p1} \equiv A = (A_{ij}) \wedge B = (B_{ij}), \quad \texttt{q1} \equiv C = (C_{ij}), \text{ where } C_{ij} = \textstyle\sum_{k=1}^{N} A_{ik} \times B_{kj};$$
$$\texttt{p2} \equiv x = (A_i) \wedge y = (B_{ij}), \quad \texttt{q2} \equiv z = (C_i) \text{ where } C_i = \textstyle\sum_{k=1}^{N} A_i \times B_{kj}.$$

In comparison to other approaches, the program has a very nice and neat top view, i.e., the `root` transaction. It specifies the basic criteria of the program: we have A and B two matrices, and we want to make matrix C from them by transaction `multi_matrix`. The pre- and post-conditions of transaction `multi_matrix` are $p1$ and $q1$ respectively. At this stage, we do not care how this can be done so long as transaction `multi_matrix` complies with its pre- and post-conditions.

There are several ways to implement transaction `multi_matrix`. In this example, we break matrix A into rows, i.e., smaller pieces, but keep matrix B as it is. A row of the matrix A and the whole matrix B can produce a row of the matrix C. The real calculations are carried out by the transaction `multiply`. Again, at this stage, we don't care about the details of transaction `multiply` so long as it complies with its pre-conditions $p2$ and post-conditions $q2$.

```
#define N       128
transaction root
       tuples
               float A[N][N], B[N][N], C[N][N];
       initialization
               init_A(); init_B();
       reactionrules
               A, B leadsto C by multi_matrix;
       termination
               on (|A|==0 && |B|==0) do output_C();
       subtransactions
               multi_matrix: p1//q1
endtrans

transaction multi_matrix
        tuples
                float x[N], y[N][N], z[N];
        initialization
                [(N*1*x)] <-- A;  [(y)] <-- B;
        reactionrules
                x, !y leadsto z by multiply;
        termination
                on (|x|==0) do [(N*1*z)] --> C;
        subtransactions
                multiply: p2//q2
endtrans

transaction multiply
#language C
#tuple float x[N], y[N][N], z[N];
    multiply()
    {   int i,j;

        for (i=0; i<N; i++) {
            z[i]=0;
            for (j=0; j<N; j++) z[i] += x[j]*B[j][i];
        }
        z@ = x@;    /* z keeps the same order as x */
    }
endtrans
```

Figure 8.8: The Multiplication of Two Matrices

The same idea applies to transaction `multiply` as well.

Any changes to a transaction are isolated from other transactions on the same level and the levels above. For example, if we change transaction `multi_matrix` to a different algorithm, it does not have any impact on the logic and structure of transaction `root`. Similarly, any changes to transaction `multiply`, even radical changes, have no impact on `multi_matrix` and `root` either.

Starting with the nice and neat top view of transaction `root` followed by carefully designed transactions `multi_matrix` and `multiply`, we can achieve an efficient implementation of the program without losing programmability (i.e., easy to manage by human beings).

8.5 Implementation Issues

We have not yet had a real implementation of those new mechanisms, i.e., hierarchical tuple spaces and tuple mapping, developed in this Chapter (previous sections). In this section, we briefly discuss some implementation issues raised by those new mechanisms.

T-Cham Machine, developed in Chapter 6, cannot handle hierarchical tuple spaces and tuple mapping. We envisage three extensions to the original T-Cham Machine model to implement those new mechanisms.

The first approach keeps the single flat monolithic tuple space structure of the T-Cham Machine and does not match the logic hierarchical tuple spaces of T-Cham programs. It relies on reaction rule rewriting, tuple renaming, and tuple mapping bookkeeping schemes to achieve the hierarchical tuple spaces and tuple mapping of the original T-Cham programs.

Let's take the matrix multiplication program in Figure 8.8 as an example. There are two layers of tuple spaces in this program. On the top layer, a tuple `A` and a tuple `B` (i.e., matrices `A` and `B`) produce a tuple `C`, which is the product of the two matrices. On the bottom layer, a tuple `x`, which is a row of the matrix `A`, and a tuple `y` (the same as the matrix `B`) yield a tuple `z`. The relationships between tuple `A` and tuple `x`, tuple `B` and tuple `y`, and tuple `C` and tuple `z` are defined in the program. To put the tuples of different logic tuple spaces into a flat monolithic tuple space, we rename those tuples and let them keep the information of their original logic tuple spaces. For example, all the tuples of the top layer tuple space have the suffix "`.root`" appended, and all the tuples on the bottom layer have "`.root.multi_matrix`" appended. Thus tuple `A` becomes tuple `A.root`, `B` to `B.root`, `x` to `x.root.multi_matrix`, and so on. Under the new names of the tuples, the reaction rules are rewritten accordingly. The new reaction rules operate on the same tuple space. In addition to those tuple space name suffixes, each tuple also has a flag which uniquely indicates the different instances of a logic tuple space. For example, if we have two instances of tuple `A` and tuple `B`, we may have two instances of the bottom layer tuple space. Three equivalent reaction rules are automatically added to the system. They are

$$\texttt{A.root}\ \text{leadsto}\ N * \texttt{x.root.multi\_matrix}$$

$$\texttt{B.root}\ \text{leadsto}\ \texttt{y.root.multi\_matrix};$$

$$N * \texttt{z.root.multi\_matrix}\ \text{leadsto}\ \texttt{C.root};$$

The tuple mapping relationships of the original program are associated with and realized by the three reaction rules. Some basic performance measurements of the matrix multiplication example under this implementation approach are given in Section 6.3.2.

As discussed in Chapter 6, T-Cham Machine scales well with large tuple space. We believe that most T-Cham programs can be efficiently implemented with this approach. When the number of tuple types (not the tuples themselves) gets very big, the test of reaction rule conditions and the termination conditions will become less efficient. The larger the number, the less efficient the test. Although we do not have experimental data yet, we believe there exists a quantitative threshold on the number of tuple types a T-Cham Machine can efficiently handle. The threshold is essential for T-Cham to choose this flat monolithic tuple space implementation approach. It also serves as the benchmark for the third approach to decide to what extent this monolithic tuple space can be.

The second implementation approach matches the logic tuple space structures. The operations inside each tuple space are conducted by individual T-Cham Machine. Keeping using the matrix multiplication program example, under this approach, we will have a T-Cham Machine for the root transaction and a T-Cham Machine for transaction multi_matrix. If we have two instances of tuple A and tuple B, we will have a T-Cham Machines for each of the two instances of transaction multi_matrix. The approach keeps the hierarchical tuple space structures of the original T-Cham programs. Tuple spaces are dynamically created and reclaimed. They are relatively small, and thus the test of reaction rule conditions and the termination conditions will be more efficient. The overhead of this approach comes from the creating and reclaiming of the hierarchical tuple spaces. It may be significant if the ratio of tuple space operations, which relate to tuple space creation and reclaiming, to transaction operations is high. The quantitative measurement of the ratio needs to be established. It will be the other bench mark for the third approach.

The third approach is a mix of the first and second approaches. Hopefully, it could properly balance their benefits and difficulties. The success of this approach will depend on the experimental data and quantitative measurements of the other two approaches.

8.6 Summary

In this chapter, we first discussed internal tuple structure, hierarchical tuple space and transaction granularity issues of T-Cham, and then, proposed the idea of tuple mappings and mapping masks. Two kinds of masks—plain masks and regular masks—are studied. A regular mask is just a shorthand of a plain mask. Masks play a central role in revealing the internal tuple structures of tuples and providing hierarchical views to T-Cham tuples and tuple spaces. Transactions at different levels of the hierarchical tuple spaces give T-Cham programs modularity and top-down abstraction. The implementation of the hierarchical tuple space structure and tuple mapping is also mentioned. Three approaches are suggested.

Chapter 9

A Compositional Proof System

In the last chapter, we discussed tuple mappings and subtransactions. Generally speaking, a T-Cham program can be constructed by the *union* and/or *superposition* of transactions. In this chapter, we study the theory of those two kinds of transaction compositions and their effects on the T-Cham proof system. The union operation combines two smaller transactions into a big one, while the superposition makes a transaction as a sub-transaction of another one. In other words, union is used to juxtapose the corresponding sections of two different T-Cham transactions together, while superposition is responsible for the layers, or a hierarchical structures, of the result transactions.

There may have some other kinds of compositions, such as *intersection* and *Cartesian product* etc. As our main concern is to build a large transaction from a number of smaller ones, we will not discuss them in this thesis.

9.1 Union of Transactions

The union of two transactions T_1 and T_2 is written as $T_1 \| T_2$. To be united, the two transactions should be compatible, by which we mean that there are no inconsistencies in their tuple declaration, tuple initialization, and termination condition sections.

