

**Computer Science Department Technical Report  
University of California  
Los Angeles, CA 90024-1596**

**DISTRIBUTED COORDINATION OF PROCESS  
INTERACTIONS - FAIRNESS AND FAULT-TOLERANCE**

**Y.-K. Tsay**

**October 1993  
CSD-930034**



UNIVERSITY OF CALIFORNIA  
Los Angeles

**Distributed Coordination of  
Process Interactions  
— Fairness and Fault-Tolerance**

A dissertation submitted in partial satisfaction  
of the requirements for the degree  
Doctor of Philosophy in Computer Science

by

**Yih-Kuen Tsay**

1993

(This is a single-spaced version of the original dissertation submitted to UCLA.)

© Copyright by  
Yih-Kuen Tsay  
1993

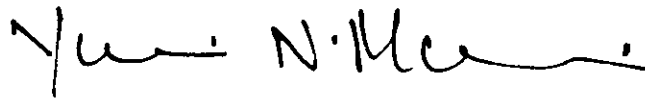
The dissertation of Yih-Kuen Tsay is approved.



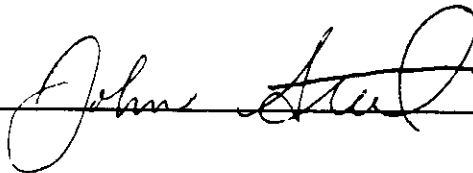
Eli Gafni



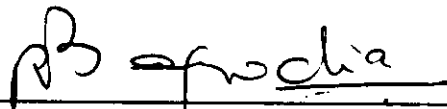
Sheila Greibach



Yiannis N. Moschovakis



John Steel



Rajive L. Bagrodia, Committee Chair

University of California, Los Angeles

1993

*To my mother and the memory of my father*

# TABLE OF CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation	1
1.2	The Process Interaction Problem	3
1.2.1	Computational Model	3
1.2.2	The Problem	3
1.2.3	Fairness	6
1.2.4	Fault-Tolerance	7
1.2.5	Performance Measures	8
1.3	Related Work	9
1.3.1	Fairness in Distributed Programming	9
1.3.2	Coordination Algorithms	9
1.3.3	Formal Specification and Verification	10
1.4	Organization of the Dissertation	11
<b>2</b>	<b>Impossibility Results</b>	<b>13</b>
2.1	Informal Description	13
2.2	Formal Model	15
2.2.1	Program and Computation	15
2.2.2	Temporal Logic	16
2.2.3	Modeling Distributed Systems	18
2.3	Problem Specification	19
2.3.1	Specification of <i>USER</i> (the Given)	20
2.3.2	Specification of $\mathcal{P}$ (the Composite)	21
2.3.3	Constraints on <i>OS</i> (the Solution)	21
2.3.4	Additional Properties: Fairness	22
2.4	Main Results	22
2.4.1	Some Characteristics of Distributed Systems	22
2.4.2	Impossibility Proofs	23
2.5	Discussion	29

2.5.1	Generalization of Main Results . . . . .	29
2.5.2	Basic Assumptions Revisited . . . . .	31
<b>3</b>	<b>Fair Algorithms . . . . .</b>	<b>33</b>
3.1	Algorithm A . . . . .	33
3.1.1	Informal Description . . . . .	33
3.1.2	The Algorithm . . . . .	34
3.1.3	Correctness Proof . . . . .	38
3.1.4	Performance Analysis . . . . .	41
3.2	Algorithm B . . . . .	46
3.2.1	Process Interactions and Dining Philosophers . . . . .	46
3.2.2	Alternative Transformation . . . . .	48
3.2.3	Further Improvement . . . . .	52
3.3	Generalized Algorithms . . . . .	53
<b>4</b>	<b>Fault-Tolerance . . . . .</b>	<b>54</b>
4.1	Detectable Failures . . . . .	54
4.1.1	Single Failures . . . . .	55
4.1.2	Multiple Failures . . . . .	59
4.2	Undetectable Failures . . . . .	60
4.2.1	Algorithms with Small Failure Locality . . . . .	60
4.2.2	Minimizing Failure Locality . . . . .	61
4.3	Remarks . . . . .	70
<b>5</b>	<b>Fairness and UNITY . . . . .</b>	<b>71</b>
5.1	An Introduction to UNITY . . . . .	71
5.1.1	Program and Execution Model . . . . .	71
5.1.2	Programming Logic . . . . .	72
5.1.3	Theorems . . . . .	73
5.1.4	Program Composition . . . . .	75
5.1.5	Conditional Properties . . . . .	75
5.2	Expressiveness of UNITY . . . . .	76
5.2.1	Unconditional Properties . . . . .	77



5.2.2	Conditional Properties . . . . .	78
5.3	Deducing Fairness Properties in UNITY . . . . .	84
5.3.1	UNITY and Temporal Logic . . . . .	85
5.3.2	Relative Completeness . . . . .	86
5.3.3	Specialized Rules . . . . .	90
<b>6</b>	<b>UNITY Proof of Fair Algorithms . . . . .</b>	<b>92</b>
6.1	Preparation . . . . .	92
6.1.1	Modeling Distributed Systems . . . . .	92
6.1.2	Plausible Inference Rules . . . . .	92
6.2	Specification of the Problem . . . . .	93
6.2.1	Specification of <i>USER</i> (the Given) . . . . .	95
6.2.2	Specification of $\mathcal{P}$ (the Composite) . . . . .	95
6.2.3	Constraints on <i>OS</i> (the Solution) . . . . .	96
6.2.4	Simple Refinement of the Specification . . . . .	96
6.3	Verification of a Solution . . . . .	97
6.3.1	The <i>OS</i> . . . . .	97
6.3.2	Correctness of the <i>OS</i> . . . . .	99
<b>7</b>	<b>Conclusion . . . . .</b>	<b>108</b>
7.1	Contributions . . . . .	108
7.2	Future Research . . . . .	109
	<b>References . . . . .</b>	<b>111</b>

## LIST OF FIGURES

1.1	Possible execution of a system of three interacting processes . . . . .	5
2.1	The interaction graph of a binary interaction problem instance . . . . .	14
2.2	The interaction graph of a multiway interaction problem instance . . . . .	15
2.3	Compositions of <b>users</b> and <b>os</b> 's . . . . .	20
2.4	Major state transitions in a phase of the computation under construction . . . . .	26
2.5	The transition to idle of <b>user<sub>k</sub></b> has no effect on starting interaction $I (= \{i, j\})$ . . . . .	27
2.6	“Minimal” interaction graphs ( $j$ and $j'$ may be identical) . . . . .	30
3.1	Basic idea of Algorithm A . . . . .	35
3.2	The order of sending tokens by a process in Algorithm A . . . . .	36
3.3	Experimental performance . . . . .	46
3.4	The interaction graph and the conflict graph of a problem instance . . . . .	47
4.1	How an extra copy of token is removed . . . . .	57
5.1	The operational implication of “ $p$ unless $q$ ” . . . . .	78
5.2	The operational implication of “ $p \mapsto q$ ” . . . . .	78
6.1	Compositions of <b>users</b> and <b>os</b> 's . . . . .	94

## LIST OF TABLES

4.1	Comparison of algorithms and the lower bounds for the dining philosophers problem . . . . .	62
4.2	Comparison of algorithms for binary interactions . . . . .	70

## ACKNOWLEDGMENTS

I am profoundly indebted to my advisor, Prof. Rajive L. Bagrodia, who introduced me to the interesting and important problem of interprocess synchronization. This work would not have been possible without his guidance and constant encouragement. Lastly, I also want to thank him for providing me with financial support (through NSF PYI Award number ASC 9157610 and an ONR grant number N00014-91-J-1605).

During my time at UCLA, I have been influenced by many professors and fellow graduate students; my research has greatly benefited from interactions with them. I would like to thank Prof. Eli Gafni, who made me aware of some important results in distributed computing that are related to this work. I am also grateful to Prof. Sheila Greibach who organized the CS288S seminar series every quarter where I had many chances to present my work and receive constructive criticism. I was enlightened by the numerous presentations of the participants of the seminar including Prof. Greibach, Prof. Gafni, Elizabeth Borowsky, Ching-Tsun Chou, Joseph Pemberton, and Weixiong Zhang and many other occasional participants.

I am grateful to Prof. Yiannis N. Moschovakis and Prof. John Steel of the Department of Mathematics for serving in my dissertation committee. They were also responsible for introducing me to the world of mathematical logic through lectures and conversations. Prof. Moschovakis, in particular, has answered many of my questions regarding the application of logic to concurrency theory.

I had chances to converse, in person and/or through mail, with Prof. Jay Misra of UT Austin and Prof. K. Mani Chandy of Caltech. Their works on UNITY and distributed algorithms have been a great inspiration to me and I thank them both.

My office-mates Maneesh Dhagat, Vikas Jha, Wen-Toh Liao, and Chien-Chung Shen all deserve special thanks; our office has been both a serious and a cheerful research environment. Thanks are also due to Valerie Aylett, who often provided me with much needed secretarial assistance.

I would like to take this rare opportunity to thank my parents. Being blue-collar workers, life has not been easy for them. Without their understanding and support, I could not have been able to come to the States to pursue my Ph.D. in the first place. My father, to my greatest sorrow, passed away near the end of my first year at UCLA; my mother had the courage to keep his death secret to me for a month to not disturb my study. To them I owe an insurmountable debt.

Last but certainly not least, I want to express my deepest gratitude to my wife Ing-hwa. Besides being a graduate student herself, she has spent much time in taking care of the housekeeping chores. With her company and encouragement, the past two years have been most enjoyable and productive, which enabled me to finish this dissertation in a shorter time.

## VITA

- 1962 Born, Taipei, TAIWAN.
- 1982–1983 Research Assistant, Department of Computer Science and Information Engineering, National Taiwan University. Participated in the design and implementation of a portable Pascal compiler.
- 1984 B.S. in Computer Science and Information Engineering, National Taiwan University.
- 1984–1986 Second Lieutenant, The Army on Taiwan.
- 1986–1987 Research Assistant, Department of Mathematics, National Taiwan University. Implemented an interactive teaching software for college Calculus.
- 1989 M.S. in Computer Science, University of California, Los Angeles.
- 1989–present Research Assistant, Computer Science Department, University of California, Los Angeles.

## PUBLICATIONS

1. **“Some Impossibility Results in Interprocess Synchronization,”** Y.-K. Tsay and Rajive L. Bagrodia. *Distributed Computing*, 6(4):221–231, 1993. (Also issued as Technical Report CSD-920023, UCLA, 1992.)
2. **“A Real-Time Algorithm for Fair Interprocess Synchronization,”** Y.-K. Tsay and Rajive L. Bagrodia. In *Proceedings of 12th International Conference on Distributed Computing Systems*, pages 716–723, June 1992.
3. **“Fault-Tolerant Algorithms for Fair Interprocess Synchronization,”** Y.-K. Tsay and Rajive L. Bagrodia. To appear in *IEEE Transactions on Parallel and Distributed Systems*.
4. **“Operational Implication of Conditional UNITY Properties,”** Y.-K. Tsay and Rajive L. Bagrodia. Submitted for publication.

5. **“Deducing Fairness Properties for UNITY Programs: A New Completeness Result,”** Y.-K. Tsay and Rajive L. Bagrodia. Submitted for publication.
6. **“An Algorithm with Optimal Failure Locality for the Dining Philosophers Problem,”** Y.-K. Tsay and Rajive L. Bagrodia. Submitted for publication.

ABSTRACT OF THE DISSERTATION

**Distributed Coordination of  
Process Interactions  
— Fairness and Fault-Tolerance**

by

**Yih-Kuen Tsay**

Doctor of Philosophy in Computer Science  
University of California, Los Angeles, 1993  
Professor Rajive L. Bagrodia, Chair

The *process interaction problem* abstracts the basic issues in implementing the symmetric, nondeterministic, and synchronous communication constructs of programming languages like CSP and IP on a distributed architecture. A solution to the problem is required to ensure that (a) a communication, or an interaction, is initiated only when all participants are ready, (b) interactions with common participants do not proceed simultaneously, and (c) if all participants of an interaction are ready, then at least one of them will eventually participate in some interaction. A special case of the problem, where each interaction involves exactly two processes, is called the *binary interaction problem*; the general case is also referred to as the *multiway interaction problem*.

We strengthen the problem requirements to include two fairness properties: *strong process fairness* and *strong interaction fairness*. We prove that, in general, strong process fairness is impossible for multiway interactions and strong interaction fairness is impossible for binary interactions and hence for multiway interactions. We describe an efficient algorithm for binary interactions that satisfies strong process fairness. The algorithm has the best known message cost and response time.

We then introduce *process failures* and modify the problem requirements accordingly. Two failure models are considered: the *detectable* and the *undetectable fail-stop* models. We show how the fair algorithm for binary interactions can be extended to tolerate detectable failures. Based on an existing algorithm for the dining philosophers problem, we derive another fair algorithm in the undetectable model that has a constant failure locality. The algorithm is further improved to have a response time asymptotically as good as that of the first algorithm.



Finally, we use the process interaction problem as an example to explore the use of UNITY logic in the *formal specification and verification* of reactive systems that need to satisfy certain strong fairness properties. We prove that UNITY logic is sound and relatively complete for proving strong fairness properties. The result is applied to specifying the process interaction problem with strong fairness and to verifying the correctness of a solution.



# CHAPTER 1

## Introduction

### 1.1 Motivation

“... input and output are basic primitives of programming and ... parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra’s guarded command, these concepts are surprisingly versatile.”

— C.A.R. Hoare,  
CACM, August 1978

This seminal proposal of CSP (Communicating Sequential Processes) [Hoa78] spawned two threads of research:

- One studies the use of the “handshaking”, or synchronous, communication construct. The construct has been adopted as a joint-action primitive in various formal models for concurrent systems, more notably in process algebra, e.g. CCS [Mil89] and CSP [Hoa85]. It has also been explored as a convenient language feature in distributed programming, e.g. Ada [Dod82]. The generalization of pairwise communications to multiway communications in both aforementioned areas is also widely pursued [Fra86b, Cha87, Bac88, Fra90].
- The other concerns the implementation of the communication construct, pairwise or multiway, especially on asynchronous distributed architectures [Sch82, Buc83, Rei84a, Sis84, Ram87a, Ram87b, Bag89a, Bag89b, Jou91, Cho92, Par92b]. The non-triviality of implementation comes from the language design decision that allows synchronous communication commands (inputs or outputs) to appear in the guards of the alternative entries of a guarded command.

We abstract the implementation problem as the *process interaction problem*; an alternative formulation called the *committee coordination problem* can

be found in [Cha88]. The problem may roughly be stated as follows: A process may from time to time become ready for several possible communications with other processes. The goal is to design a scheduler that allows the processes to participate in communications such that (a) a communication is initiated only when all participants are ready, (b) communications with common participants do not proceed simultaneously, and (c) if all participants of a communication are ready, then at least one of them will eventually participate in some communication.

We strengthen the problem requirements to include two additional constraints, namely *fairness* and *fault-tolerance*. The importance of fairness in concurrent programming is most eloquently spelled out by Francez's book [Fra86a], which is devoted solely to the subject of fairness. Many other researchers have also studied the subject in various specific contexts, e.g. [Rei84b, Par92a]. In [Fra86a] Francez gives an extensive overview of fairness notions and demonstrates the effects of some of them on program correctness. For instance, some CSP programs will terminate only under the assumption that the semantics of the language stipulates *strong process fairness*. Hence, it is desirable that the implementation enforces certain fairness properties. With respect to fault-tolerance, we consider process failures in two models that have received much attention in the distributed computing literature, viz. the *detectable and restartable fail-stop* model [Sch82, Sch83] and the *undetectable fail-stop* model [Fis85]. The problem requirements are modified accordingly in the presence of process failures.

The goal of this dissertation is to do an in-depth study of the process interaction problem with fairness and fault-tolerance:

First of all, since additional properties have been taken into consideration, the immediate question to ask is whether solutions to the problem exist such that various additional properties are also satisfied. Both positive and negative results should be interesting. In the event that certain additional properties can be satisfied, we seek efficient solutions to make the concerned programming language construct more feasible. The existence of an interesting relationship between the process interaction problem and the *dining philosophers problem* [Dij78, Cha84]<sup>1</sup> has long been recognized. This relationship is further explored in the *dissertation*.

Additionally, we consider the process interaction problem a good example for the study of formal specification and verification of reactive systems<sup>2</sup> that need

---

<sup>1</sup>A description of the dining philosophers problem can be found in Section 3.2.

<sup>2</sup>A reactive system is one that maintains an ongoing interaction with its environment. An operating system is an example of such systems, whose environment is the collection of user programs.

to satisfy certain strong fairness properties. More specifically, we want to explore the use of UNITY logic [Cha88] in this subject. Whereas the subject has been studied extensively in other formalisms including temporal logic [Man92, Lam91], its has not received much attention in the context of UNITY.

## 1.2 The Process Interaction Problem

This section gives a precise definition of the process interaction problem, including the computational model and performance measures. The additional properties of fairness and fault-tolerance are also defined.

### 1.2.1 Computational Model

A distributed system consists of a set of processes that communicate with one another via message-passing. A process is a set of local variables and a set of state transition rules (or actions). The values assumed by the variables of a process constitute the local state of the process; the local states of all processes and the messages in transit (messages that are sent but not yet received) constitute the state of the system. Each state transition rule of a process is in the form of a guarded command where the guard, or enabling condition, depends on its local state and an incoming message. A rule is enabled at a state if its enabling condition is satisfied at that state. An execution (or computation) of the system starts from an initial state where each variable assumes a well-defined initial value and no message is in transit; in each step, either an enabled rule of some process is executed or some message is delivered to its destination. A continuously enabled rule will eventually be executed (i.e. the system is weakly fair); process failures are defined in a later section. Messages are assumed to be delivered in FIFO order unless otherwise stated.

The system is completely asynchronous; no assumptions are made about the relative speeds of the processes or the time bound on message delivery. The notion of time becomes relevant only in measuring the response time of an algorithm.

### 1.2.2 The Problem

The process interaction problem abstracts the basic issues in implementing the symmetric, nondeterministic, and synchronous communication constructs of programming languages like CSP [Hoa78], Script [Fra86b], Joint Action [Bac88], and IP [Fra90] on a distributed architecture.

In the problem, a number of *interactions* are defined on the set of processes in a distributed system; each interaction is a non-empty subset of the process set that represents some synchronization activity among its members. Two interactions are said to *conflict* if they are distinct and have some common member. A process can be in either *active* or *idle* state. An active process may autonomously become idle; an idle process remains idle until it participates in some interaction of which it is a member. (Note that, in general, it is not possible to determine when, or if, an active process will become idle.) An interaction is *enabled* if all of its members are idle; it is *disabled* otherwise.

The goal is to augment each process with a scheduler to select interactions for execution such that the following safety and liveness properties are satisfied:

- **Synchronization:** Only enabled interactions can be started.
- **Mutual Exclusion:** A process can participate in at most one interaction at a time, or equivalently, an interaction cannot be started if some conflicting interaction has been started but not yet terminated.
- **Progress:** When an interaction is enabled, either the interaction or some conflicting interaction will eventually be started.

In the special case where each interaction has exactly two members, the problem is called the *binary interaction problem*; the general case is also referred to as the *multiway interaction problem*<sup>3</sup>. Figure 1.1 shows how a system of three interacting processes may evolve in accordance with the problem specification.

The individual schedulers, one for each interacting process (also referred to as the host process), collectively form a distributed scheduler<sup>4</sup>. We stipulate that each scheduler and its corresponding host process share two variables: *state* and *flag*. The first variable indicates the state (*active* or *idle*) of the host process. This variable can be inspected, but not modified, by the scheduler. The variable *flag* is updated by a scheduler to indicate the specific interaction in which its host process will participate; when the interaction is terminated, the variable is reset by the host process<sup>5</sup>. We assume that each interaction that has been started will

---

<sup>3</sup>The **problem** could be generalized such that an idle process may be willing to participate in one of a **subset** of the interactions of which it is a member. However, a solution to the original problem can easily be adapted to solve the generalized version; an approach will be demonstrated in Section 3.3.

<sup>4</sup>Each individual scheduler is a set of extra variables and a set of extra state transition rules that are added onto the host process.

<sup>5</sup>As starting an interaction should be mutually agreed upon by all the members of the interaction, only one scheduler of the members needs to set its *flag* to indicate that the interaction is started. This is formalized in Chapters 2 and 5.

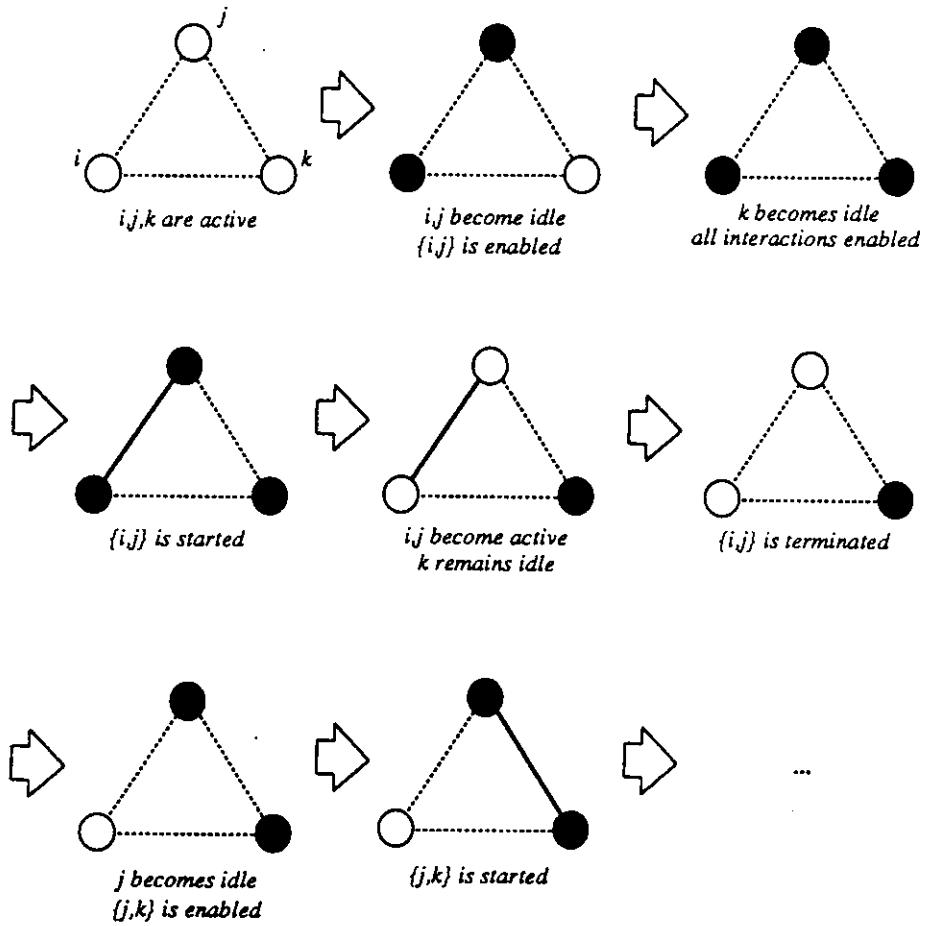


Figure 1.1: Possible execution of a system of three interacting processes

eventually be terminated. Prior to terminating an interaction, each member of the interaction may make a transition from idle to active.

Hereafter a scheduler and its host process will be regarded as a single unit and referred to as a process, except when we need to be more specific.

## Notation

For a given instance of the binary interaction problem, we can view the set of processes and the set of interactions as an undirected graph, called its *interaction graph*. A node in the interaction graph represents a process and an edge represents an interaction between the two processes represented by the end nodes of the edge. In the general case of multiway interactions, the interaction graph of a given problem instance will be a hypergraph, where each edge (interaction) is a set of nodes (processes). Alternatively, one may view the sets of processes and interactions as a *conflict graph*, where a node corresponds to an interaction and an edge indicates that the incident nodes, or interactions, are conflicting with one another.

We will use  $\{i,j,k\}$  to refer to an interaction among process  $i$ , process  $j$ , process  $k$  (hereafter  $p_i$ ,  $p_j$ , and  $p_k$ ). Throughout the dissertation,  $D$  denotes the maximum degree of a given interaction graph; thus  $D$  also represents the maximum number of interactions of which some process is a common member. Finally, the total number of processes in the system is usually denoted by  $n$ .

### 1.2.3 Fairness

Besides the basic safety and liveness properties as required in the problem definition, we are concerned with a stronger class of properties — strong fairness properties. We only consider *strong* fairness properties, as *weak* fairness properties are much easier to implement, perhaps even easier than the basic liveness property. Fairness properties usually appear as assumptions on the execution model of a concurrent program: For instance, the model may require that if a state transition of the program is enabled infinitely often then it is taken infinitely often. Alternatively, the model may assume a weaker notion of fairness, but it may be desirable for a specific program in the model to satisfy a strong fairness property. An algorithm that satisfies a strong fairness property may be described in a model such as UNITY that assumes a weaker notion of fairness.

Although many types of fairness properties have been defined in the literature [Fra86a, Apt88], we restrict our attention to two common forms of strong fairness that are relevant for this problem: *strong process fairness* and *strong*



*interaction fairness.*

- **Strong Process Fairness (SPF):** If an idle process is ready to participate in some enabled interaction infinitely often, then it participates in some, though not necessarily the same, interaction infinitely often.
- **Strong Interaction Fairness (SIF):** If an interaction is enabled infinitely often, then it is started infinitely often.

SPF ensures that a process does not “*starve*”. It subsumes the progress property: Consider the case of binary interactions and assume that the progress property is not satisfied. This implies that although some interaction, say  $\{i,j\}$ , is enabled, neither  $p_i$  nor  $p_j$  will ever participate in any interaction. From the problem definition,  $p_i$  and  $p_j$  must forever remain idle. As a result,  $p_i$  (and  $p_j$ ) will be ready to participate in some interaction, e.g.  $\{i,j\}$ , infinitely often but will never do so, violating SPF.

SIF in turn is strictly stronger than SPF. As an example, suppose that the scenario in Figure 1.1 is repeated indefinitely. In this particular execution, all three interactions are enabled infinitely often; however, interaction  $\{i,k\}$  is never started, violating SIF. It is easy to see that SPF is satisfied, as no process starves.

#### 1.2.4 Fault-Tolerance

Many types of failures have also been studied in different contexts [Sch82, Sch83, Fis85, Dol87, Cha91]. We consider *detectable* and *undetectable fail-stop* process failures, which are more interesting for the problem.

In the detectable failure model [Sch82, Sch83], the failure (and restart) of a process is detectable by other processes in the system. When a process fails, its local variables are reset to their initial values and a failure message is broadcast to each of its neighbors. A process may restart and, when it does, it executes a designated restart step and then resumes normal operation. In general, the synchronization and fairness requirements of the problem cannot be satisfied in the presence of process failures. For example, a process  $p_i$  that has “committed” to a request from  $p_j$  for interaction may subsequently fail, recover, and stay in active state. Meanwhile,  $p_j$  will start interaction  $\{i,j\}$ , which is no longer enabled, violating the synchronization requirement. Regarding fairness, a process  $p_j$  may fail whenever it is waiting to request or commit to interaction  $\{i,j\}$ , causing  $p_i$  to starve, violating SPF and hence SIF. Thus, the problem requirements must be reformulated in the presence of failures. While no change is necessary for the

mutual exclusion and progress requirements, the modified synchronization and fairness requirements state that, for a system with a *finite number* of failures, the synchronization and fairness requirements as specified in Sections 1.2.2 and 1.2.3 will eventually be satisfied.

In the undetectable failure model [Fis85], when a process fails, it will not take any step in the rest of execution of the system and the failure cannot be detected by any other processes. It has been shown that, in a message-passing asynchronous or similar system, mutual exclusion among the processes in the system cannot be achieved under the undetectable model [Dol87]. The implication of the result is that, in general, the requirements of the process interaction problem cannot be satisfied. More recently, the notion of *failure locality* was introduced [Cho92] to measure the size of the network that is affected by the failure of a process. In the context of the process interaction problem, an algorithm has a failure locality of  $m$  if a process behaves as specified by the problem requirements provided that no processes within  $m$  hops away from the process fail; two processes are  $m$  hops away if the length of the shortest path between the two processes in the interaction graph is  $m$ . The goal in the presence of undetectable failures hence is to design algorithms with small failure localities.