Definition 7 (The Union of Transactions) *The union of two transactions is obtained by combining the corresponding sections of the two transactions together.* ∎

Normally, there are five sections in a transaction specification. As the transactions to be united are compatible, there are no problems in the combination of two `tuples` sections. The `initialization` section creates the initial tuples in a tuple space. After being united, the two transactions share one new tuple space, that is, the two tuple spaces of these two transactions are also united. The initial tuples in the new tuple space are the summation of each individual's initial tuples. The union of two `reactionrules` sections simply juxtaposes the reaction rules in each section. A reaction rule in T-Cham is actually a *reaction rule schema* and can have multiple instances at a time; therefore, it is not necessary to keep multiple appearances of the same reaction rules. The `termination` and the `subtransaction` sections are easy to join given the two transactions are compatible.

Example 13 (Sleeping Barber: the union version) *The Sleeping Barber problem will be even clearer if we break it down into three pieces: transaction* `shop` *specifies the activities related to the barber's shop,* `customer` *the activities of customers, and* `barber` *the activities of the barber. The resulting T-Cham program is given in Figure 9.1. Each of the three transactions takes care of its own activities, and the completed system is the union of those three transactions, i.e.,* `shop||customer||barber`. ∎

9.2 Union of Temporal Formulae

Recall that in Section 7.2, we discussed the temporal logic interpretation of a transaction. This section studies the effect of transaction union on these temporal logic formulae.

Definition 8 (The Union of Two Temporal Formulae) *Suppose that we have two transactions T_1 and T_2 to be united, and Π_1 and Π_2 are the temporal logic formulae translated from them respectively. The union of Π_1 and Π_2 juxtaposes the formulae of them together, provided that the new united transaction is NRURC if it is under*

```
transaction shop
        tuples
                fifo boolean pin, pout;
                boolean chair;
        initialization
                [i:1..N]::chair=TRUE;
        reactionrules
                nil leadsto pin;    -- a new customer is coming
                pout leadsto nil;   -- a customer is leaving

endtrans

transaction customer
        tuples
                fifo boolean pin, pwt, pcut;
                boolean bsp, bwk, chair;
        reactionrules
                pin, bsp leadsto pcut, bwk;
                pin, chair leadsto pwt when (!bsp);
endtrans

transaction barber
        tuples
                fifo boolean pwt, pcut, pout;
                boolean bsp, bwk, bfin, chair;
        initialization
                bsp=TRUE;
        reactionrules
                pcut, bwk leadsto pout, bfin;
                pwt, bfin leadsto pcut, chair, bwk;
                bfin leadsto bsp when (|pwt|==0);
endtrans
```

Figure 9.1: The Transactions of shop, customer, and barber

race condition: for those formulae which are assertions on the initial tuple space state, the new formula is the assertion on the the initial state of the new shared tuple space; otherwise, the formulae are simply put together. ∎

Let's translate the transactions `shop`, `customer`, and `barber` to temporal logic formulae and then study the union of these formulae.

The temporal logic formulae translated from transaction `shop`, referred as $\mathcal{A}_{shop}$, are:

$$(\sigma, 0) \models |\texttt{chair}| = N \qquad \textit{(shop0)}$$
$$\texttt{TURE} \Rightarrow \Diamond \texttt{pin} \qquad \textit{(shop1)}$$
$$\texttt{pout} \Rightarrow \Diamond \texttt{TURE} \qquad \textit{(shop2)}$$

Those from transaction `customer` ($\mathcal{A}_{cust}$) are:

$$(\sigma, 0) \models \texttt{TRUE} \qquad \textit{(cust0)}$$
$$\texttt{pin} \wedge \texttt{bsp} \Rightarrow \Diamond(\texttt{pcut} \wedge \texttt{bwk}) \qquad \textit{(cust1)}$$
$$\texttt{pin} \wedge \texttt{chair} \wedge \neg\texttt{bsp} \Rightarrow \Diamond \texttt{pwt} \qquad \textit{(cust2)}$$

and those from transaction `barber` ($\mathcal{A}_{barb}$) are:

$$(\sigma, 0) \models \texttt{bsp} \qquad \textit{(barb0)}$$
$$\texttt{pcut} \wedge \texttt{bwk} \Rightarrow \Diamond(\texttt{pout} \wedge \texttt{bfin}) \qquad \textit{(barb1)}$$
$$\texttt{pwt} \wedge \texttt{bfin} \Rightarrow \Diamond(\texttt{pcut} \wedge \texttt{chair} \wedge \texttt{bwk}) \qquad \textit{(barb2)}$$
$$\texttt{bfin} \wedge \neg\texttt{pwt} \Rightarrow \Diamond \texttt{bsp} \qquad \textit{(barb3)}$$

From the point of view of transactions, transaction `root` is the union of transactions `shop`, `customer`, and `barber`, while from temporal logic formulae, $\mathcal{A}_{sb}$ is the union of $\mathcal{A}_{shop}$, $\mathcal{A}_{cust}$, and $\mathcal{A}_{barb}$. For example, the formula *(sb0)* is the union of formulae of *(shop0)*, *(cust0)*, and *(barb0)*.

If we add one more barber, i.e., `shop`||`customer`||`barber`||`barber`, the assertion on the number of the barbers (`bsp`) in the initial tuple space will be revised accordingly.

An interesting observation is that the addition or removal of one `barber` transaction has nothing to do with the other two, while the addition of one more `customer` transaction has no effect on the system behavior.

9.3 Properties of the United Transactions

The properties of a transaction can be studied directly by transforming the transaction into temporal logic formulae, or indirectly from the properties of its components if it is obtained by the union of other transactions.

Definition 9 (Notations) *Suppose that we have two transactions T_1 and T_2 to be united. We use $\mathcal{V}_1$ and $\mathcal{V}_2$ to denote their representative tuple sets respectively. Π_1 and Π_2 are the formulae translated from T_1 and T_2. ψ_1 and ψ_2 are the temporal properties satisfied by Π_1 and Π_2, i.e., $\Pi_1 \vdash \psi_1$ and $\Pi_2 \vdash \psi_2$*[1]*. ψ_1 and ψ_2 are in their* conjunctive normal forms, *e.g., $\psi_1 \equiv \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n$, where φ_i $(1 \leq i \leq n)$ is a* "sub-property"*. $\psi \backslash \mathcal{V}$ means the conjunction of the left components of ψ by deleting these components which contain the variables in $\mathcal{V}$.* ■

Take transactions `shop` and `customer` for example: $T_1 =$ `shop`, $T_2 =$ `customer`, $\mathcal{V}_1 = \{$`pin`, `pout`, `chair`$\}$, $\mathcal{V}_2 = \{$`pin`, `pwt`, `pcut`, `bsp`, `bwk`, `chair`$\}$; Π_1 is $\mathcal{A} \cup \mathcal{A}_{shop}$; Π_2 is $\mathcal{A} \cup \mathcal{A}_{cust}$; ϕ_1 and ϕ_2 are the temporal properties of the two transactions.

Definition 10 (Critical Tuples) *For any two transactions T_1 and T_2, the intersection set of their representative tuple sets, $\mathcal{V}_1$ and $\mathcal{V}_2$, is called* critical tuple set *$\mathcal{V}$, $\mathcal{V} = \mathcal{V}_1 \cap \mathcal{V}_2$, and the elements of the set called* critical tuples. ■

The critical tuple set of `shop` and `customer` is $\{$`pin`, `chair`$\}$. Tuple `pout` cannot be seen from transaction `customer`. Similarly, tuples `pwt`, `pcut`, and `bsp` cannot be seen by transaction `shop`.

If the tuples in the two transactions to be united are completely disjoint (subject to renaming), i.e., $\mathcal{V} = \emptyset$, the union is trivial. The behavior of the composed one is exactly the same as the behaviors of the two components; otherwise, the interference between the two transactions needs to be considered.

For any two transactions T_1 and T_2, let T be their union, i.e., $T \equiv T_1 \| T_2$, $\mathcal{V}$ the critical tuple set, and Π the temporal interpretation of T. To prove the correctness of the new transaction T, we need to proof that if $\mathcal{V} = \emptyset$, $\Pi \vdash \psi_1 \wedge \psi_2$; otherwise,

[1] *Π_1 and Π_2 here mean Π_1 and Π_2 together with the general temporal logical axioms*

$\Pi \vdash (\psi_1 \backslash \mathcal{V}) \wedge (\psi_2 \backslash \mathcal{V}) \wedge \psi$, where ψ is the new property of T. It is introduced by the union operation[2]. In the case of $\mathcal{V} = \emptyset$, i.e., no common tuples in the two transactions, each of the two transactions will keep its behavior unchanged when put together. The composition is trivial. The property of $T_1 \| T_2$ is exactly those of T_1's and T_2's, i.e., $\Pi \vdash \psi_1 \wedge \psi_2$. If $\mathcal{V} \neq \emptyset$, there will be interference between the two transactions. The interference comes from their critical tuples. For these properties which are not harmed by the critical tuples, such as $\psi_1 \backslash \mathcal{V}$ and $\psi_2 \backslash \mathcal{V}$, they are still held in the new transaction T. The others, which are affected by the interferences, are not the properties of T any more, except these happen to be implied by T again. As the result of the union, new property ψ is expected. Thus, we get $\Pi \vdash (\psi_1 \backslash \mathcal{V}) \wedge (\psi_2 \backslash \mathcal{V}) \wedge \psi$.