### 1.2.5 Performance Measures

Two metrics have been defined to measure the performance of an algorithm for the process interaction problem: *message cost* and *response time*. The message cost is the worst-case count of number of messages sent by a process or received as a reply, from the time the process becomes idle until it participates in some interaction. Assuming that every message is delivered within one unit of time and the time for doing any computation by a scheduler is negligible, the response time is the elapsed time in the worst case from the time an interaction is enabled until the interaction or some conflicting interaction is started.

An algorithm is considered *efficient* if its response time is independent of the total number of processes  $n$  and is polynomial in  $D$ , i.e. the maximum degree of the interaction graph.

## 1.3 Related Work

### 1.3.1 Fairness in Distributed Programming

In [Apt88], Apt et al. proposed three criteria<sup>6</sup> for determining the appropriateness of fairness notions in distributed languages. They concluded that none of the common forms of fairness (including SPF and SIF) for multiway interactions meets all of the suggested criteria and only SPF for binary interactions does. We shall prove that, in general, SIF is impossible to implement (in a distributed manner as stipulated by our problem definition) for binary and hence for multiway interactions and SPF is impossible for multiway interactions<sup>7</sup>. We shall also present algorithms for binary interactions that satisfy SPF. Our results corroborate their conclusions.

Dijkstra [Dij88] has contended that fairness is a void obligation for language implementors in that it is impossible to detect if the obligation has been fulfilled. Our impossibility results show that under the assumptions of our model, some fairness notions for process interactions are, in fact, impossible to implement.

### 1.3.2 Coordination Algorithms

A large number of algorithms have been devised for binary interactions [Buc83, Sis84, Ram87b, Bag89b] and for multiway interactions [Ram87a, Cha88, Bag89a, Jou91, Cho92, Par92b]; algorithms for multiway interactions, of course, work for binary interactions. In light of the impossibility results on implementing fairness for multiway interactions, we shall review existing algorithms in the context of binary interactions. Although most algorithms satisfy the basic safety and liveness properties, few implement fairness (SPF) or consider process failures.

Existing efficient algorithms (with response time independent of the number of processes), e.g. [Rei84a], are not fair and do not handle process failures. Sistla's algorithm [Sis84] satisfies a stronger fairness property in a slightly different model<sup>8</sup>, but has response time dependent on the number of processes and,

---

<sup>6</sup>The three criteria are: *feasible*, *equivalence robust*, and *liveness enhancing*.

<sup>7</sup>The *fairness* properties that we show impossible to implement distributedly all turn out not to be *equivalence robust*. However, there are fairness properties, e.g. weak process fairness for binary interactions, that are not equivalence robust but can be implemented distributedly.

<sup>8</sup>Sistla [Sis84] considered a fairness property stronger than SPF: if a process is continuously idle and is ready to participate in some enabled interaction infinitely often, then the same interaction is started infinitely often. SPF does not require the same interaction be started infinitely often; instead, it only requires the process participate in some interaction infinitely often. However, if a process always makes a transition from idle to active after some interaction

more significantly, on the *duration of an interaction between two processes*. Finally, the response time of existing solutions that tolerate detectable failures, e.g. [Sch82], is not independent of the number of processes.

In contrast, we shall present an algorithm that is a significant improvement over existing solutions for binary interactions in the following respects: (a) the algorithm satisfies the modified problem requirements as described in Section 1.2.4 in the presence of detectable failures and (b) assuming no failures actually occur, its message cost and response time are independent of the number of processes.

To cope with undetectable failures, Choy and Singh [Cho92] recently proposed a solution to the process interaction problem that has a constant failure locality but *does not satisfy SPF for the case of binary interactions*; the solution is based on their algorithm for the dining philosophers problem with a constant failure locality. As we will show in a later chapter, the lack of fairness in their solution is inherent in the usual transformation that they adopted to transform the process interaction problem to the dining philosophers problem.

We shall propose a transformation of the binary interaction problem to the dining philosophers problem that can guarantee SPF. We use this transformation and the solution to the dining philosophers problem in [Cho92] to derive an algorithm for binary interactions that satisfies fairness while maintaining a constant failure locality. The response time of the derived algorithm is further improved to be asymptotically as good as that of the fair algorithm proposed for detectable failures.

### 1.3.3 Formal Specification and Verification

As planned, we are mainly interested in the use of the UNITY formalism. Nonetheless, in the proof of the impossibility results, we apply a variation of the branching time temporal logic in [Eme89] to make the presentation formal and precise. Our work perhaps is the first that uses branching time temporal logic in proving impossibility results.

UNITY was proposed by Chandy and Misra in [Cha88]; a brief introduction to the formalism is presented in Chapter 5. Their book contains abundant examples, ranging from communication protocols, termination detection to sorting and searching. Many other researchers have also explored the applications of UNITY [Kna90, Cun90, San91b, Liu92a, Liu92b] as well as meta-results about its logic [Ger89, Jut89, San91a, Rao91, Kna92]. In particular, a recent work on the application of UNITY in specifying and verifying fault-tolerant programs can

---

of which it is a member is started, then the two fairness properties become identical.

be found in [Liu92b].

It has been shown that UNITY is sound and relatively complete (in the sense of [Coo78]<sup>9</sup>) in that an unconditional property in terms of the UNITY logic relations: *unless*, *invariant*, or  $\mapsto$ , is provable from a program if and only if a corresponding operational property is satisfied by each execution of the program [Jut89, San91a, Rao91, Kna92]. However, no analogous result for conditional properties has been provided in the literature.

We shall prove new results about UNITY logic. In particular, we show that a strong fairness property “if  $p$  holds infinitely often then  $q$  also holds infinitely often” (or  $\Box\Diamond p \Rightarrow \Box\Diamond q$  in temporal logic notation) can be specified by the conditional property “Hypothesis:  $true \mapsto p$  Conclusion:  $true \mapsto q$ ” in UNITY. We further show that UNITY is relatively complete for proving strong fairness properties. Hence, it is possible to specify the process interaction problem with strong fairness and prove the correctness of a solution entirely within the UNITY formalism.

## 1.4 Organization of the Dissertation

The rest of the dissertation is organized as follows:

Chapter 2 contains the impossibility results; both formal and informal proofs are presented.

Chapter 3 describes algorithms for binary interactions that satisfy SPF. The second algorithm is based on a new transformation from the binary interaction problem to the dining philosophers problem.

Chapter 4 shows how the fair algorithms can be extended to tolerate detectable failures. To cope with undetectable failure, an existing dining philosophers algorithm is adopted by the second algorithm to achieve both fairness and constant failure locality. It also shows how the failure locality can be minimized at the cost of increasing response time.

Chapter 5 gives an introduction to UNITY and shows its expressive power along with its relative completeness in proving strong fairness properties.

Chapter 6 applies UNITY in specifying the process interaction problem with strong fairness and verifying the correctness of a solution.

Chapter 7 concludes the dissertation with a summary of its main contributions

---

<sup>9</sup>Roughly speaking, a proof system for program correctness is relatively complete if the truth of a program property can be reduced to the truth of assertions on program statements.

and a list of possible directions for future research.

Technically, Chapter 2 is independent from its subsequent chapters, while Chapters 3 and 4 form a coherent unit and so do Chapters 5 and 6.

## CHAPTER 2

### Impossibility Results

We prove that, in general, SIF is impossible for binary and hence for multiway interactions and SPF is impossible for multiway interactions. Although the impossibility results are proven in a message-passing model of distributed system, the results hold in any model where (a) each process autonomously decides when and if it is willing to participate in some interaction and (b) the model assumes low atomicity, i.e. in one atomic step a process cannot both change its local state and inform other processes of the change.

Much of the material in this chapter is adapted from our work reported in [Tsa93d].

#### 2.1 Informal Description

We sketch the impossibility results, focusing on the crucial assumptions of the problem and their consequence to the results. All technical terms and assumptions will be made formal and precise in subsequent sections.

In the description of the process interaction problem (Section 1.2.2), we have made three assumptions which are crucial to the impossibility results: (i) An active process may or may not become idle. (ii) If an active process becomes idle, it does so autonomously. (iii) The state transition of a process is not immediately observable by other processes or their schedulers. Modifications to the assumptions and their effect on the impossibility results are examined in Section 2.5.

Consider an instance of the binary interaction problem with three processes  $i$ ,  $j$ , and  $k$  and three interactions  $\{i,j\}$ ,  $\{j,k\}$ , and  $\{k,i\}$ ; its interaction graph is shown in Figure 2.1. The schedulers are referred to as schedulers  $i$ ,  $j$ , and  $k$ , respectively.

In accordance with assumptions (i) and (ii), we further assume that the three processes have the following property: (iv) It is always possible for an active process to remain active or autonomously become idle. We shall construct an execution of the system where interaction  $\{k,i\}$  becomes enabled infinitely often but is never started, violating SIF.

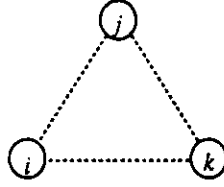


Figure 2.1: The interaction graph of a binary interaction problem instance

The construction is developed around the following key observations: First, if some interaction  $I$  is enabled, the schedulers cannot indefinitely postpone the execution of  $I$  while waiting for other interactions to become enabled; otherwise, if other interactions never become enabled as allowed by assumption (iv), the progress requirement will be violated. This observation will be established formally by Lemma 3 in Section 2.4.2. Secondly, when the schedulers decide to start some interaction  $I$ , it is possible that a conflicting interaction  $K$  becomes enabled before  $I$  is started. However, assumption (iii) implies that in general it is impossible for the schedulers to detect that  $K$  is enabled before  $I$  is started. This will be established formally in Theorem 1 in Section 2.4.2.

In some execution of the system, the following scenario may occur repeatedly in violation of SIF: Initially, all processes are active and no interaction is started. Processes  $i$  and  $j$  go from active to idle, while process  $k$  remains active. To satisfy the progress requirement, schedulers  $i$  and  $j$  decide to start interaction  $\{i,j\}$ . Meanwhile, process  $k$  becomes idle right before interaction  $\{i,j\}$  is actually started thus causing interactions  $\{j,k\}$  and  $\{k,i\}$  to also become enabled together with interaction  $\{i,j\}$ . After participating in interaction  $\{i,j\}$ , process  $j$  becomes idle again causing  $\{j,k\}$  to become enabled, while process  $i$  remains active. Schedulers  $j$  and  $k$  then start interaction  $\{j,k\}$ . Subsequently, processes  $j$  and  $k$  again become active and interaction  $\{j,k\}$  is terminated. Now, all processes are active and no interaction is started; and the above scenario is repeated.

To see that SPF is impossible for multiway interactions, we add process  $l$  to the previous instance and change interaction  $\{k,i\}$  into  $\{k,i,l\}$ . Figure 2.2 shows the interaction graph of the new problem instance of multiway interaction.

In an execution of the new system, assume process  $l$  becomes idle. Processes  $i$ ,  $j$ , and  $k$  behave exactly the same as above. It follows that process  $l$  is ready to participate in interaction  $\{k,i,l\}$  infinitely often but never does so.

In the remainder of this chapter, we define a formal model, construct a formal specification of the process interaction problem with strong fairness, and derive the impossibility results.



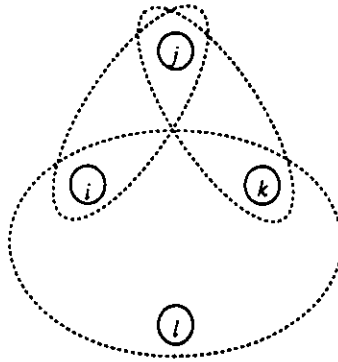


Figure 2.2: The interaction graph of a multiway interaction problem instance

## 2.2 Formal Model

The formal model is essentially the same as the one described in Section 1.2.1 but in greater detail. The model resembles very much the program model in UNITY. We also define a branching time temporal logic to express and reason about program properties.

### 2.2.1 Program and Computation

A *program* consists of a set of variables and a set of (state transition) rules. Each variable may assume values in some domain, a subset of which is specified as possible initial values of the variable. Every program includes an auxiliary variable called *label*, which may assume the name of a rule or an initial value *null*. The *state* of a program is the tuple of values assumed by the program variables, including *label*; an *initial* state is a state satisfying the specification of initial values of the program variables. Each *rule* is specified by a unique non-*null* tag, called its name, a predicate on program states, called its guard, and a sequence of assignment statements, called its body. A rule is *enabled* at a program state if the state satisfies its guard; otherwise it is *disabled*.

A *computation* (or *execution*) of a program starts from any initial state and goes on **forever**. In each step of the computation, a rule is selected nondeterministically for execution and the value of *label* is updated with the name of the selected rule. If the selected rule is enabled, its body is executed; nothing else happens otherwise. The execution of a rule (enabled or disabled) results in a deterministic state transition of the program. Thus, each computation uniquely determines an infinite sequence of program states. To exclude computations where a continuously enabled rule is indefinitely ignored, we postulate a fair selection criterion:

each rule of the program is selected infinitely many times (regardless of whether or not the rule is enabled) in a computation<sup>1</sup>.

We introduce some notations:

$s$  (or  $s'$ ,  $s_0, s_1, \dots$  etc.) denotes a program state.

$x_i$  denotes the  $i$ -th element of sequence  $x$ . We assume the elements of a sequence are numbered from 0.

$x^i$  denotes the suffix of sequence  $x$  starting from the  $i$ -th element, i.e.  $x_i x_{i+1} x_{i+2} \dots$ .

$\langle s, x \rangle$  denotes the sequence of states determined by the execution of sequence of rules  $x$  starting from state  $s$ .

$(s, x)$  denotes the last state in  $\langle s, x \rangle$ , assuming  $x$  is finite.

$xy$  denotes the concatenation of sequences  $x$  and  $y$ , assuming  $x$  is finite. From the definition of the computational model, it follows that  $(s, xy) = ((s, x), y)$ .

$x \leq y$  denotes that sequence  $x$  is a prefix of sequence  $y$ .

$Init_{\mathcal{D}}$  denotes the predicate that specifies the initial states of a program  $\mathcal{D}$ .

$Rule(\mathcal{D})$  denotes the set of rules of  $\mathcal{D}$ .

$Rule^*(\mathcal{D})$  denotes the set of all infinite sequences of rules of  $\mathcal{D}$  such that each rule is selected infinitely many times.

$Pref(\mathcal{D}) \equiv \{x \mid \exists \alpha : \alpha \in Rule^*(\mathcal{D}) \wedge x \leq \alpha\}$  is the set of all prefixes of sequences in  $Rule^*(\mathcal{D})$ .

$Comp(\mathcal{D}) \equiv \{\sigma \mid \exists s, \alpha : (Init_{\mathcal{D}} \text{ at } s) \wedge \alpha \in Rule^*(\mathcal{D}) \wedge \sigma = \langle s, \alpha \rangle\}$  is the set of all possible computations of  $\mathcal{D}$ .

$Comp^*(\mathcal{D}) \equiv \{\sigma \mid \exists x : x\sigma \in Comp(\mathcal{D})\}$  is the suffix closure of  $Comp(\mathcal{D})$ .

$Branch(s) \equiv \{\sigma \mid \sigma \in Comp^*(\mathcal{D}) \wedge \sigma_0 = s\}$  is the set of all possible “futures” of the state  $s$ . Note that each possible future of a program state is a sequence of states.

### 2.2.2 Temporal Logic

Our logic is a variation of the branching time temporal logic in [Eme89]. We do not distinguish between state formulae and path formulae, but simply refer to them as temporal formulae. We use  $\circ, \square, \diamond$  instead of the more standard

---

<sup>1</sup>The fair selection criterion only requires that each rule be selected infinitely often. In particular, it is possible that a rule is enabled infinitely often (not continuously) but the body of the rule is never executed, because it may be selected only when it is disabled.

*X, G, F*. Quantifiers are introduced to abbreviate the conjunction or disjunction of a number of temporal formulae with similar pattern.

We directly define the semantics of our logical language with respect to a program  $\mathcal{D}$ ; its syntax is implicitly defined by these semantic definitions. Suppose  $a, b, c$  are predicates on program states and  $p, q$  are temporal formulae.  $\sigma$  is an infinite sequence in  $Comp^*(\mathcal{D})$ ; recall that  $\sigma^i$  denotes its suffix  $\sigma_i\sigma_{i+1}\sigma_{i+2}\dots$ . In the following definitions as well as subsequent sections, the logical connectives  $\neg$ ,  $\wedge$ ,  $\vee$ , and  $\Rightarrow$  and quantifiers  $\forall$  and  $\exists$ , when not occurring as part of a temporal formula, should be interpreted according to their standard meanings in classical logic.

$$a \mid \sigma \equiv a \text{ at } \sigma_0 \text{ (} a \text{ is true at state } \sigma_0 \text{)} \quad (\text{A1})$$

$$\neg p \mid \sigma \equiv \neg(p \mid \sigma) \quad (\text{A2})$$

$$\bigcirc p \mid \sigma \equiv p \mid \sigma^1 \quad (\text{A3})$$

$$\Box p \mid \sigma \equiv \forall i : i \geq 0 \Rightarrow (p \mid \sigma^i) \quad (\text{A4.1})$$

$$\Diamond p \mid \sigma \equiv \exists i : i \geq 0 \wedge (p \mid \sigma^i) \quad (= \neg \Box \neg p \mid \sigma) \quad (\text{A4.2})$$

$$A p \mid \sigma \equiv \forall \tau : \tau \in Branch(\sigma_0) \Rightarrow (p \mid \tau) \quad (\text{A5.1})$$

$$E p \mid \sigma \equiv \exists \tau : \tau \in Branch(\sigma_0) \wedge (p \mid \tau) \quad (= \neg A \neg p \mid \sigma) \quad (\text{A5.2})$$

$$p \vee q \mid \sigma \equiv (p \mid \sigma) \vee (q \mid \sigma) \quad (\text{A6.1})$$

$$p \wedge q \mid \sigma \equiv (p \mid \sigma) \wedge (q \mid \sigma) \quad (= \neg(\neg p \vee \neg q) \mid \sigma) \quad (\text{A6.2})$$

$$p \Rightarrow q \mid \sigma \equiv (\neg p \vee q) \mid \sigma \quad (\text{A6.3})$$

$$p \Leftrightarrow q \mid \sigma \equiv ((p \Rightarrow q) \wedge (q \Rightarrow p)) \mid \sigma \quad (\text{A6.4})$$

$$a \text{ Until } b \mid \sigma \equiv \exists i : i \geq 0 \wedge (b \mid \sigma^i) \wedge (\forall j : 0 \leq j < i \Rightarrow (a \mid \sigma^j)) \quad (\text{A7.1})$$

$$a \text{ Unless } b \mid \sigma \equiv (a \Rightarrow (\Box a \vee (a \text{ Until } b))) \mid \sigma \quad (\text{A7.2})$$

(Notice that “ $a \text{ Unless } b \mid \sigma$ ” is true if  $a$  is false at  $\sigma_0$ , regardless of the truth value of  $b$ . This definition, motivated by the “*unless*” in [Cha88], is very useful in specifying safety properties of a program.)

A quantified temporal formula is interpreted as multiple occurrences of the temporal formula with the quantified variables replaced by their possible values. “ $(\forall x : Q(x) \Rightarrow p(x)) \mid \sigma$ ” is evaluated to true if all occurrences of  $p(x)$  with  $x$  satisfying  $Q(x)$  are evaluated to true. *An important constraint on the predicate  $Q(x)$  is that its truth value does not depend on program states.* For example,  $x$  can be the index of processes and  $Q(x)$  asserts that  $x$  range over some set of numbers. Similarly, “ $(\exists x : Q(x) \wedge p(x)) \mid \sigma$ ” is evaluated to true if at least one occurrence of  $p(x)$  with  $x$  satisfying  $Q(x)$  is evaluated to true. For brevity, temporal formulae will often be written without explicit quantification; they are

assumed to be universally quantified over all values of the free variables.

The properties of a program  $\mathcal{D}$  are expressed by statements of the form “ $p$  in  $\mathcal{D}$ ,” where  $p$  is a temporal formula.

$$p \text{ in } \mathcal{D} \equiv \forall \sigma : \sigma \in \text{Comp}^*(\mathcal{D}) \Rightarrow (p \mid \sigma) \quad (= \forall \tau : \tau \in \text{Comp}(\mathcal{D}) \Rightarrow (\Box p \mid \tau)) \quad (\text{P1})$$

The following are some temporal formulae that are true for any sequence of  $\text{Comp}^*(\mathcal{D})$ . Their validity can easily be verified from the definitions (A1)–(A7.2). Notice again that  $a$ ,  $b$ ,  $c$ , and  $d$  are predicates on program states and do not involve temporal operators.

$$\Box(p \wedge q) \Leftrightarrow (\Box p \wedge \Box q) \quad (\text{T1})$$

$$(a \wedge \mathbf{A}p) \Leftrightarrow \mathbf{A}(a \wedge p) \quad (\text{T2})$$

$$(\Box(p \Rightarrow q) \wedge \Box p) \Rightarrow \Box q \quad (\text{T3})$$

$$(a \text{ Unless } b \vee c) \Rightarrow ((a \text{ Unless } b) \vee (a \text{ Unless } c)) \quad (\text{T4})$$

$$((a \text{ Unless } b) \wedge (c \text{ Unless } d)) \Rightarrow (a \wedge c \text{ Unless } b \vee d) \quad (\text{T5})$$

### 2.2.3 Modeling Distributed Systems

Programs can be composed to produce composite programs in a natural way: The set of variables (rules) of the composite program is the union of the sets of variables (rules) of all component programs. Each component program of a composite program will be referred to as a *module*. Variables belonging to more than one module are termed *shared* variables. A constraint on program composition requires that each shared variable be initialized “consistently” by all sharing modules. A program composed of modules  $F$  and  $G$  is denoted by  $F \parallel G$ <sup>2</sup>. Note that  $F$  and  $G$  may themselves be composite programs. In a computation of  $F \parallel G$ , each rule of  $F$  or  $G$  must be selected infinitely often. A computation of  $F$  is no longer a computation of  $F \parallel G$ , since the rules of  $G$  are not selected; analogously for  $G$ . For clarity, the state of a module in a composite program will be referred to as the *local* state of the module.

We consider programs where modules are functionally divided into two categories: *processes* which do significant computations and *channels* which simply relay messages. Distinct processes have disjoint sets of variables and so do distinct channels; variables may be shared only between a process and a channel. A sender process may send a message to a receiver process by depositing the message in a message queue shared by the sender and a channel; the channel

---

<sup>2</sup>The notation “ $\parallel$ ” is borrowed from UNITY; another commonly used notation is “ $\|$ ”.

then delivers the message by removing the message and depositing it in another message queue shared by the channel and the receiver process. (Note that the notions of process and channel are relative to a program. A module in a process, which shares variables with other modules in the same process, is *not* a process of the entire program; analogously for modules in a channel.)

Programs composed in the above manner are called *distributed* programs. A distributed program models a distributed system with message-passing.

## 2.3 Problem Specification

Let *USER* refer to a distributed program which contains a set of processes and the channels that relay messages among the processes and *OS* refer to the distributed scheduler that implements synchronizations among the asynchronous processes in *USER*. The composite program *USER*||*OS* is referred to as  $\mathcal{P}$ .

A process in *USER* with index  $i$  is denoted by  $\text{user}_i$ ; analogously for *OS*. Let  $p_i$  denote  $\text{user}_i$ || $\text{os}_i$ ; each  $p_i$  is a process in  $\mathcal{P}$ . We shall refer to a process in *USER* as a *user*, a process in *OS* as an *os*, and a process in  $\mathcal{P}$  as a *process*. An interaction among  $\text{user}_i$ ,  $\text{user}_j$ , and  $\text{user}_k$  is represented by  $\{i, j, k\}$ .  $\mathcal{I}$  is the set of all interactions defined among users; each element of  $\mathcal{I}$  is a nonempty subset of the process indexes. Two interactions are said to be *conflicting* if they are distinct and have a non-empty intersection.

Each *user* and the corresponding *os* share two variables: *state* and *flag*. *state* may assume the value *active* or *idle*. The two states of a *user* (or loosely, a *process*) correspond to a *user* that does not want to participate in any interaction and a *user* that is waiting to participate in some interaction. A *user* is said to be *participating* in an interaction if it is a member of the interaction and the interaction is started. Interaction  $I$  is started if one of its members, say  $p_i$ , sets  $\text{flag}_i$  to  $I$  and is terminated if  $\text{flag}$  is set back to *null* for all members of  $I$ . The relationship between a *user* and its *os* and that between two processes (each formed by the composition of a *user* and its *os*) are depicted in Figure 2.3. As a pair of *processes* in  $\mathcal{P}$  communicate via channels and do not share variables, an update of  $\text{state}_i$  in any computation step is not observed by any  $p_j$  ( $j \neq i$ ) in the next step, enforcing assumption (iii) in Section 2.1.

We introduce some abbreviations for commonly used predicates:

$$\text{active}_i \equiv (\text{state}_i = \text{active}), \text{ analogously for } \text{idle}_i \quad (\text{d1})$$

$$\text{enable}^I \equiv (\forall i : i \in I \Rightarrow \text{idle}_i) \quad (\text{d2})$$

$$\text{start}^I \equiv (\exists i : i \in I \wedge \text{flag}_i = I) \quad (\text{d3})$$

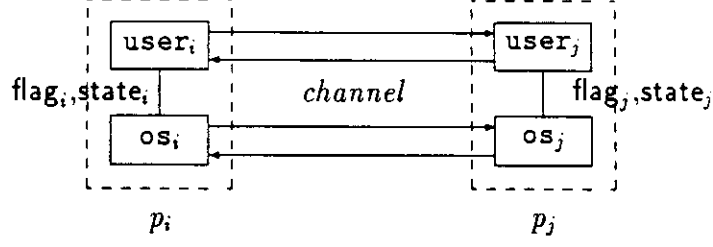


Figure 2.3: Compositions of users and os's

$$engaged_i \equiv (\exists I : i \in I \wedge start^I) \quad (d4)$$

$$E[I, J] \equiv (I \neq J \wedge I \cap J \neq \phi), \text{ interactions } I \text{ and } J \text{ are conflicting} \quad (d5)$$

We use the temporal logic language introduced in Section 2.2.2 to specify the properties of *USER* and  $\mathcal{P}$  as well as the constraints on *OS*<sup>3</sup>. Again, all temporal formulae are assumed to be universally quantified over all values of their free variables.

### 2.3.1 Specification of *USER* (the Given)

This part specifies the behavior of the *USER* program at its interface with *OS* and also specifies some properties that are guaranteed when *USER* is composed with *OS*.

For each *user*, the variable *state* is initialized to *active* and *flag* to *null*. A *user* may not start an interaction — (u1). Provided that an *os* may not terminate an interaction and an *os* may not change the state of a *user* ((o1), (o2.1), and (o2.2) in Section 2.3.3), *USER* will satisfy the following two properties: An idle *user* may become active only after it becomes engaged — (u2) and a started interaction will eventually be terminated — (u3).

$$(flag_i = null) \wedge \circ(label \in Rule(USER)) \Rightarrow \circ(flag_i = null) \text{ in } \mathcal{P} \quad (u1)$$

If *OS* satisfies (o1), (o2.1), and (o2.2) in Section 2.3.3, then

$$idle_i \text{ Unless } engaged_i \text{ in } \mathcal{P} \quad (u2)$$

$$start^I \Rightarrow \Diamond \neg start^I \text{ in } \mathcal{P} \quad (u3)$$

<sup>3</sup>We omit the exact temporal specification of the initial states as well as the precise specification of other restrictions (e.g. the restriction on what variables are shared among different modules). Although the omitted specifications can easily be verified from the program text, their explicit inclusion would considerably lengthen the problem specification and the impossibility proofs.

### 2.3.2 Specification of $\mathcal{P}$ (the Composite)

This part specifies the safety and liveness properties that must be provided by the composition of *USER* and *OS*.