The properties of `shop` are $S = S_1 \wedge S_2$:

1. $S_1 \equiv \Box\Diamond$`pin`: there is always a new customer coming;

2. $S_2 \equiv$ `pout` $\Rightarrow \Diamond$`TRUE`: a customer will get out of the barber's shop after he has his hair cut.

Similarly, the properties of `customer` are $C = C_1 \wedge C_2$:

1. $C_1 \equiv$ `pin` $\Rightarrow \Diamond$`pcut` $\oplus \Diamond$`pwt`: a new customer will wake up the sleeping barber to have his hair cut, or wait if the barber is busy and there is a chair available;

2. $C_2 \equiv \Diamond\Box(|\texttt{chair}| = 0)$: the number of chairs will eventually become zero. The property is true from the point of view of `customer`, as the `chairs` are continuously consumed by new customers. It will be falsified by putting `customer` and `barber` together.

Finally, the properties of the transaction `barber` are $B = \bigwedge_{i=1}^{4} B_i$:

1. $B_1 \equiv$ `pcut` $\Rightarrow \Diamond$`pout`: the barber will finish his service on the customer who is having his hair cut;

2. $B_2 \equiv$ `pwt` $\Rightarrow \Diamond$`pcut`: a waiting customer will eventually have his hair cut;

[2]We won't discuss how to generally find out what the ψ is. A real ψ formula depends on the real transactions T_1, T_2, and T. We leave the problem to whoever conducts the verification practice, as we believe the property ψ is expected before the union operation.

3. $B_3 \equiv \texttt{bwk} \land \neg\texttt{pwt} \Rightarrow \Diamond\texttt{bsp}$: the barber is going to sleep if there are no waiting customers;

4. $B_4 \equiv \forall M, M > 0 : \Diamond(|\texttt{chair}| > M)$: the number of available chairs continuously increases, where M is an integer. The property is the counterpart of C_2.

The proof of these properties is not vary hard after we translate the transaction `shop`, `customer`, and `barber` to temporal logic formulae.

Let transaction `root` be the union of these three transactions,

$$\texttt{root} \;=\; \texttt{shop} \parallel \texttt{customer} \parallel \texttt{barber},$$

which can be worked out by

$$(\texttt{shop} \parallel \texttt{customer}) \parallel \texttt{barber} \;\text{ or }\; \texttt{shop} \parallel (\texttt{customer} \parallel \texttt{barber}).$$

As a result, the property C_2 and B_4 are refuted, because they are assertions on a critical tuple `chair`. The interferences introduced by the union establish a new assertion on the population of tuple `chair`—the number of the tuple `chairs` is between 0 and N, i.e., P_6 in Section 7.4. In addition to C_2 and B_4, all other properties of `shop`, `customer`, and `barber` are also needed to be re-examined if they contain critical tuples.

9.4 Superposition

Superposition contributes the layers, or hierarchical structures, to a T-Cham program. If transaction T_1 is built on T_2, i.e., T_2 is a sub-transaction of T_1, a superposition occurs. It is written as $T_1[T_2]$.

Superposition does not cause any problem in T-Cham verification. $T_1[T_2]$ means that there is at least one reaction rule in T_1 which has the form of

$$\cdots\ \texttt{leadsto}\ \cdots\ \texttt{by}\ T_2\ \cdots;$$

From the `subtransactions` section of T_1, we know the pre-condition and post-condition of T_2, say p and q respectively. No matter what kind of internal structures of T_2 is, the conditions are always fulfilled. In other words, T_2 has no effect in the temporal logic interpretation of transaction T_1. The property of T_1, say ψ_1, can be proven on p and

q. No details of how to get q from p are needed. While the property of T_2, ψ_2, can be proved within T_2 itself by its own logic interpretation. To be fitted into T_1, one major property of T_2 should be "$p \Rightarrow \Diamond q$".

The hierarchical structure of T-Cham provides an abstract view to a T-Cham program, and also an abstract view of the temporal logic interpretation of the program.

The Matrix Multiplication example in Section 8.4 has the hierarchical structure of `root[multi_matrix[multiply]]`. The temporal logic interpretation of the `root` transaction in Figure 8.8 is $\mathcal{A}_m$:

$$(\sigma, 0) \models \mathtt{A} \wedge \mathtt{B} \qquad \textit{(m0)}$$

$$\mathtt{A} \wedge \mathtt{B} \wedge \mathtt{p1} \Rightarrow \Diamond(\mathtt{C} \wedge \mathtt{q1}) \qquad \textit{(m1)}$$

`p1` and `q1` are listed in Section 8.4. The verification of the program in Figure 8.8 is straightforward: the condition that tuple `C` contains the matrix product of tuples `A` and `B` is directly implied by *(m1)*. In the proof, we do not need to consider the behavior of transaction `multi_matrix`. No matter what it is, it fulfills its pre-condition `p1` and post-condition `q1`; besides, it should terminate so that `q1` can be expected in a finite amount of time.

To ensure that transaction `multi_matrix` fulfills its `p1` and `q1`, we can translate it into temporal logic formulae and then verify `p1` and `q1` against the formulae. The verification is localized within the `multi_matrix` temporal proof system and does not affect the temporal proof systems of `root` or `multiply`.

9.5 Summary

In this chapter, we have discussed the issues of the compositional proof system of T-Cham programs. Two kinds of composition operators—union and superposition—and their effect on the T-Cham temporal logic proof system have been studied. As an example, we developed a transaction union version T-Cham program of the Sleeping Barber problem and studied its proof system under this union operation. In the real world, it is always desirable, in both program development and verification practice, to break down a large problem into a number smaller pieces, which would be much easier to manage. This is so-called divide-and-conquer strategy [5, 95]. With those two composition operators, T-Cham provides an easy way to realize the strategy. In addition, as each of the transactions to be composed are autonomous, any of them can be replaced by an equivalent transaction without the involvement of the others, T-Cham has a strong ability of code re-using.

[illegible] and always [illegible] of the temporal logic interpretation of the program.

The Matrix Multiplication program in Section 8.4 has the hierarchical structure of [illegible]. The temporal logic interpretation of the root transaction is [illegible]

$$\{A, B\} \Leftarrow A \wedge B \wedge [illegible] \quad (m0)$$
$$A \wedge B \wedge p1 \Rightarrow \Diamond(C \wedge q1) \quad (m1)$$

p1 and q1 are listed in Section 8.4. The verification of the program in Figure 8.8 is straightforward: the condition that tuple C contains the matrix product of tuples A and B is directly implied by (m1). In the proof, we do not need to consider the behavior of transaction [illegible]. No matter what it is, it fulfills its pre-condition p1 and post-condition q1. However, it should terminate so that q1 can be expected in a finite amount of time.

To ensure that [illegible] fulfills its p1 and q1, we can translate it into temporal logic formulae and then verify p1 and q1 against the formulae. The verification is localized within the [illegible] temporal proof system and does not affect the temporal proof [illegible].

9.5 Summary

In this chapter, we have discussed the issues of the compositional proof system of T-Chan programs. Two kinds of composition operators—union and superposition—and their effect on the T-Chan temporal logic proof system have been studied. As an example, we developed a transaction union version T-Chan program of the Sleeping Barber problem and studied its proof system under this union operation. In the real world, it is always desirable, in both program development and verification practice, to break down a large problem into a number smaller pieces, which would be much easier to manage. This is so-called divide-and-conquer strategy [5, 9]. With these

Chapter 10

Conclusion and Future Work

In this thesis, we proposed a new programming language: *T-Cham*. It is based on the Chemical Abstract Machine (Cham) model with the extension of transactions. Our work was carried out in three different but closely related directions: the design of the T-Cham programming language itself, its implementation prototype, and the verification of T-Cham programs.

The T-Cham programming language design is the major work of the thesis. It determines the other two directions. T-Cham is designed as a compromise between three critical but not always compatible criteria of a programming language. They are (i) programmability, (ii) efficient implementation, and (iii) a provable and simple underlying computational model.