The safety properties require that only *enabled* interactions can be started — (pp1), i.e. the synchronization requirement, and that conflicting interactions cannot be started simultaneously — (pp2), i.e. the mutual exclusion requirement. The liveness property requires that if an interaction *I* is *enabled*, either *I* or a conflicting interaction be eventually started — (pp3), i.e. the progress requirement.

$$\neg \text{start}^I \text{ Unless } \text{enable}^I \text{ in } \mathcal{P} \quad (\text{pp1})$$

$$E[I, J] \Rightarrow \neg(\text{start}^I \wedge \text{start}^J) \text{ in } \mathcal{P} \quad (\text{pp2})$$

$$\text{enable}^I \Rightarrow \diamond(\text{start}^I \vee (\exists J : E[I, J] \wedge \text{start}^J)) \text{ in } \mathcal{P} \quad (\text{pp3})$$

### 2.3.3 Constraints on *OS* (the Solution)

The only shared variables between *user<sub>i</sub>* and *os<sub>i</sub>* are *state<sub>i</sub>* and *flag<sub>i</sub>*. For each *os*, *state* is initialized to *active* and *flag* to *null* (consistent with the initialization in *USER*). An *os* may not terminate an interaction — (o1) and an *os* may not change the state of a *user* — (o2.1) and (o2.2).

$$(\text{flag}_i = I) \wedge \circ(\text{label} \in \text{Rule}(\text{OS})) \Rightarrow \circ(\text{flag}_i = I) \text{ in } \mathcal{P} \quad (\text{o1})$$

$$\text{active}_i \wedge \circ(\text{label} \in \text{Rule}(\text{OS})) \Rightarrow \circ \text{active}_i \text{ in } \mathcal{P} \quad (\text{o2.1})$$

$$\text{idle}_i \wedge \circ(\text{label} \in \text{Rule}(\text{OS})) \Rightarrow \circ \text{idle}_i \text{ in } \mathcal{P} \quad (\text{o2.2})$$

(Assumption (ii) in Section 2.1 is enforced by (o2.1) and (o2.2).)

#### Remark

The specification in Sections 2.3.1–2.3.3 formalizes the problem description in Section 1.2.2. It is intended to be a general abstraction of the problem of implementing nondeterministic synchronous communications among asynchronous processes in a distributed system. It does not place any restrictions on the membership of an interaction or on the allowable state transitions of an active process. In particular, a process that never becomes idle and a process that will always eventually become idle will both satisfy the specification, enforcing assumption (i) in Section 2.1. Property (u3) might appear to be unnecessary; the rationale behind this restriction is explained in the final remark of Section 2.4.2.

A number of algorithms have been designed, e.g. [Cha88, Bag89a], for the problem as specified in the preceding sections. This indicates that the specification is “consistent,” or more precisely, for any *USER* (satisfying the specification of *USER*) there exists an *OS* (satisfying the constraints on *OS*) such that their

composition satisfies the specification of  $\mathcal{P}$ .

### 2.3.4 Additional Properties: Fairness

We formally specify the two fairness properties introduced in Section 1.2.3: SIF and SPF.

$$\text{SIF} \equiv \Box \Diamond \text{enable}^I \Rightarrow \Box \Diamond \text{start}^I$$

$$\text{SPF} \equiv \Box \Diamond \text{ready}_i \Rightarrow \Box \Diamond \text{engaged}_i, \text{ where } \text{ready}_i \equiv (\exists I : i \in I \wedge \text{enable}^I)$$

Recall that “SPF in  $\mathcal{P}$ ”, or simply SPF, subsumes (pp3).

## 2.4 Main Results

Given an additional property  $\varphi$ , a *USER* is said to be  $\varphi$ -compatible if there exists an *OS* such that their composition satisfies the specification of  $\mathcal{P}$  and also the additional property  $\varphi$ ; otherwise the *USER* is  $\varphi$ -incompatible. We prove that a *USER* is  $\varphi$ -incompatible by showing that, for any *OS* such that the composition of *USER* and *OS* satisfies the specification of  $\mathcal{P}$ ,  $\text{Comp}^*(\mathcal{P})$  always contains some sequence violating  $\varphi$ . We shall use this approach to prove that there are SIF-incompatible instances for the binary interaction problem and there are SPF-incompatible instances for the multiway interaction problem. As a consequence, SIF is in general impossible for binary or multiway interactions and SPF is in general impossible for multiway interactions.

We start with some general properties of distributed programs (or systems).

### 2.4.1 Some Characteristics of Distributed Systems

Consider a distributed program  $\mathcal{D}$  that models a distributed system. Let  $Q$  be the composition of some modules in  $\mathcal{D}$  and  $\bar{Q}$  be the composition of some other modules such that  $Q$  and  $\bar{Q}$  do not share any variables.  $s[Q]$  denotes the projection of program state  $s$  on  $Q$ , i.e. the local state of  $Q$  at  $s$ . The following two lemmas describe conditions under which the projections of (possibly different) states of  $\mathcal{D}$  on  $Q$  are equivalent. These results capture the ideas behind fusion of computations in [Cha86], which is one of the basic techniques in our impossibility proofs and in others, e.g. [Fis85].

**Lemma 1** *If the local states of  $Q$  corresponding to two program states  $s$  and  $s'$  are the same, the execution of a sequence of rules in  $Q$  starting respectively from  $s$  and  $s'$  will also result in identical local states of  $Q$ . In other words, if*



$(s[Q] = s'[Q]) \wedge x \in \text{Pref}(Q)$ , then  $(s, x)[Q] = (s', x)[Q]$ .

*Proof.* According to our model, the execution of a rule of a program results in a deterministic state transition of the program. Starting from the same state, a program will reach a unique state after the execution of the same sequence of rules. Since  $Q$  is also a program, the lemma follows. *End of Proof.*

**Lemma 2** *The execution of a sequence of rules in  $\bar{Q}$  has no effect on the local state of  $Q$ . In other words, if  $(s[Q] = s'[Q]) \wedge x \in \text{Pref}(\bar{Q})$ , then  $s[Q] = (s', x)[Q]$ .*

*Proof.* From the assumption,  $Q$  and  $\bar{Q}$  do not share any variables. Also, by the definition of program, rules in  $\bar{Q}$  may only reference variables in  $\bar{Q}$  and cannot change the value of any variable in  $Q$ . The lemma follows immediately. *End of Proof.*

Lemma 2 has the following application: According to the problem specification, any pair of processes in  $\mathcal{P}$  communicate via channels and do not share variables. The execution of a rule or a sequence of rules in  $p_i$  will not change the values of the variables in any  $p_j$  ( $j \neq i$ ); an update of  $\text{state}_i$  in any step is not observed by any  $p_j$  ( $j \neq i$ ) in the immediately following step. In general, the preceding applies to the composition of some **users** and **os**'s and the composition of some other **users** and **os**'s, as these two compositions do not share any variables (they communicate with each other through the channels with which they share message queues).

## 2.4.2 Impossibility Proofs

In the following proofs, we consider  $\mathcal{BIN}$ , a collection of instances of the binary interaction problem in which  $USER$  has three processes  $\text{user}_i$ ,  $\text{user}_j$ , and  $\text{user}_k$  and  $\mathcal{I} = \{I, J, K\}$ , where  $I = \{i, j\}$ ,  $J = \{j, k\}$ , and  $K = \{k, i\}$ .

Besides (u1)–(u3), the three processes also satisfy the following properties: It is always possible for an active **user** to become idle due to the execution of some rule (there can be more than one such rule) belonging to the same **user** — (u4) and, if the  $OS$  satisfies its constraints, it is always possible that an active **user** never becomes idle — (u5). Assumption (iv) in Section 2.1 is enforced by (u4) and (u5); it is straightforward to construct a  $USER$  program that satisfies the two properties.

$\text{active}_i \Rightarrow E_{\circ}(\text{label} \in \text{Rule}(\text{user}_i) \wedge \text{idle}_i)$  in  $\mathcal{P}$ , analogously for  $\text{user}_j$  and  $\text{user}_k$ . (u4)

If  $OS$  satisfies (o1), (o2.1), and (o2.2) in Section 2.3.3, then

$$active_i \Rightarrow E\Box active_i \text{ in } \mathcal{P}, \text{ analogously for } user_j \text{ and } user_k. \quad (u5)$$

The following lemma formally establishes the first key observation in Section 2.1, i.e. if some interaction is enabled, the schedulers cannot indefinitely postpone the execution of the interaction while waiting for other interactions to become enabled.

**Lemma 3** *For any instance in  $BIN$ , if at some state interaction  $I$  is enabled,  $p_k$  is active (so, interactions  $J$  and  $K$  are disabled), and no interaction is started, then there exists a possible future of the state in which interaction  $I$  is started and  $p_k$  remains active until  $I$  is started. In other words:*

$$\begin{aligned} & (enable^I \wedge active_k \wedge \neg start^+) \Rightarrow \\ & E((enable^I \wedge active_k \wedge \neg start^+ \text{ Unless } start^I) \wedge \Diamond start^I) \text{ in } \mathcal{P}, \end{aligned}$$

where  $start^+$  denotes  $(start^I \vee start^J \vee start^K)$ .

*Proof.* Assuming the contrary, we shall demonstrate a possible future of a state satisfying  $(enable^I \wedge active_k \wedge \neg start^+)$  such that interaction  $I$  remains enabled but neither  $I$ ,  $J$ , nor  $K$  will be started, violating (pp3).

$$\begin{aligned} \exists \sigma : \sigma \in Comp^*(\mathcal{P}) \wedge ((enable^I \wedge active_k \wedge \neg start^+) \wedge \\ A\neg((enable^I \wedge active_k \wedge \neg start^+ \text{ Unless } start^I) \wedge \Diamond start^I) \mid \sigma) \\ , \text{ from the assumption and definitions (P1), (A5.2), and (A6.3).} \end{aligned}$$

Fix the above  $\sigma$ .

$$\begin{aligned} \forall \tau : \tau \in Branch(\sigma_0) \Rightarrow ((enable^I \wedge active_k \wedge \neg start^+) \wedge \\ \neg((enable^I \wedge active_k \wedge \neg start^+ \text{ Unless } start^I) \wedge \Diamond start^I) \mid \tau) \\ , \text{ from the above, (T2), and (A5.1).} \quad (1) \end{aligned}$$

$$\begin{aligned} \forall \tau : \tau \in Branch(\sigma_0) \Rightarrow (enable^I \wedge active_k \wedge \neg start^+ \mid \tau) \\ , \text{ from the above.} \quad (2) \end{aligned}$$

$$\begin{aligned} \forall \tau : \tau \in Branch(\sigma_0) \Rightarrow \\ (\neg(enable^I \wedge active_k \wedge \neg start^+ \text{ Unless } start^I) \vee \Box \neg start^I \mid \tau) \\ , \text{ from (1), (A4.2), and (A6.2).} \quad (3) \end{aligned}$$

We deviate to prove the following property:

$$\begin{aligned} (enable^I \wedge active_k \wedge \neg start^+ \text{ Unless } start^I) \vee \\ (enable^I \wedge active_k \wedge \neg start^+ \text{ Unless } start^J \vee start^K \vee idle_k) \text{ in } \mathcal{P}. \quad (4) \\ \neg start^I \text{ Unless } start^I \text{ in } \mathcal{P}, \text{ analogously for } J \text{ and } K \end{aligned}$$

, from that an interaction is either started or not started.

$$\begin{aligned} \neg start^+ \text{ Unless } start^+ \text{ in } \mathcal{P} \\ , \text{ (T5) on the above.} \end{aligned}$$

$(s_1', r)[os_i \parallel os_j]$ .

Since  $start^I$  at  $(s_1, y_1)$ , the preceding statement implies  $start^I$  at  $(s_1', r)$ . (1)

Also, from Lemma 2,  $idle_k$  at  $s_1'$  implies  $idle_k$  at  $(s_1', r)$ . (2)

The above scenario is depicted in Figure 2.5, which shows that the transition to idle of  $user_k$  (execution of  $y_2$ ) has no effect on starting interaction  $I$  (by the execution of  $r$ ), as the local state transition of  $user_k$  is not immediately visible to  $os_i \parallel os_j$ .  $I$  is started irrespective of whether  $user_k$  is active or idle.

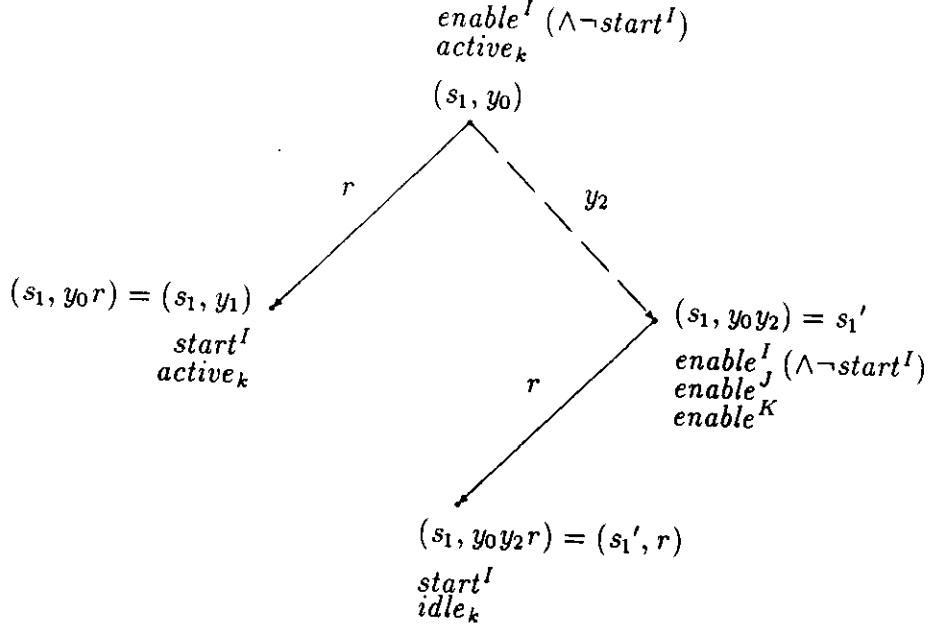


Figure 2.5: The transition to idle of  $user_k$  has no effect on starting interaction  $I$  ( $= \{i, j\}$ )

Those rules of  $\mathcal{P}$  except  $user_i$  and  $user_j$ , not selected in the sequence  $y_0y_2r$  can form an arbitrary sequence  $y_3$ . As  $y_3$  does not contain any rules of  $user_i$  or  $user_j$ , from Lemma 2 and (1),  $start^I$  at all states in  $\langle s_1', ry_3 \rangle$ ; this implies  $\neg start^K$  at all states in  $\langle s_1', ry_3 \rangle$ , due to (pp2). According to (o2.2), (u2), and (2),  $idle_k$  at  $(s_1', ry_3)$ . Let  $s_2 = (s_1', ry_3)$ . So,

$$(start^I \wedge idle_k) \text{ at } s_2. \quad (3)$$

**Stage 3:**  $user_i$  becomes active and  $user_j$  becomes idle after a number of state transitions. Consequently, interaction  $J$  is enabled.

Interaction  $I$  was started in stage 2 at  $(s_1', r)$ . By virtue of (u2),  $user_i$  and  $user_j$  go from *idle* to *active*; by (u3), interaction  $I$  is terminated. Similar to Stage 1, apply (u4) to  $user_j$  such that  $user_j$  becomes idle again; however  $user_i$

remains active. Let  $z$  be the corresponding sequence and  $s_3 = (s_2, z)$ .  $idle_j$  at  $s_3$  and  $active_i$  at  $s_3$ . From (3) and Lemma 2,  $idle_k$  at  $s_3$ . As  $z$  does not involve rules in  $os$ 's, no other interaction is started. In summary,  $(enable^J \wedge active_i \wedge \neg start^+)$  at  $s_3$ .

Stage 4: Similar to Stage 2 (with  $J$  and  $i$  substituting for  $I$  and  $k$ , respectively), interaction  $J$  is started and, similar to Stage 3, both  $user_j$  and  $user_k$  eventually become active. Let  $w$  be the sequence and  $s_4 = (s_3, w)$ . All processes are active and no interaction is started at  $s_4$ .

All interactions are enabled in Stage 2 and interaction  $J$  is enabled in Stage 3. Interaction  $I$  is started in stage 2 and interaction  $J$  is started in Stage 4; while interaction  $K$  is never started. Repeat the four stages indefinitely, we obtain an infinite sequence  $\alpha$  such that  $(\Box \Diamond enable^K \wedge \Box \neg start^K) \mid (s_0, \alpha)$ . Each rule in  $\mathcal{P}$  is selected at least once either in Stage 1 or Stage 2, so  $\alpha \in Rule^*(\mathcal{P})$  and  $\langle s_0, \alpha \rangle \in Comp^*(\mathcal{P})$ . *End of Proof.*

Add one process  $user_l$ , which has properties (u1)–(u5), to each *USER* in  $\mathcal{BIN}$  and change  $K$  to  $\{k, i, l\}$ . We obtain a collection  $\mathcal{MUL}$  of instances of the multiway interaction problem.

**Theorem 2** *USERS in  $\mathcal{MUL}$  are SPF-incompatible. (So, in general, SPF is impossible for the multiway interaction problem.)*

*Proof.* At some point of computation, assume that  $p_l$  becomes idle, while other processes remain active. Since  $p_l$  may participate only in interaction  $K$ , in order to satisfy SPF, interaction  $K$  should be started infinitely often if it is enabled infinitely often. Ignore  $p_l$  altogether and treat this problem as implementing SIF for the equivalent binary interaction problem. The conclusion follows from the constructed computation in the proof of Theorem 1. *End of Proof.*

### Remark

Property (u3) was used in Stage 3 of the proof of Theorem 1. Removal of this property from the specification may trigger some superficial impossibility results. For instance, assume that some interaction is started but never terminated while its members remain idle. A conflicting interaction may become and remain continuously enabled but can never be started (to ensure mutual exclusion, i.e. (pp2)). This violates a fairness property referred to as weak interaction fairness. The problem can be avoided by modifying the specification to introduce an additional state for a process (other than *active* or *idle*) which refers to a process that is participating in an interaction. However, this modification makes the specification longer and more complex.

$active_k$  Unless  $idle_k$  in  $\mathcal{P}$

, from that a process is either *active* or *idle*.

$enable^I$  Unless  $start^+$  in  $\mathcal{P}$

, (T5) on (u2).

$enable^I \wedge active_k \wedge \neg start^+$  Unless  $start^+ \vee idle_k$  in  $\mathcal{P}$

, (T5) on the above three.

Applying (T4) to the above, we obtain (4).

$\forall \tau : \tau \in Branch(\sigma_0) \Rightarrow$

$((enable^I \wedge active_k \wedge \neg start^+ \text{ Unless } start^J \vee start^K \vee idle_k) \vee \Box \neg start^I \mid \tau)$   
, from (3) and (4). (5)

$\exists \tau' : \tau' \in Branch(\sigma_0) \wedge (\Box active_k \mid \tau')$  (the second conjunct =  $(\Box \neg idle_k \mid \tau')$ )

, from (u5), (2), and (A5.2). (6)

Fix the above  $\tau'$ .

$\neg start^J \wedge \neg start^K \wedge \Box(\neg enable^J \wedge \neg enable^K) \mid \tau'$

, from the above, (d2), and (2).

$\Box(\neg start^J \wedge \neg start^K) \mid \tau'$

, from the above, (pp1), and (A7.2).

$(enable^I \wedge active_k \wedge \neg start^+) \wedge \Box(\neg start^J \wedge \neg start^K \wedge \neg idle_k) \mid \tau'$

, from (2), the above, and (6).

$enable^I \wedge \Box(\neg start^I \wedge \neg start^J \wedge \neg start^K) \mid \tau'$

, from the above, (5), and (A7.2).

The above violates (pp3).

*End of Proof.*

**Theorem 1** *USERS in BLN are SIF-incompatible. (So, in general, SIF is impossible for the binary or multiway interaction problem.)*

*Proof.* Starting from a state of  $\mathcal{P}$  where all processes are active and no interaction is started (initial states are such states), we are able to construct an infinite sequence of rules  $\alpha$  satisfying the fair selection criterion, i.e.  $\alpha \in Rule^*(\mathcal{P})$ , such that in the corresponding computation interaction  $K$  is enabled infinitely often but never started. Formally,  $\exists \sigma : \sigma \in Comp^*(\mathcal{P}) \wedge ((\Box \Diamond enable^K \wedge \Box \neg start^K) \mid \sigma)$ . The construction proceeds in phases, where each phase consists of four stages. During each phase all interactions are enabled at least once but only  $I$  and  $J$  are started. At the end of each phase the program reaches a state where all processes again become active and no interaction is started. To satisfy the fair selection criterion, each rule in  $\mathcal{P}$  is selected at least once in each phase. Figure 2.4 outlines the major state transitions in various stages of a phase.

Starting from a state  $s_0$  where all processes are active and no interaction is started, each phase proceeds as follows:

← Stage 1 ←	Stage 2		Stage 3		Stage 4		→
$s_0$	$s_1$		$s_2$		$s_3$		$s_4$
$active_i$ $active_j$ $active_k$ $\neg start^+$	$enable^I$ $active_k$ $\neg start^+$	$enable^I$ $enable^J$ $enable^K$ $\neg start^+$	$enable^I$ $enable^J$ $enable^K$ $start^I$	$active_i$ $active_j$ $idle_k$ $\neg start^+$	$active_i$ $enable^J$ $\neg start^+$	$active_i$ $enable^J$ $start^J$	$active_i$ $active_j$ $active_k$ $\neg start^+$

Figure 2.4: Major state transitions in a phase of the computation under construction

Stage 1:  $user_i$  and  $user_j$  become idle (so, interaction  $I$  is enabled), while  $user_k$  remains active.

Apply (u4) first to  $user_i$  then to  $user_j$  (or the other way around) to obtain a sequence  $x_1$  consisting of rules of  $user_i$  and  $user_j$  such that  $(idle_i \wedge idle_j)$  at  $(s_0, x_1)$ . From (d2),  $enable^I$  at  $(s_0, x_1)$ . Those rules of  $user_i$  and  $user_j$  not selected can be arranged in arbitrary order to form a sequence  $x_2$ . Let  $s_1 = (s_0, x_1 x_2)$ . As  $x_1 x_2$  contains rules from only  $user_i$  and  $user_j$ , due to (u1) and (u2), no interaction is started hence  $enable^I \wedge \neg start^+$  at  $s_1$ ; and, since  $active_k$  at  $s_0$  and  $x_1 x_2$  does not contain any rules in  $user_k$ , from Lemma 2, we get  $active_k$  at  $s_1$ .

Stage 2: Interaction  $I$  is started;  $user_k$  becomes idle just before  $I$  is started such that  $I$ ,  $J$ , and  $K$  are enabled simultaneously. However, it is impossible for the schedulers to determine that  $J$  and  $K$  are enabled before  $I$  is started (the second key observation in Section 2.1).

Given  $(enable^I \wedge active_k \wedge \neg start^+)$  at  $s_1$ , from Lemma 3, there exists a sequence  $y_1$  of rules in  $\mathcal{P}$  such that  $start^I$  at  $(s_1, y_1)$  and  $\neg start^J \wedge \neg start^K$  at all states in  $\langle s_1, y_1 \rangle$ . Without loss of generality, we assume  $y_1 = y_0 r$ , where  $r$  is a rule of  $os_i$  or  $os_j$ , and  $\neg start^I$  at  $(s_1, y_0)$ , i.e. the execution of  $r$  starts interaction  $I$ . From (pp1) and  $\neg start^I$  at  $(s_1, y_0)$ ,  $enable^I$  at  $(s_1, y_0)$ . According to (u4), there exists some rule or in general a sequence of rules  $y_2$  in  $user_k$  such that  $idle_k$  at  $(s_1, y_0 y_2)$ . (If  $idle_k$  at some state in  $\langle s_1, y_1 \rangle$ , then  $y_2$  is simply the empty sequence.) Let  $s_1' = (s_1, y_0 y_2)$ . So,  $(enable^I \wedge enable^J \wedge enable^K)$  at  $s_1'$ . Rule  $r$  can be selected for execution at  $s_1'$ .

From Lemma 2 (replacing  $s$  and  $s'$  in the lemma by  $(s_1, y_0)$ ,  $Q$  by  $os_i \parallel os_j$ ,  $\bar{Q}$  by  $os_k$ , and  $x$  by  $y_2$ ),  $(s_1, y_0)[os_i \parallel os_j] = ((s_1, y_0), y_2)[os_i \parallel os_j]$ , i.e.  $(s_1, y_0)[os_i \parallel os_j] = s_1'[os_i \parallel os_j]$ , which is to say that the transition to idle of  $user_k$  did not change the local state of  $os_i \parallel os_j$  at  $(s_1, y_0)$ .

From Lemma 1 (replacing  $s$  by  $(s_1, y_0)$ ,  $s'$  by  $s_1'$ ,  $Q$  by  $os_i \parallel os_j$ , and  $x$  by  $r$ ),  $((s_1, y_0), r)[os_i \parallel os_j] = (s_1', r)[os_i \parallel os_j]$ . As  $y_1 = y_0 r$ , we get  $(s_1, y_1)[os_i \parallel os_j] =$

## 2.5 Discussion

### 2.5.1 Generalization of Main Results

Two factors determine whether a problem instance is SIF and/or SPF-*incompatible*: (a) the properties exhibited by the users in addition to (u1)-(u3) and (b) the “system configuration”, i.e. the number of processes and the interactions defined among the processes. In the previous section, we proved the impossibility results for problem instances in  $BIN$  and  $MUL$  by assuming that every user satisfies the additional properties (u4) and (u5). In this section, we extend the applicability of the results first by weakening (u4) and (u5) and subsequently by considering system configurations other than those of  $BIN$  and  $MUL$ .

Property (u4) may be weakened to allow an active user to become idle on the execution of a sequence of rules (instead of one rule) belonging to the same user. The results and proofs in Section 2.4.2 would still be valid; however, a precise specification of the weaker version will be tedious in the temporal logic adopted here. Note that both (u4) and the weaker version capture the essence of the assumption that an active user becomes idle autonomously.

Property (u5) states that it is always possible that an active user never becomes idle. As the proofs do not make use of this property for  $user_j$ , the results in Section 2.4.2, particularly Lemma 3, are still valid if only  $user_i$  and  $user_k$  are assumed to satisfy (u5). The requirement that every user satisfy (u5) may be further weakened as follows: Assume that either  $user_i$  or  $user_k$  satisfies (u5) but which of them does is not known. Lemma 3 still holds and, as a consequence, Theorems 1 and 2 will also hold. However, the proof of Lemma 3 would involve game-playing arguments that assume an adversary; the current proof of Lemma 3, based only on the computational model and the temporal logic, will not be sufficient. It is also possible to assume a property weaker than (u5) for both  $user_i$  and  $user_k$ . For example, it could be the case that a user is guaranteed to make only a finite number of transitions from active to idle. We believe that Theorems 1 and 2 are still true, though Lemma 3 is no longer true and the proofs of the theorems are no longer valid. However, the adopted temporal framework is inadequate for developing a precise specification of the weaker assumption as well as the required proofs.

We now consider the impact of expanding the system configuration. The interaction graphs of problem instances in  $BIN$  and  $MUL$  are shown again in Figures 2.6 (a) and (b) respectively, where we identify  $j'$  with  $j$ .

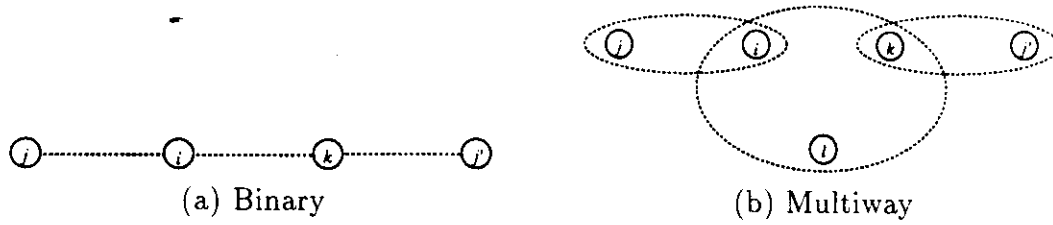


Figure 2.6: “Minimal” interaction graphs ( $j$  and  $j'$  may be identical)

Assume that  $j$  and  $j'$  in Figures 2.6 (a) and (b) are distinct processes and that the interaction graphs represent problem instances in  $\mathcal{BIN}'$  and  $\mathcal{MUL}'$  respectively. Also assume that all users satisfy (u1)–(u5). Let  $J'$  denote interaction  $\{j', k\}$ . It is easy to show that Theorems 1 and 2 apply to problem instances in  $\mathcal{BIN}'$  and  $\mathcal{MUL}'$  respectively: the proofs must simply be modified to replace interaction  $J (= \{j, k\})$  by interaction  $J' (= \{j', k\})$ ; furthermore, references to  $\text{user}_j$  in the context of interaction  $J$  must be replaced by  $\text{user}_{j'}$  in the context of interaction  $J'$ . The interaction graph in Figure 2.6 (a) is “minimal” in the sense that an instance of the problem with a simpler interaction graph (with fewer processes or interactions) will become *SIF-compatible*. We show the minimality of  $\mathcal{BIN}'$  by demonstrating that the removal of any edge leads to a problem instance for which SIF can be satisfied; the minimality of Figure 2.6 (b) can be shown analogously.