The underlying computational model is the foundation of a programming language. It largely decides all the other aspects of the language. For example, the Turing machine decides that an imperative programming language, such as the C programming language, has assignment statements and control statements; the first order predicate logic decides that a Prolog program consists of facts and clauses; the λ calculus decides that in Lisp, both program and data are all in the list format. Some computational models are easier and more efficient to implement on a certain type of computer architectures than others, for example, the Turing machine on the von Neumann structure computers; while some other models have better provability, i.e., the programs written in those programming languages based on these kinds of models are easier to verify their correctness than others, for example, the first order predicate logic model and the λ calculus model. In the real world, unfortunately, a model which could be easily and efficiently implemented does not necessarily mean it has a good provability, and *vice versa*. Programmability is another very important issue. Taking the Turing machine and imperative programming languages as an example, three language structures, *sequence*, *branch* and *goto*, are theoretically enough for any kind of programming [57], but any practical programming language has much more features, for example, *block structure*, *procedure structure*, *data structure*, and *variable scope mechanism* etc., by which we call it good programmability. A good computational model itself does not mean good programmability just like a good foundation does not mean a good building, but with a good model, it is possible to design a programming language with good programmability.

T-Cham chooses the Chemical Abstract Machine (Cham) as its underlying computational model. The basic idea behind the choice is our belief that the difference between parallel and sequential programming does not lie on the single thread nature versus the multi-thread with communications but a functionality program versus a reactive one. T-Cham is an attempt to develop a programming language based on an interactive computational model. It tries to abandon the old concepts of shared memory or message passing at the programming level. Those concepts actually come from computer hardware architectures and should not play any roles at programming level. To the best of our knowledge, T-Cham is the first programming language completely built on an interactive underlying computational model.

T-Cham is also designed as an open paradigm programming language, i.e., more than one programming paradigm can be orthogonally integrated together (called coordination), to take the advantage of the current computer architectures. While the underlying skeletons of any T-Cham programs reflect the Chemical Abstract Machine model, the computational units (or chunks) of the programs can be written in any programming languages, which fit the situation best. Those computational units are not just normal functions or procedures, but with the enhancement of ACID (atomicity, consistency, isolation, and durability) properties to make them transactions. In other words, T-Cham can, on the one hand, take the advantage of the simplicity of Cham control structures, and on the other hand, with the help of conventional programming languages, be efficiently implemented on the current computer architectures. Transactions are the keys to simplify the coordination. A programmer does not have to worry about the order of the transactions to be executed, the places where the executions are carried out, nor the interference among those transactions.

In this thesis, we also proposed a T-Cham implementation model, the *T-Cham Machine*. It is an extension to the master/worker model. A T-Cham Machine can have more than one master to alleviate the communication congestion between the single master and the workers. Four basic algorithms, *Tuple Space Partition*, *Tuple Migration*, *Task Bid Handling*, and *Task Bidding and Receiving*, are developed to balance the task loads among the masters and guide task distributions to the workers. We undertook a prototype implementation of the T-Cham Machine on the AP1000 multicomputers and acquired some basic performance measurement data. The initial experience is very encouraging.

T-Cham implementation is a big job in the future. A full-fledged compiler and the related tools are expected to be developed as the next step. In addition, Y. P. Boglaev's chemical kinetics model [32, 31] may serve as a theoretical model for T-Cham Machine performance analysis and also worth further investigation.

The automatic verification of program correctness is always desirable. T-Cham is carefully designed to meet the requirement of easy formal verification. The ACID properties, the pre-, and post-conditions of transactions and the hierarchical transaction structures are among the considerations. With the understanding that a programmer may not be necessarily the same person who does the correctness verification, T-Cham keeps the theoretical part of the language in background. It can be ignored, but it is there and ready to use.

In this thesis, we also applied a formal temporal logic proof system to T-Cham program verification. The proof system was also tested on some examples. It demonstrated the significance of transactions and their pre- and post-conditions in the formal proof system. Meanwhile, we also realise that linear logic [79, 78, 166, 10] is a very hopeful candidate in dealing with the resource-like nature of T-Cham and should be seriously investigated.

In summary, as a newcomer to this crowded parallel programming language community, T-Cham has the following distinguishing characteristics:

1. T-Cham uses tuple spaces to coordinate a number of transactions. With this approach, parallel structures are a programmer's primary focus and then sequential tasks instead of, like most of other approaches, adding parallel facilities to a sequential structure. If we draw an analogy between programming and painting a picture, there are two radically different approaches. The first approach starts with the basic pieces of the picture and then put them together, like playing jig-saw games. The other approach starts from a big picture and then divides it to smaller pieces. Each of those pieces is refined independently. T-Cham belongs to the second approach. We believe it reveals more parallelism, and it also makes the synchronization among the pieces easier.

2. Hierarchical transaction and tuple space structures provide dynamic and abstract views to transactions and their tuple spaces. They mean program modularity. Keeping the analogy we discussed previously, the big picture is divided into smaller pieces. Each of the pieces is treated the same way as we treat the big picture itself. In other words, those pieces are continuously divided into even smaller pieces until they reach the right sizes. Different layers give different details of the picture, in programming language terminology, different abstract views. Taking the daily news as an example, the level could be the global news, national news, local news, or even the news of a particular family talking at their dining table.

3. In T-Cham, the issues of sequential, parallel, or distributed are not at the programming language level because we believe they belong to the issues of a program implementation on different computation resources. From the point of view of a T-Cham program, only reactions exist. A program is a kind of logic specification. It describes the logic relations among its components. How to realize the relations is a matter of implementation and has nothing to do with the relations themselves. For example, the course dependence in a university's curriculum

is the logic relations and is related to programming level, while the order of a particular student taking those courses belongs to the implementation issues.

4. T-Cham is a high-level portable programming language: a programmer does not have to know the architecture of the underlying computer: *parallel*, *sequential*, or *networked*. Taking the university curriculum example again, when specifying the course dependence relations, there is no need to know the case of each individual student.

5. Multi-lingual transactions make T-Cham multi-paradigm. A programmer can take the advantages of different program languages without worrying about their interferences as they are integrated orthogonally by the tuple spaces.

6. Transaction granularity can be easily adjusted by changing the number of operations contained in the transaction. For example, a transaction can be a very complex function (coarse-grain), or only a simple summation operation (fine-grain). The changing of one transaction is isolated from the others.

7. The temporal logic proof system provides a formal tool for T-Cham program verification. We keep this proof system in the background. It can be ignored if a programmer does not like this theoretical part, but if he/she does want to formally verify the programs, it can be used. The translation of a T-Cham transaction to its correspond temporal logic formulae is straightforward and can be achieved semi-automatically by algorithms.

The work presented in the thesis is only the first step towards this direction. More research work and investigation should be done in the future. But, we do expect that T-Cham can lead us to a new way of separating (i) the logic of a program from its implementation, (ii) correctness from efficiency, and (iii) the rigid formal reasoning aspect from a comfortable intuitive presentation, without heavy penalties on execution efficiency.

Appendix A

The BNF Definition of T-Cham Syntax

Program	→	{*NonLeafTrans* | *LeafTrans*}+
NonLeafTrans	→	`transaction` *NName NBody* `endtrans`
NName	→	*PlainIdentity*
NBody	→	*Tuples Init React* [*Term*] [*SubTrans*]
Tuples	→	`tuples` *TupleDcl* {*TupleDcl* | *MaskDcl*}
TupleDcl	→	[*DclHead*] *DclBody*
DclHead	→	`fifo` | `filo` | `random`
DclBody	→	*Type NameList* ;
Type	→	[`tuple`] *SimpleType* | *StructType*
SimpleType	→	`integer` | `real` | `boolean` | `char`
StructType	→	`tuple` "{" {*ComponentDcl*} + "}"
ComponentDcl	→	*DclBody*