Assume  $\{j', k\}$  is removed from Figure 2.6 (a) (the consequence of removing another edge may be argued similarly). We must show that if some interaction in the reduced graph is enabled infinitely often, it will be executed infinitely often. Using a construction similar to that used in the proof of Theorem 1, assume that at some point in the computation, interaction  $\{i, j\}$  is started after  $p_k$  becomes idle. As interaction  $\{i, k\}$  is the only interaction of which  $p_k$  is a member, the OS can easily satisfy SIF by ensuring that if  $p_i$  subsequently becomes idle,  $\{i, k\}$  is the next interaction to be started.

The minimality of  $\mathcal{BIN}'$  and  $\mathcal{MUL}'$  can be used to extend the impossibility results to other process configurations: Any problem instance with interaction graph reducible to Figure 2.6 (a) is *SIF-incompatible* and one reducible to Figure 2.6 (b) is *SPF-incompatible* provided that every user satisfies (u1)–(u4) and at least  $\text{user}_i$  and  $\text{user}_k$  satisfy (u5). (A graph is said to be *reducible* to another graph if the latter graph is a result of removing some of the nodes or edges from the former.)



## 2.5.2 Basic Assumptions Revisited

We restate the basic assumptions of the problem and show that they are motivated by practical considerations:

- (A) It is impossible to determine *a priori* whether an arbitrary process will make a transition from active to idle.
- (B) The scheduler cannot control the actions of an active process. In particular, the scheduler cannot control when an active process becomes idle.
- (C) The transition from *active* to *idle* of a process is not immediately observable by other processes or their schedulers.

The validity of assumption (A) can be proven by arguments similar to those used to demonstrate the undecidability of the halting problem. For a set of processes with restricted behavior wherein it is possible to assume that every active process will eventually become idle, SIF can easily be guaranteed. Assume that some total order is assigned to the set of interactions. Given that at any point of a computation each active process will eventually become idle, every interaction must eventually become enabled. A scheduler may then simply choose each interaction in turn and wait until it is enabled, implying that the complexity of such algorithms will be dependent on the average time each process remains in the active state.

Contrary to assumption (B), it is possible to assume a more powerful scheduler which is responsible for scheduling both local and communication actions of a process. This would imply that the transition of a process from active to idle can also be controlled by the scheduler and indirectly by other processes in the system, thus violating the autonomy of a process in executing a local action. Such a scheduler can prevent a process from executing its active to idle transition, thus allowing it to control which interactions are enabled. In the extreme case, such a scheduler can always ensure that conflicting interactions are never enabled simultaneously and thus guarantee SIF in a straightforward manner. However, such a powerful scheduler is just an artifact that defines away the real problem. Furthermore, postulating such a powerful scheduler, in effect, implies that a scheduler has an instantaneous “global snapshot” about which interactions are enabled, a requirement that is met by very few real-life distributed systems.

Assumption (C) is a consequence of unbounded communication delays in asynchronous systems. Moreover, SIF is impossible even if a known upper bound

(other than zero) is assumed for communication delays. If the active to idle transition of a process is immediately observable by other processes in the system, SIF can be guaranteed, as once again the scheduler has an instantaneous global snapshot of the enabled interactions.

Under the three assumptions, the impossibility results hold in a model which allows more than one atomic actions (or rules) to be executed in each computation step in contrast to the interleaving model assumed here; this follows immediately, as the former model will contain all computations allowed by the latter model. Also, it should be clear that any fairness property stronger than SPF will be impossible for multiway interactions and any fairness property stronger than SIF will be impossible for binary interactions. For example, a fairness property, which requires that two interactions which are enabled equally many times be started equally many times, is impossible for binary (and hence multiway) interactions. Interested readers are referred to [Fra86a, Apt88] for stronger variations of SIF and SPF.

## CHAPTER 3

### Fair Algorithms

In this chapter, we present solutions to the binary interaction problem that satisfy SPF. We show that algorithms based on the usual transformation from the process interaction problem to the dining philosophers algorithm cannot guarantee SPF. We then propose a transformation that ensures SPF and derive algorithms based on this transformation.

#### 3.1 Algorithm A

##### 3.1.1 Informal Description

The basic idea of the algorithm is as follows: A single token is associated with each interaction. Each token (and the corresponding edge in the interaction graph) is assigned an identification number, or id, using a minimal proper edge-coloring of the interaction graph; at most  $(D + 1)$  id's are needed [Ber76]. A token is initially assigned arbitrarily to one of the two processes named in the corresponding interaction; tokens held by a process are stored in a token queue. A process may request an interaction only if it has the corresponding token; it does so by sending the token to the other process. Upon receiving a request (token), the requested process captures the token and either commits to the request by sending a *yes* message, denies the request by sending a *no* message, or delays the request. A token captured by a process is appended at the end of its token queue.

A process requests an interaction if (a) it is idle, (b) it has not committed to any request, (c) it is not participating in any interaction, (d) it holds the corresponding token, and (e) it does not have a pending request, i.e. it has not sent a **request** for which a reply has not been received. For brevity, a process satisfying conditions (a)–(c) will be referred to as a *waiting* process. As we prove subsequently, every process must eventually receive a *yes* or *no* message in response to its request. If it receives a *yes* message, it starts the corresponding interaction. If it receives a *no* message and if conditions (a)–(e) still hold, it will request another interaction by sending the corresponding token.

A process selects tokens from its token queue according to the following strat-

egy: Every time the process becomes waiting, the first interaction it requests is determined by the token at the head of its token queue; subsequent tokens, if needed, are selected from the queue in decreasing order of their id's. The reason of adopting this strategy will become clear in the proof of SPF and in the performance analysis.

A process commits to an incoming request if it is waiting and it does not have any pending request. A process delays a request if it has a pending request (which implies it is also waiting, as a process sends a request only if it is waiting), it has not delayed any other request, and the id of the incoming request is *greater than or equal to* that of the pending request. The process subsequently denies the delayed request if it receives a *yes* message in response to its pending request and it commits to the delayed request if it receives a *no*. A process denies a request immediately if it can neither commit to nor delay the request.

The basic ideas of Algorithm A are summarized in Figure 3.1, while the order of sending tokens is illustrated in Figure 3.2.

### 3.1.2 The Algorithm

The following variables and transition rules are defined for the scheduler of each process  $p_i$  (subscripts of variables are omitted when no confusion may arise):

VARIABLES:

$\text{flag}_i$ : Interaction  $\{i, j\}$  is started if  $\text{flag}_i = \{i, j\}$  or  $\text{flag}_j = \{i, j\}$ .

$\text{token\_q}$ : a queue of tokens. Each token is an unordered pair of process id's.

$\text{ino}[\{i, j\}]$ : id of interaction  $\{i, j\}$ .

$\text{pend}_i$ : If  $p_i$  has a pending request, say  $\{i, j\}$ ,  $\text{pend}_i$  is set to  $j$ .

$\text{delay}_i$ : id of the process, if any, whose request has been delayed by  $p_i$ .

$\text{commit}_i$ : ( $\text{commit}_i = j$ ) indicates that  $p_i$  has committed to a request from  $p_j$ .

INITIALIZATION:

1. Edge-color the interaction graph with positive integers  $\leq (D + 1)$ . Assign  $\text{ino}[\{i, j\}]$  the color (id) of edge  $\{i, j\}$ .
2. Assign each token arbitrarily to one of the two processes in the corresponding interaction.
3. All tokens initially assigned to a process are stored (in an arbitrary order) in its  $\text{token\_q}$ .
4. Variables  $\text{flag}$ ,  $\text{pend}$ ,  $\text{delay}$ , and  $\text{commit}$  of each process are initialized to *null*.

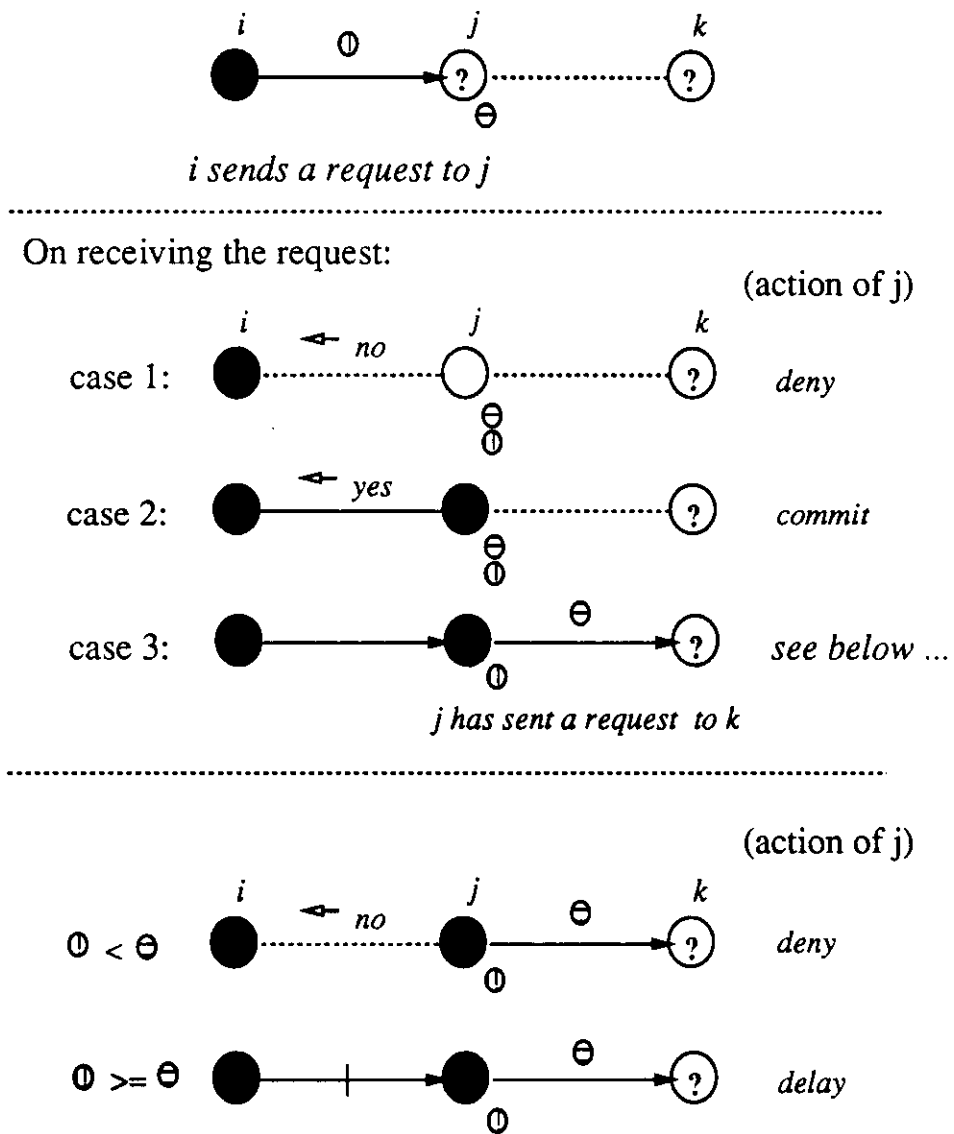


Figure 3.1: Basic idea of Algorithm A

Incoming Request



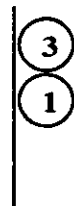
Outgoing Request



*(2 was at the head)*



*(6 is the largest)*



*(4 is the largest)*



*(3 is the largest)*

Figure 3.2: The order of sending tokens by a process in Algorithm A

### TRANSITION RULES:

(For brevity, the variable *pend* will also be used as a boolean expression whose value is *true* if and only if (*pend*  $\neq$  *null*); analogously for other variables. Recall that a process is said to be *waiting* if it is idle, it has not committed to any request, and it is not participating in any interaction; *waiting* is implemented by “(state = *idle*) and  $\neg$ commit and  $\neg$ flag.”)

Rule 1: *waiting* and  $\neg$ pend and  $\neg$ empty(token\_q).

1. **token-selection:** For the first time after the process becomes *waiting*, the token at the head of *token\_q* is selected; subsequent tokens, if needed, are selected from *token\_q* in decreasing order of their id's.
2. Remove the selected token  $\{i,j\}$  from *token\_q* and send it to  $p_j$ .
3. *pend* :=  $j$ .

Rule 2: On receiving a request (token  $\{i,j\}$ ) from  $p_j$ .

1. **if** *waiting* and  $\neg$ pend  
    **then** send a *yes* message to  $p_j$ ; *commit* :=  $j$ .
2. **if** *waiting* and *pend*  
    **then** if  $\neg$ delay and ( $\text{ino}[\{i,j\}] \geq \text{ino}[\{i,\text{pend}\}])$   
        **then** *delay* :=  $j$   
        **else** send a *no* message to  $p_j$ .
3. **if**  $\neg$ *waiting* **then** send a *no* message to  $p_j$ .
4. Append token  $\{i,j\}$  to the end of *token\_q*.

Rule 3: On receiving a *yes* message.

1. *flag* :=  $\{i,\text{pend}\}$ . /\* Interaction  $\{i,\text{pend}\}$  is started. \*/
2. **if** *delay* **then** send a *no* message to  $p_{\text{delay}}$ .
3. *pend* := *null*; *delay* := *null*.

Rule 4: On receiving a *no* message.

1. **if** *delay* **then** send a *yes* message to  $p_{\text{delay}}$ ; *commit* := *delay*.
2. *pend* := *null*; *delay* := *null*.

As the transition from active to idle or from idle to active is done autonomously by the **host** process, the corresponding rules are not included here. For brevity, we have omitted the rules that deal with termination of interactions. A host process terminates an interaction by resetting its **flag** to *null*; its scheduler detects that the **flag** has been reset and notifies the other scheduler by means of an appropriate message, which resets its *commit* to *null*.

### 3.1.3 Correctness Proof

We prove that Algorithm A satisfies the safety and SPF requirements; recall that SPF subsumes the progress requirement.

**Theorem 3** *Algorithm A satisfies the mutual exclusion requirement.*

*Proof.* Assume some interaction  $\{i,j\}$  is started. Without loss of generality, assume that the interaction was requested by  $p_i$  and (as the interaction does start) eventually  $p_j$  sent a *yes* message to  $p_i$ . We prove mutual exclusion by showing that neither  $p_i$  nor  $p_j$  may simultaneously request or commit to another interaction. We first consider  $p_i$ : From Rule 1, after  $p_i$  sends a request to  $p_j$ ,  $\text{pend}_i$  must hold and the guard for Rule 1 ensures that  $p_i$  will not send any other request. Further,  $\text{pend}_i$  requires that  $p_i$  delay or deny subsequent incoming requests (step 2 of Rule 2, or simply Rule 2(2)). As  $p_i$  is assumed to eventually receive a *yes* message from  $p_j$ , Rule 3(2) requires that it must deny any delayed request. After receiving the *yes* message,  $p_i$  is no longer waiting and can neither request interactions nor commit to other requests (Rule 1 and 2(3)).