NameList	→	*Name* { , *Name* }
Name	→	*SimpleName* | *ArrayName*
SimpleName	→	*PlainIdentity*
ArrayName	→	*PlainIdentity Dimension*
Dimension	→	“[” *Digits* “]” { “[” *Digits* “]” }
MaskDcl	→	`mask` “{” { *MaskComponent* } + “}” *MaskName*
MaskComponent	→	“{” *MaskWindow* {, *MaskWindow* } “}” “ ∗ ” *SimpleType*
MaskWindow	→	{ 0 | 1 | *Digits* “ ∗ ” (0 | 1) }
MaskName	→	*PlainIdentity*
Init	→	`initialization` *InitList*
InitList	→	{*ActvInit* | *PassInit* | *MappingInit*}
ActvInit	→	*LName*
PassInit	→	[*IdxRange* ::] *SimpleName* = *Value*
IdxRange	→	“[” *Digits* .. *Digits* “]”
MappingInit	→	*DataMask* < − − *SimpleName*
DataMask	→	“[(” *PlainMask* | *RegularMask* “)]”
PlainMask	→	*SimpleName* \ *MaskName* {, *SimpleName* \ *MaskName* }
RegularMask	→	*MappingPattern* “ ∗ ”*SimpleName*[˜ *Digits*]
MappingPattern	→	*Digits* {“ ∗ ” *Digits*}
Value	→	*SimpleValue* | *ValueList*
SimpleValue	→	*ValueExp*
ValueList	→	“{” *ValueExp* {, *ValueExp*} “}”
ValueExp	→	*ValueExp Op ValueExp* | *ValueExp ROp ValueExp* | *ValueExp BOp ValueExp* | “−” *ValueExp* | “!” *ValueExp* | *SimpleValue*

```
SimpleValue        →  IntegerValue | RealValue | BooleanValue | CharValue
IntegerValue       →  [+ | −] Digits
RealValue          →  [+ | −] Digits[ . Digits [ E [ + | − ] Digits ] ]
BooleanValue       →  TRUE | true | FALSE | false | 1 | 0
CharValue          →  "'"(Letter | Digit )"'"

React              →  reactionrules {ReactRule}+
ReactRule          →  LHS leadsto RHS [by Trans] [when BExp] ;
LHS                →  SimpleTupleList
RHS                →  SimpleTupleList
SimpleTupleList    →  Tuple{, Tuple}
Tuple              →  PlainIdentity

Trans              →  SimpleTransName | OnLineTrans
SimpleTransName    →  PlainIdentity
OnLineTrans        →  "{" {AssignmemtStmt} + "}"
AssignmemtStmt     →  VarName = Exp ;
VarName            →  SimpleVarName | ArrayVarName
SimpleVarName      →  FullIdentity
ArrayVarName       →  FullIdentity Dimension
Exp                →  Exp Op Exp | Exp ROp Exp | Exp BOp Exp | "−" Exp |
                      "!" Exp | VarName | ValueExp | |FullIdentity|[']

BExp               →  BExp ROp BExp | BExp BExp BExp |
                      "!" BExp | FunctionCall | VarName
FunctionCall       →  PlainIdentity ( {VarNameList} )
VarNameList        →  VarName { , VarName }

Term               →  terminination {TermStmt}+
TermStmt           →  on ( BExp ) do (Trans | AssembleData);
AssembleData       →  DataMask − − > SimpleName
```

SubTrans	→	`subtransactions` {*TransStmt*}+
TransStmt	→	*Trans* : *PreCond* // *PostCond* ;
PreCond	→	*PlainIdentity*
PostCond	→	*PlainIdentity*
LeafTrans	→	`transaction` *LName* *LBody* `endtrans`
LName	→	*PlainIdentity*
LBody	→	*Micros* *BodyCode*
Micros	→	`#language` *LangName* [`#tuplein` *SimpleTupleList*]
		[`#tupleout` *SimpleTupleList*]
Op	→	+ | − | "*" | /
ROp	→	> | < | >= | <= | == | !=
BOp	→	&& | "||"
FullIdentity	→	*PlainIdentity* | *QualifiedIdentity*
QualifiedIdentity	→	*PlainIdentity* $ *Digits*
PlainIdentity	→	*Letter* {*Letter* | *Digit*}
Letter	→	`A` | `B` | ··· | `Z` | `a` | `b` | ··· | `z`
Digits	→	{*Digit*}+
Digit	→	`0` | `1` | ··· | `9`

Appendix B

The AP1000 Multicomputer

This chapter is adapted from the CAP (Collaborative Research Project) documents. For more details, please look at `http://cafe.anu.edu.au`.

The AP1000 is an experimental large-scale MIMD parallel computer developed by Fujitsu Laboratories, Japan. Configurations range from 64 to 1024 individual processors or *cells*, connected by three separate high-bandwidth communications networks: the *B-net*, *T-net* and *S-net*, Figure B.1. The cells do not share memory. The AP1000 is connected to and controlled by a host computer which is typically a Sun SPARCServer.

Each cell consists of a SPARC processor running at 25MHz with 16MB of RAM (four-way interleaved, with ECC), 128KB of direct-mapped cache memory, floating-point unit, a message controller, and interfaces to the three communications networks. The cells have no address translation hardware, but they do have a *memory-protection table* (MPT) which provides access and caching control for 4KB pages. The message controller provides DMA facilities for sending and receiving messages.

The B-net is a 32-bit, 50MB/sec broadcast network which connects all cells and the host, and is used for communication between the host and the cells. Using the B-net, the host can transmit data to one cell or to all cells simultaneously, and each cell can transmit data to the host. The B-net also supports scatter/gather operations.

The T-net provides cell-to-cell communication. It is arranged as a two-dimensional

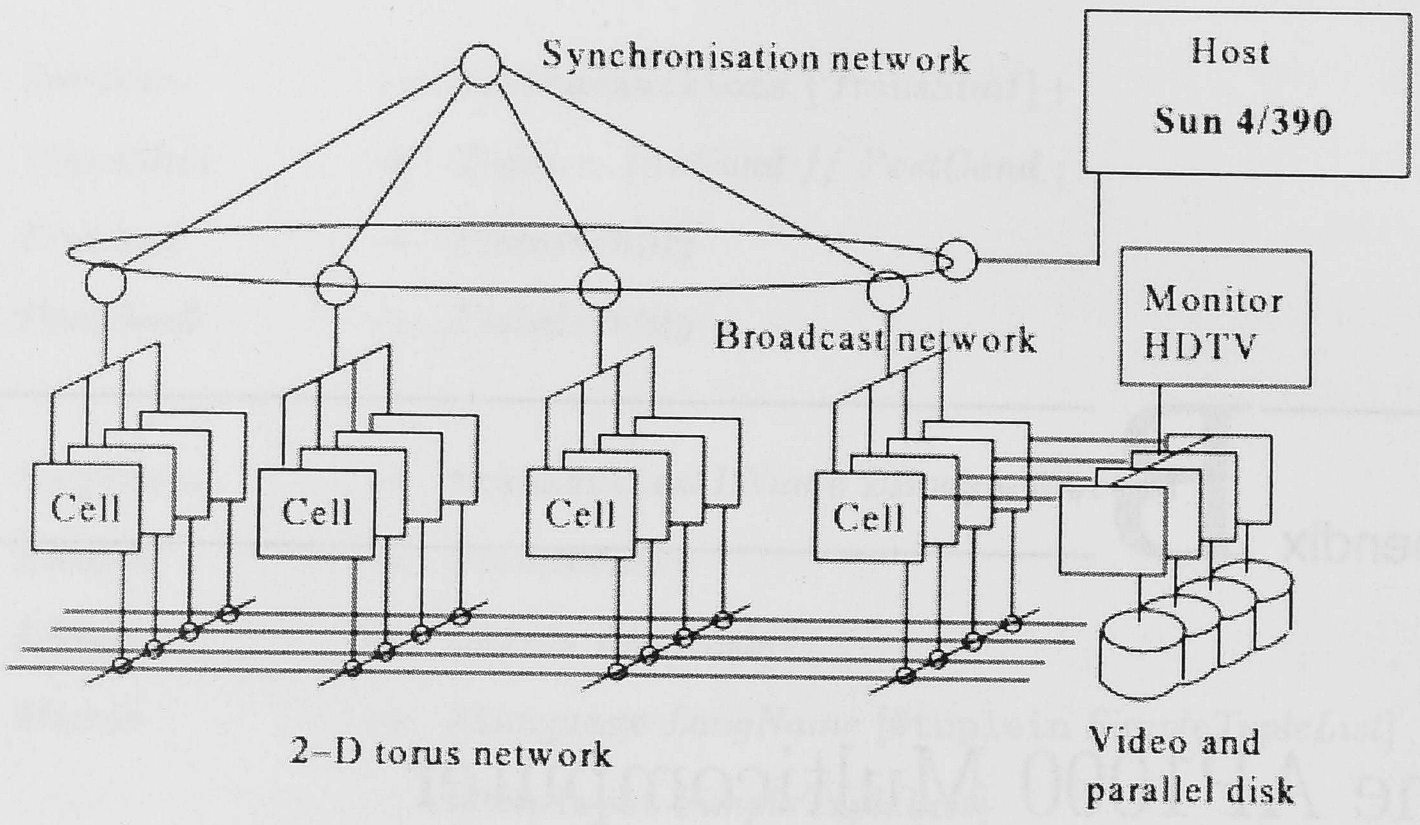


Figure B.1: The Architecture of the AP1000 Multicomputer

torus in which each cell has links to its four neighbours in a rectangular grid. The T-net is controlled by a *Routing Controller* (RTC) chip in each cell; its bandwidth is 25MB/sec on each link. Wormhole routing is used, with a structured buffer pool in each RTC to avoid deadlock. Each RTC will forward messages on toward their destinations without requiring any action by the cell CPU.

The S-net supports synchronization and status checking. It carries 40 signals which are `AND`ed together over all cells: all cells output a value for each signal, and receive the `AND` of the values output by all cells. To achieve synchronization, each cell sets a particular output to 1 and then waits until it sees the corresponding input at a 1. The S-net also allows the cells to synchronize with the host.

Programs for the AP1000 are written in either C or FORTRAN. Library calls are used for communication over the networks described above. Run-time control is provided in the cells by *CellOS*, the cell operating system, and on the host by the *CAREN* (Cellular Array Runtime ENvironment) program. CAREN provides facilities for run-time monitoring of cell activity, performance measurement, error logging and debugging. Symbolic debugging of cell tasks is provided through the use of GDB.

Bibliography

[1] S. Abramsky. The collection works of S. Abramsky. http://www.dcs.ed.ac.uk/home/samson.

[2] S. Abramsky. Game semantics for programming languages. In I. Privara and P. Ruzicka, editors, *Proceedings of the 22nd International Symposium on Mathemtical Foundations of Computer Science*, LNCS 1295, pages 3–4. Springer-Verlag, 1997.

[3] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proceedings of the Thirteenth International Symposium on Logic in Computer Science*, pages 334–344. IEEE Computer Society Press, 1998.

[4] D. Agrawal and A. El Abbadi. Transaction management in database systems. In Ahmed K. Elmagarmid, editor, *Database Transaction Models: For Advanced Applications*, pages 1–32. Morgan Kaufmann Publishers, San Mateo, California, USA, 1992.

[5] A. V. Aho. *The Design and Analysis of Computer Algorithm.* Addison-Wesley, 1974.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[7] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, pages 26–34, August 1986.

[8] H. Aït-Kaci. The WAM: A (real) tutorial. Technical report, digital, Paris Research Laboratory, 85 Avenue Victor Hugo, 92563 Rueil Malmaison Cedex, France, January 1990.

[9] S. G. Aki. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[10] V. Alexiev. Applications of linear logic to computation: An overview. *Bull. of the IGPL*, 2(1):77–107, 1994.

[11] G. R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[12] ANSI. *American National Standard Programming Language: PL/I.* ANSI, 1976.

[13] ANU and Fujitsu. CAP homepage. `http://cafe.anu.edu.au`.

[14] W. B. Arthur and R. Alice. The ENIAC: First general-purpose electronic computer. *Annals of the History of Computing*, 3(4):310–399, October 1981.

[15] W. Aspray. Computing before computers. Iowa State University Press, 1990.

[16] J. Backus. Can programming be liberated from the von neumann style? *Comm. the ACM*, 21(8):613–641, August 1978.

[17] J. Bacon. *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems.* Addison-Wesley, 1993.

[18] H. E. Bal. A comparative study of five parallel programming languages. *Future Generation Computer Systems*, 8:121–135, 1992.

[19] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[20] J.-P. Banâtre. Parallel multiset processing: From explicit coordination to chemical reaction. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, First International Conference, COORDINATION'96*, LNCS 1061, pages 1–11. Springer-Verlag, 1996.

[21] J.-P. Banâtre, A. Coutant, and D. Le Métayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer System*, 4:133–144, 1988.

[22] J.-P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[23] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Comm. ACM*, 36(1):98–111, January 1993.

[24] H. Barringer. The use of temporal logic in the compositional specification of concurrent systems. In A. Galton, editor, *Temporal Logics and Their Applications*, pages 53–90. Academic Press, 1987.

[25] A. Beguelin, J. Dongarra, A. Geist, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *Computer*, pages 88–95, June 1993.

[26] A. Beguelin and G. Nutt. Visual parallel programming and determinacy: A language specification, an analysis technique, and a programming tool. *Journal of Parallel and Distributed Computing*, (22):235–250, 1994.

[27] T. Bemmerl and P. Braun. Visualization of message passing parallel programs with the TOPSYS parallel programming environmanet. *Journal of Parallel and Distributed Computing*, (18):118–128, 1993.

[28] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Publishing Company, 1987.

[29] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[30] R. Bjornson, N. Carriero, and D. Gelernter. From weaving threads to untangling the web: A view of coordination from Linda's perspective. In D. Garlan and D. Le Métayer, editors, *Coordination Languages and Models, Second International Conference, COORDINATION'97*, LNCS 1282, pages 1–17. Springer-Verlag, 1997.

[31] Y. P. Boglaev. Exact dynamic load balancing of MIMD architectures with linear programming algorithms. *Parallel Computing*, 18:615–523, 1992.

[32] Yuri P. Boglaev. Chemical kinetics structure of parallel computing. In S. K. Aityan, L. T. Hathaway, et al., editors, *Proceedings of the First International Conference on Neural, Parallel, and Scientific Computations*, pages 53–58. Dynamic Publishers, Inc., Atlanta, Georgia, USA, 1995.

[33] A. P. W. Böhm and R. R. Oldehoeft. Two issues in parallel language design. *ACM Transactions on Programming Languages and Systems*, 16(6):1675–1683, November 1994.

[34] G. Boudol. Some chemical abstract machines. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, LNCS 803, pages 92–123. Springer-Verlag, 1993.

[35] P. Branguart, J. Lewis, M. Sintzoff, and P. L. Wodor. The composition of semantics of Algol68. *Comm. ACM*, 16, 1971.

[36] A. G. Bromley. Difference and analytical engines. In W. Aspray, editor, *Computing Before Computers*. Iowa State University Press, 1990.

[37] BSP. The collection works related to BSP. `http://web.comlab.ox.ac.uk`.

[38] J. P. Burgess. Basic tense logic. In D. M. Gabbay and F. Guethner, editors, *Handbook of Philosophical Logic, Vol. II*, pages 89–134. D. Reidel Publishing Company, 1984.

[39] P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, September 1994.

[40] N. Carriero and D. Gelernter. Linda in context. *Comm. ACM*, 32(4):444–458, 1989.

[41] N. Carriero and D. Gelernter. *How to Write Parallel Program: A First Course*. The MIT Press, 1990.

[42] N. Carriero, D. Gelernter, T. Mattson, and A. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20:632–655, 1994.

[43] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[44] M. Chaudron and E. Jong. Towards a compositional method for coordinating Gamma programs. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, First International Conference, COORDINATION'96*, LNCS 1061, pages 107–123. Springer-Verlag, 1996.

[45] B. F. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.

[46] D. Y. Cheng. A survey of parallel programming languages and tools. Technical Report RND–93–005, Computer Sciences Corporation, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA, March 1993.

[47] P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12:251–283, 1994.

[48] P. Ciancarini, R. Gorrieri, and G. Zavattaro. An alternative semantics for the calculus of Gamma programs. In J.-M. Andreoli, C. Hankin, and D. Le Métayer, editors, *Coordination Programming: Mechanism, Models and Semantics*, pages 224–247. IC Press, London, 1996.

[49] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, LNCS 803, pages 125–175. Springer-Verlag, 1993.

[50] E. M. Clarke, O. Grumberg, H. Hiraishi, et al. Verifying of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and Their Applications*. North-Holland, 1993.

[51] N. Cohen. *Ada as a Second Language*. McGraw-Hill, Inc., 1986.

[52] R. Cohen and B. Molinari. Implementation of C-Linda for the AP1000. In *The Proceedings of the Second ANU/Fujitsu CAP Workshop*. 1991.

[53] D. J. Cooke and H. E. Bez. *Computer Mathematics*. Cambridge University Press, 1984. Cambridge Computer Science Texts–18.

[54] Thinking Machines Corporattion. CMMD reference manual. Technical report, Thinking Machines Corporation, Cambridge, Massachusetts, USA, 1993.

[55] Thinking Machines Corporattion. CMMD user's guide. Technical report, Thinking Machines Corporation, Cambridge, Massachusetts, USA, 1993.

[56] H. B. Curry. *Foundations of mathematical logic*. McGraw-Hill, 1963.

[57] O.-J. Dahl, E. W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press, 1972.

[58] C. J. Date. *An introduction to database systems*, volume 1. Addison-Wesley Publishing Company, 1990.

[59] C. J. Date. *An introduction to database systems*, volume 2. Addison-Wesley Publishing Company, 1990.

[60] R. David and H. Alla. *Petri Nets and Grafcet: Tools for modelling discrete event systems*. Printice Hall, 1992.

[61] J. B. Dennis. Machines and models for parallel computing. *International Journal of Parallel Programming*, 22(1):47–77, 1994.

[62] J. B. Dennis, G. R. Gao, and K. W. Todd. Modeling the weather with dataflow supercomputers. *IEEE Transactions on Computer*, 33:592–603, 1984.

[63] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic dataflow processor. In *Proceedings of 2nd Symposium on Computer Architectures*, pages 126–132, 1975.

[64] Yale University Department of Computer Science. Linda group homepage. `http://www.cs.yale.edu/Linda/linda.html`.

[65] E. W. Dijkstra. Co-operating sequential processes. In F. Genyus, editor, *New Programming Languages*, pages 43–112. Academic Press, 1968.

[66] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[67] I. O. Elliot. *The Multics System — An Examination of its Structure*. The MIT Press, 1972.

[68] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computer*, 21:948–960, 1972.

[69] I. Foster and C Kesselman. Language constructs and runtime systems for compositional parallel programming. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 — VAPP VI*, LNCS 854, pages 5–16. Springer-Verlag, 1994.

[70] I. Foster, C. Kesselman, and S. Taylor. Concurrency: Simple concepts and powerful tools. *The Computer Journal*, 33(6):501–507, 1990.

[71] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: the PCN approach. *Scientific Programming*, 1(1):51–66, 1992.

[72] I. Foster and R. Overbeek. Bilingual parallel programming. In A. Nicolau, D. Gelernter, T. Gross, et al., editors, *Advances in Languages and Compilers for Parallel Processing*, pages 24–43. The MIT Press, 1991.

[73] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1990.

[74] G. C. Fox. Applications of parallel supercomputers: Scientific results and computer science lessons. In M. A. Arbib and J. A. Robinson, editors, *Natural and Artificial Parallel Computation*, pages 47–90. The MIT Press, 1990.

[75] N. Francez. *Fairness*. Springer-Verlag, 1986.

[76] D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE'89, Parallel Architectures and Languages Europe*, LNCS 366, pages 20–27. Springer-Verlag, 1989.

[77] D. Gelernter and N. Carriero. Coordination languages and their significance. *Comm. ACM*, 35:96–107, 1992.

[78] J. Girard. Linear logic: A survey. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 63–112. Springer-Verlag, 1991.

[79] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[80] A. Goldberg and D. Robson. *Smalltalk80: The Language and its Implementation.* Addison-Wesley, 1983.

[81] A. J. Goldberg and L. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.

[82] M. J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag, 1979.

[83] R. Gotzhein. Temporal logic and application — A tutorial. *Computer Networks and ISDN systems*, 24:203–218, 1992.

[84] C. Hankin, D. Le Métayer, and D. Sands. A calculus of GAMMA programs. LNCS 757. Springer-Verlag, 1992.

[85] C. Hankin, D. Le Métayer, and D. Sands. Refining multiset transformers. *Theoretical Computer Science*, 192:233–258, 1998.

[86] P. B. Hansen. *Operating Systems Principles.* Prentice-Hall, Inc., 1973.

[87] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985.

[88] P. Henderson. *Functional Programming, Application and Implementation.* Prentice-Hall, Inc., 1980.

[89] J. M. D. Hill and D. B. Skillicorn. Lessons learned from implementing BSP. Technical Report Technical Report TR-96-21, Oxford University Computing Laboratory, 1996.

[90] C. A. R. Hoare. Monitors: An operating system structuring concept. *Comm. the ACM*, 17(10):549–557, October 1974.

[91] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, August 1978.

[92] R. W. Hockney and C. R. Jesshope. *Parallel Computers: architecture, programming and algorithms.* Adam Hilger Ltd, Techno House, Redcliffe Way, Bristol BS1 6NX, UK, 1981.

[93] C. J. Hogger. *Introduction to Logic Programming.* Academic Press, 1984.

[94] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computations.* Addison-Wesley, 1979.

[95] E. Horowitz. *Fundamentals of Computer Algorithm.* Computer Science Press, 1978.

[96] P. Hudak. The conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[97] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., 1993.

[98] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[99] K. Jensen. An introduction to the theoretical aspects of coloured Petri nets. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, LNCS 803, pages 230–272. Springer-Verlag, 1993.

[100] C. W. Johnson. personal communication.

[101] D. E. Knuth. Von Neumann's first computer program. *ACM Computing Surveys*, 2(4):247–260, December 1970.

[102] P. M. Kogge. *The Architecture of Symbolic Computers*. McGraw-Hill, Inc., 1991.

[103] J. S. Kowalik and K. W. Neves. Software for parallel computing: Key issues and research directions. In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, pages 3–33. Springer-Verlag, 1993.

[104] E. V. Krishnamurthy and V. K. Murthy. Parallel programming paradigm based on multiset transformation. In David Arnold, Ruth Christie, et al., editors, *Parallel Computing and Transputers, PCAT–93*, pages 43–51. IOS Press, Amsterdam, Netherlands, 1993.

[105] F. Kröger. *Temporal Logic of Programs*. Springer-Verlag, 1987.

[106] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In Duff and Stewart, editors, *Sparse Matrix Proceedings*, pages 47–90. Knoxville, TN, SIAM, 1978.

[107] C. LaMorte and J. Lilly. Computers: History and development. In *Jones Telecommunications and Multimedia Encyclopedia*. 1997. `http://www.digitalcentury.com/encyclo/update/comp-hd.html`.

[108] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.

[109] L. Lamport. What good is temporal logic. In R. E. A. Mason, editor, *Information Processing, IFIP 83*, pages 657–668. Elsevier Science Publishers B. V., North-Holland, 1983.

[110] L. Lamport. A simple approach to specifying concurrent systems. *Comm. ACM*, 32(1):32–45, January 1989.

[111] L. Lamport. Verification and specification of concurrent programs. In J. W. de Bakker, W. P. de Rover, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, LNCS 803, pages 347–374. Springer-Verlag, 1993.

[112] L. Lamport. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):827–923, May 1994.

[113] J. V. Leeuwen. *Handbook of Theoretical Computer Science: Formal Models and Semantics*, volume B. Elsevier, The MIT Press, 1990.

[114] H. R. Lewis and C. H. Papadimitrion. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[115] C. Lin and L. Snyder. Data ensembles in orca c. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, LNCS 757, pages 112–123. Springer-Verlag, 1992.

[116] B. Liskov. Distributed programming in Argus. *Comm. the ACM*, 31(3):300–312, March 1988.

[117] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. on Programming Languages and Systems*, 5(3):381–404, July 1983.

[118] J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, second edition, 1987.

[119] Inmos Ltd. *Reference Manual of Transputer.* Prentice-Hall, 1987.

[120] Inmos Ltd. *A Tutorial Introduction to Occam Programming.* Prentice-Hall, 1987.

[121] Inmos Ltd. *Occam 2 Reference Manual.* Prentice-Hall, 1988.

[122] W. Ma, , M. A. Orgun, and C. W. Johnson. Towards a temporal semantics for Frame. In *Proceedings SEKE'98, Tenth International Conference on Software Engineering and Knowledge Engineering*, pages 44–51. Knowledge Systems Institute, 3420 Main Street, Skokie, IL 60076, USA, 1998.

[123] W. Ma, C. W. Johnson, and R. P. Brent. Concurrent programming in T-Cham. In Kotagiri Ramamohanarao, editor, *The Proceedings of the 19th Australasian Computer Science Conference (ACSC'96)*, pages 291–300. 1996.

[124] W. Ma, C. W. Johnson, and R. P. Brent. Programming with transactions and chemical abstract machine. In Guo-Jie Li, D. F. Hsu, S. Horiguchi, and B. Maggs, editors, *Proceedings of Second International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'96)*, pages 562–564. 1996.

[125] W. Ma, E. V. Krishnamurthy, and M. A. Orgun. On providing temporal semantics for the GAMMA programming model. In C. Barry Jay, editor, *CATS: Proceedings of Computing: the Australian Theory Seminar*, pages 121–132. University of Technology, Sydney, Australia, 1994.

[126] W. Ma, V. K. Murthy, and E. V. Krishnamurthy. Multran — A coordination programming language using multiset and transactions. In S. K. Aityan, L. T. Hathaway, et al., editors, *Proceedings of the First International Conference on Neural, Parallel, and Scientific Computations*, pages 301–304. Dynamic Publishers, Inc., Atlanta, Georgia, USA, 1995.

[127] W. Ma and M. Orgun. Verifying Multran programs with temporal logic. In M. Orgun and E. Ashcroft, editors, *Intensional Programming I*, pages 186–206. World-Scientific, 1996.

[128] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

[129] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proc. 10th Ann. ACM Symp. on Principles of Programming Lang.*, pages 141–154. ACM Press, 1983.

[130] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[131] G. Matos and J. Purtilo. Reconfiguration of hierarchical tuple space: Experiments with linda polylith. Technical Report CS-TR-3153, Department of Computer Science, University of Maryland, Maryland, USA, 1993.

[132] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[133] Sun Microsystems. Java homepage. `http://www.javasoft.com/`.

[134] M. P. Miller. What to draw? When to draw? An essay on parallel program visualization. *Journal of Parallel and Distributed Computing*, (18):265–269, 1993.

[135] R. Miller and Q. F. Stout. *Parallel Algorithms for Regular Architectures*. The MIT Press, 1989.

[136] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[137] R. Milner. The polyadic π-calculus: a tutorial. Technical Report Technical Report ECS-LFCS-91-180, Department of Computer Science, University of Edinburgh, 1991.

[138] R. Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, NATO ASI Series, pages 203–246. Springer-Verlag, 1993.

[139] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992.

[140] US Department of Defence. *Reference Manual for the Ada Programming Language*. US Department of Defence, 1983.

[141] M. Orgun and W. Ma. An overview of temporal and modal logic programming. In Dov M. Gabbay and H. J. Ohlbach, editors, *The First International Conference on Temporal Logic*, LNAI 827, pages 445–479. Springer-Verlag, 1994.

[142] C. M. Pancake. Software support for parallel computing: Where are we headed? *Comm. the ACM*, 34(11):53–64, November 1991.

[143] G.-R. Perrin and J.-P. Finance. Communication relations: a paradigm for parallel program design. *Science of Computer Programming*, 19:25–59, 1992.

[144] R. H. Perrott. Parallel languages and parallel programming. In D. J. Evans, G. R. Joubert, and F. J. Peters, editors, *Parallel Computing 89*, pages 47–58. Elsevier Science Publishers B. V., 1990.

[145] J. L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, 1981.

[146] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, German, 1962.

[147] W. Reisig. *Petri Nets.* Springer-Verlag, 1985.

[148] N. Rescher and A. Urquhart. *Temporal Logic.* Springer-Verlag, 1971.

[149] M. Reynolds. Temporal semantics for Gamma. In J.-M. Andreoli, C. Hankin, and D. Le Métayer, editors, *Coordination Programming: Mechanism, Models and Semantics*, pages 141–170. IC Press, London, 1996.

[150] E. S. Richard. A history overview of computer architectures. *Annals of the History of Computing*, 10:277–303, 1988.

[151] G.-C. Roman and K. C. Cox. A taxonomy of program visualization system. *Computer*, pages 11–24, December 1993.

[152] G.-C. Roman and H. C. Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Transactions on Software Engineering*, 16(12):1361–1373, December 1990.

[153] G.-C. Roman and H. C. Cunningham. Reasoning about synchronic groups. In J. P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, LNCS 574, pages 21–38. Springer-Verlag, 1991.

[154] M. Rosing, R. B. Schnabel, and R. P. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, (13):30–42, 1991.

[155] J. Rumbaugh, M. Blaha, W. Premerlani, et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[156] T. Shimizu, T. Horie, and H. Ishihata. Low-latency message communication support for the AP1000. In *Proceedings of the Second Fujitsu-ANU CAP Workshop*, pages 1–14. The Australian National University, 28–29, November 1991.

[157] D. B. Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 22(2):133–158, 1992.

[158] D. L. Slotnick, W. C. Borck, and R. C. McReynolds. The SOLOMON computer. In *AFIPS Conf. Proc., 22*, pages 97–107. 1962.

[159] V. P. Srini. An architecture comparison of dataflow systems. *Computer*, pages 68–88, March 1986.

[160] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[161] A. S. Tanenbaum. A tutorial on Algol68. *ACM Computing Surveys*, 8(2), 1970.

[162] E. Tick. *Parallel Logic Programming.* The MIT Press, 1991.

[163] A. Turing. On computable numbers, with an application to the entscheidungs problem. In *Proceedings of the London Mathematical Soceity*, volume 42, pages 230–265. 1936. reprinted in M. David (ed.), *The Undecidable*, Hewlett, NY, Raven Press, 1965.

[164] L. G. Valiant. A bridging model for parallel computation. *Comm. the ACM*, 33(8):103–111, August 1990.

[165] R. van der Goot, J. Schaeffer, and G. V. Wilson. Safer tuple spaces. In D. Garlan and D. Le Métayer, editors, *Coordination Languages and Models, Second International Conference, COORDINATION'97*, LNCS 1282, pages 289–301. Springer-Verlag, 1997.

[166] P. Wadler. A taste of linear logic. In A. M. Borzyszkowski and S. Sokoloski, editors, *Mathematical Foundations of Computer Science*, LNCS 714, pages 185–210. Springer-Verlag, 1993.

[167] D. H. D. Warren. An abstract Prolog instruction set. Technical Report Technical Note 309, SRI International, Menlo Park, CA, USA, October 1983.

[168] P. Wegner. Why interaction is more powerful than algorithm. *Comm. ACM*, 40(5):89–91, May 1997.

[169] M. Weichert. Pipelining the molecule soup: A plumber's approach to Gamma. In P. Ciancarini and A. L. Wolf, editors, *Coordination Languages and Models, Third International Conference, COORDINATION'99*, LNCS 1594, pages 69–84. Springer-Verlag, 1999.

[170] S. E. Zenith. A rationale for programming with ease. In J. P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, LNCS 574, pages 147–156. Springer-Verlag, 1991.

[171] K. Zhang and W. Ma. Graphical assistance in parallel program development. In Allen L. Ambler and Takayuki Dan Kimura, editors, *Proceedings of IEEE Symposium on Visual Languages*, pages 168–170. IEEE Computer Society Press, 1994.

Index

F function, 96
G function, 96
π-calculus, 31

abacus, 9
ACID, 45
Argus, 23
axiom, 111

bid hit rate, 98
bid-choice function, 96
Blaise Pascal, 9

CCS, 31
Cham, 4, 16, 18, 43
Charles Babbage, 9
Chemical Abstract Machine, *see* Cham
Communicating Sequential Processes, *see* CSP
Communication and Concurrent System, *see* CCS
completeness, 111
control flow, 37
control thread, 37
coordination, 5, 44
CSP, 29

deductive system, 111
DINO, 27
direct bid, 92
distributed memory, 38
duplication (tuple space), 88

EDVAC, 10
ENIAC, 9
event interleaving, 135
example
    Dining Philosophers, 126
    Dutch Flag, 70
    Element Summation, 57
    Eratosthenes Sieve, 64
    Fibonacci Number, 75
    Matrix Multiplication, 151
    Meeting Scheduler, 74
    Producer-Consumer Problem, 68
    Sleeping Barber, 72
    Sleeping Barber: the union version, 158
    The Calculation of the Value of π, 79

Vending Machine, 66
executor, *see* task executor

F function, 99, 102
fair, 59
fairness, 52, 135
functional programming, 35
functionality program, 2, 35

G function, 99
GAMMA Model, 16
granularity, 39
coarse-grain, 39
fine-grain, 39
grain packing, 39

hierarchical tuple space structure, 141

inference rules, 111
instance reference, 55
isoefficiency, 98

Linda, 18
linear logic, 40
logic programming, 35
logical shared memory, *see* tuple space

manager, *see* task manager
mask, 148
plain mask, 148
regular mask, 150
maximum common tuple group, 122
MCTG, 122
MIMD, 11
MISD, 11
modal logic, 108

no race under race condition, 123
NRURC, 123

Occam, 31

Parallel Random Access Memory, *see* PRAM
partition (tuple space), 88
passing bid, 92
PCN, 25
Petri Nets, 14
PRAM, 31
processes, 10

race, 117
maximum common tuple group, 122
MCTG, 122
no race under race condition, 123
NRURC, 123
race (the definition), 123
race condition, 122
reactive program, 36
reactive system, 3
representative set, 115
resources, 39
return-choice function, 96

satisfy, 111
shared memory, 38
SIMD, 11
SISD, 11
soundness, 111

state formulae, 109
state predicate, 109
state term, 109
Strand, 25
stuttering, 116
Swarm, 20

T-Cham, 4–6, 20, 26, 42–46
T-Cham Machine, 85
task bid, 90
task bid handling algorithm, 94
task bidding and receiving algorithm, 95
task distribution, 95
task executor, 86
task manager, 86
TaskBidHist array, 92, 96, 98, 99, 101
temporal formula, 109
temporal logic, 46, 108
temporal operators, 109
temporal property, 108
  liveness, 128
  safety, 128
theorem, 111
thread, 2
time-stamps, 53
transaction, 5, 45
  leaf transaction, 45, 47, 56
    body, 57
    guest language, 57
  main transaction, 5, 47
  non-leaf transaction, 47, 50–56
    body, 51
    initialization, 53
    reaction rules, 54
    sub-transaction, 56
    termination, 55
    tuples, 51
  sub-transaction, 5, 47
transaction superposition, 46, 157, 163
transaction union, 46, 157
transputers, 31
tuple mapping, 141, 148
  down, 149
  up, 149
tuple migration, 90
tuple migration algorithm, 93
tuple space, 5, 38, 44
tuple space partition algorithm, 89
tuples, 5, 16, 44
Turing machine, 35

Unity, 20

valid, 111

weak fairness, 52

x-type multiset, 115
x-type tuples, 115

y-type multiset, 115
y-type tuples, 115