We next consider  $p_j$ : The *yes* message sent by  $p_j$  is either due to the execution of Rule 2(1), which requires that  $p_j$  be waiting and without a pending request, or due to that of Rule 4(1), which requires  $\text{delay}_j$ , after receiving a *no* message for its pending request. From Rule 2(2),  $\text{delay}_j$  requires that  $p_j$  be waiting. In either case, prior to sending the *yes* message,  $p_j$  must be waiting and must not have pending requests. After sending the *yes* message, it is no longer waiting and can neither request interactions nor commit to other requests (Rule 1 and 2(3)).

*End of Proof.*

**Theorem 4** *Algorithm A satisfies the synchronization requirement.*

*Proof.* Once again, assume interaction  $\{i,j\}$  is started; from Theorem 3, no conflicting interactions can be started. We show that  $p_i$  and  $p_j$  are idle immediately before the interaction is started. Again, assume that the interaction was requested by  $p_i$  and eventually  $p_j$  sent a *yes* message to  $p_i$ . From Rule 1,  $p_i$  must be idle before it sent token  $\{i,j\}$  and must subsequently remain idle until interaction  $\{i,j\}$  is started. A *yes* message is sent by  $p_j$  either on execution of Rule 2(1) which requires that  $p_j$  be idle, or Rule 4(1) which requires  $\text{delay}_j$ . From Rule 2(2),  $\text{delay}_j$  requires that  $p_j$  be idle.

*End of Proof.*

**Lemma 4** *A request will not remain pending forever.*



*Proof.* By Rule 2, a request is either replied immediately or delayed. Assume that delayed requests form a cycle, which must involve at least 2 processes. A cycle of exactly 2 processes is possible only if a pair of processes delay each other's request, i.e. both processes hold a token for the interaction between them: this is impossible because only one token exists for each interaction. A cycle with more than two processes implies that every process in the cycle has delayed the request received from its predecessor process. Due to Rule 2(2), this is possible only if the id's assigned to all these requests are the same. A proper edge-coloring, which ensures all edges incident to a node have distinct id's, implies that this is not possible. Thus a chain of delayed requests can never form a cycle and the last process in a delay chain will either commit to or deny its incoming request immediately. As the computational model guarantees reliable message delivery, we conclude inductively that a reply is generated for every incoming request.

*End of Proof.*

**Lemma 5** *A process that delays a request must eventually participate in some interaction.*

*Proof.* By Rule 2(2), a process that delays a request must have a pending request. From Lemma 4, a reply to its pending request will eventually be received. If a *yes* message is received, by Rule 3 it will participate in the corresponding interaction. Instead, if a *no* is received, by Rule 4 it will commit to the delayed request and will eventually participate in the interaction corresponding to the delayed request.

*End of Proof.*

**Lemma 6** *If a process is continuously waiting, eventually its token<sub>q</sub> must become empty and thereafter remain empty.*

*Proof.* Let  $s_1$  refer to a system state where  $p_i$  becomes waiting and remains waiting continuously thereafter.

Assume that  $\text{token}_{q_i}$  is empty at state  $s_1$ . Subsequently, if  $p_i$  receives a token, from Rule 2(1), it must commit to the request, contradicting the assumption of the lemma.

Assume that  $\text{token}_{q_i}$  is not empty at state  $s_1$ ; let  $t$  refer to the token at the head of  $\text{token}_{q_i}$ . As  $p_i$  remains continuously waiting, from Rule 1(1), it must eventually send  $t$  which must be denied. Let  $s_2$  refer to the system state immediately after  $p_i$  receives the *no* message in response to token  $t$ .

Consider any state  $s_i$  subsequent to  $s_2$ , where  $\text{token}_{q_i}$  is not empty. Let  $T$  refer to the tokens in  $\text{token}_{q_i}$  plus a possible pending request (token) at state

$s_i$  and let  $t'$  be the token with the largest id in  $T$ . We first show that, at any state subsequent to  $s_i$ ,  $p_i$  can never receive a token with id greater than that of  $t'$ . Consider the first such token that it receives. If  $p_i$  does not have a pending request, it must commit to the incoming token (Rule 2(1)) and will not remain waiting forever. If  $p_i$  has a pending request, which must be token  $t'$  (Rule 1(1)), the incoming request with id greater than that of  $t'$  must be delayed (Rule 2(2)) which also implies that  $p_i$  will not remain waiting forever (Lemma 5). We now show that eventually  $p_i$  must lose  $t'$ , i.e.  $t'$  will no longer be in  $T$ . If  $t'$  is its pending request at  $s_i$ , eventually  $t'$  must be denied (as  $p_i$  is assumed to remain waiting continuously). If  $p_i$  does not have a pending request at  $s_i$ , Rule 1 ensures that the next token selected by  $p_i$  must be  $t'$ . As a continuously enabled transition rule must eventually be executed,  $p_i$  will eventually send token  $t'$  to the other process; and the previous argument for  $t'$  as the pending request applies.

Hence at any state subsequent to  $s_2$ , where  $T$  is not empty, eventually  $p_i$  must lose the token with the largest id and can never subsequently capture it again. It follows that the largest id in  $\text{token\_q}_i$  must decrease; and, inductively,  $\text{token\_q}_i$  will eventually become empty. *End of Proof.*

**Theorem 5** *Algorithm A satisfies SPF.*

*Proof.* Assume the contrary, i.e. there exists a process  $p_i$  such that  $p_i$  is ready to participate in some enabled interaction infinitely often, but from some point in time no interaction of which  $p_i$  is a member is started. It follows that eventually  $p_i$  will be continuously waiting. And, from Lemma 6,  $\text{token\_q}_i$  will eventually remain empty.

We claim that if any of the neighbors of  $p_i$ , say  $p_j$ , becomes idle infinitely often, then  $p_j$  will eventually send token  $\{i,j\}$  to  $p_i$ , which will commit to the request if it is waiting, contradicting the assumption that  $p_i$  will be continuously waiting. If  $p_j$  becomes idle infinitely often, it must participate in some interaction infinitely often; otherwise, from Lemma 6,  $\text{token\_q}_j$  also will eventually remain empty, which is not possible. Each time  $p_j$  participates in some interaction, say  $\{j,k\}$ , there is at least one exchange of the corresponding token between  $p_j$  and  $p_k$ . To ensure that interaction  $\{i,j\}$  is eventually requested by  $p_j$ , we need to show that the token for interaction  $\{i,j\}$  will eventually emerge to the head of  $\text{token\_q}_j$ . For simplicity, we assume that when a process becomes waiting, if its  $\text{token\_q}$  is not empty, the process makes at least one request before replying to any incoming request. Thus, the distance of token  $\{i,j\}$  from the head of  $\text{token\_q}_j$  will decrease by at least 1 each time  $p_j$  becomes waiting. It follows that token  $\{i,j\}$  will eventually emerge to the head of  $\text{token\_q}_j$  and be sent by  $p_j$  to  $p_i$ . (In

reality, a waiting process may accept an incoming request before itself makes any request.- This can occur consecutively for at most  $D$  times, as there is only one token for each interaction and a process can be a member of at most  $D$  interactions. Particularly, once a process holds all the tokens for the interactions of which it is a member, it has to send at least one request next time when it becomes waiting.) *End of Proof.*

### Remark

Though we have assumed that message delivery is FIFO, Algorithm A can handle out of order message deliveries. According to the algorithm, at any state there is at most one message in transit between a pair of processes, with one exception: A process  $p_i$  may deny a request  $t_1$  (for instance, if  $p_i$  has a pending request  $t_2$  such that id of  $t_2$  is larger than id of  $t_1$ ) from one of its neighbors  $p_j$  and subsequently  $p_i$  may send  $t_1$  to  $p_j$  (for instance, if  $t_2$  is itself denied). At this point both  $t_1$  and the *no* message may be in transit. Assume that token  $t_1$  is delivered before the *no* message. In this case the prematurely received token will be delayed by  $p_j$  due to Rule 2(2) (assuming it is the case that *waiting* and  $\neg$ delay are true; otherwise,  $p_j$  will deny or commit to  $t_1$  immediately) until the *no* message from  $p_i$  is received. Before then all other requests to  $p_j$  will be denied and, when the *no* message from  $p_i$  is received,  $p_j$  will commit to request  $t_1$  (Rule 4(1)). It follows that the algorithm can handle out of order message deliveries.

## 3.1.4 Performance Analysis

### 3.1.4.1 Worst Case

**Theorem 6** *The message cost of Algorithm A is at most  $2(D + 1)$  and its response time is at most  $D^2 + 5D$ .*

*Proof.* The performance of the algorithm is affected by the number of requests sent by a process and the length of a delay chain that may be formed in the interaction graph.

Suppose  $p_i$  becomes waiting. Let  $t$  and  $T$  be as defined in Lemma 6; token  $t$  is denied and subsequently  $p_i$  sends tokens from  $T$  in decreasing order of their id's.

We claim that each token from  $T$  is sent at most once by  $p_i$  before  $p_i$  participates in some interaction. Assume the contrary, i.e. there exists a  $t_i$  in  $T$  such that  $t_i$  is sent at least twice by  $p_i$  before  $p_i$  participates in some interaction. When  $t_i$  was sent for the first time by  $p_i$ , it must be the token with the largest

id (Rule 1(1)). Subsequently,  $t_i$  is captured by  $p_i$  only when  $p_i$  receives  $t_i$  from its communication partner. On receiving the token, if  $p_i$  delays or commits to the request, some interaction of which  $p_i$  is a member will eventually be started, which contradicts the preceding assumption. Thus  $p_i$  must deny the token. However,  $p_i$  can deny the token only if it has a pending request whose id is greater than  $t_i$  (Rule 2(2)). This is impossible as  $p_i$  must request tokens from  $T$  in decreasing order of their id's, which establishes the necessary contradiction. As a process is a member of at most  $D$  interactions,  $T$  can contain at most  $D$  tokens. Together with the request corresponding to token  $t$ , this implies that  $p_i$  may send at most  $(D + 1)$  requests. As each request generates exactly one reply message, the message cost is at most  $2(D + 1)$ .

An incoming request at some  $p_i$  is delayed if  $p_i$  has a pending request with a smaller id (two interactions with a common member have distinct id's). It is possible for the requests to be made such that a chain of delayed requests is formed in the interaction graph. As a request is delayed only by another request with a smaller id, a request with id  $k$  may introduce a delay chain of length at most  $k$  and will thus be replied within  $2k$  units of time. From the analysis of message cost, at least one of the two members of an enabled interaction will participate in some interaction before they both send  $(D + 1)$  requests. Token  $t$  as in the previous paragraphs may be sent twice. Its id cannot be the largest among those of the tokens that  $p_i$  holds; otherwise, it will be committed to or delayed by  $p_i$  when it is returned. So, within at most  $2(2 + 3 + \dots + D + (D + 1)) + 2D = D^2 + 5D$  units of time after an interaction is enabled, either the interaction or some conflicting interaction will be started. *End of Proof.*

In contrast, the bound on the response time of Sistla's algorithm [Sis84] is  $D^2(T_c + L)$ , where  $T_c$  is the bound on the duration of an interaction and  $L$  is the length of the longest path in the interaction graph.

### 3.1.4.2 Optimality of Message Cost

We argue informally that the message cost of Algorithm A is within a constant difference from optimum.

If  $p_i$  is to start interaction  $\{i, j\}$ , then  $p_i$  must *know* that (1) both  $p_i$  and  $p_j$  are idle and (2) neither of  $p_i$  and  $p_j$  is participating in any other interaction. This implies that  $p_i$  must have received some message from  $p_j$  after  $p_j$  became idle so that  $p_i$  knows  $p_j$  is idle (which, in accordance with the problem description, also implies that  $p_j$  is not participating in any other interaction). We shall refer to this message as a *request* as we did for Algorithm A. An enabled interaction

may remain enabled continuously if both members are waiting for the other to send a request, so at least one of them must “take the initiative” and eventually send a request to the other. A process is said to be *aggressive* for an interaction if it is responsible for taking the initiative; it is *passive* for the interaction otherwise. Both members of an interaction may be aggressive simultaneously for the interaction.

We observe that, if a process receives a negative reply in response to its request, it should become passive for the corresponding interaction, while the requested process becomes aggressive. The message complexity of an algorithm that disobeys this principle is always worse than a similar algorithm that obeys the principle, since the requested process may again deny the second, third ... requests for the same reason as it denied the first. Furthermore, a process that is passive for an interaction should remain passive unless the other member of the interaction sends it a request or itself participates in some interaction. A process aggressive for an interaction should remain aggressive unless it sends a request to the other member of the interaction.

On receiving a request from  $p_j$ ,  $p_i$  must eventually respond to the message: the response may be either positive, if  $p_i$  is idle and is not participating in any other interaction, or negative, if  $p_i$  is not ready to participate in any interaction or is participating in some interaction. Suppose  $p_i$  has  $D$  neighbors. It may happen that, at a certain point of computation,  $p_i$  is idle, not participating in any interaction, and aggressive for all interactions of which it is a member, but all its neighbors happen not to be idle. All requests from  $p_i$  will be denied by its neighbors. The total number of messages that should be charged to  $p_i$  is at least  $2D$ , if each request is replied directly. The preceding discussion has assumed that at most one process is aggressive for any interaction. If both processes are simultaneously aggressive for a given interaction, in the worst case, each *idle* process will still induce at least  $2D$  messages.

We consider a few common alternatives and indicate their impact on message complexity. An alternative to direct replies is that instead of replying negatively to  $p_j$ ,  $p_i$  may relay the original request to another neighbor of  $p_j$ . Although, this technique may reduce the total number of messages to  $D + 1$  (up to  $D$  relayed requests and 1 reply message), each message would have to be longer than in the case of direct replies and the total amount of “information bits” in this case is greater. A request typically needs at least  $\log N$  bits, where  $N$  is the total number of processes, for the receiver to identify which process is the sender. (Though a token in our algorithm carries some extra information, this information can be stored within each of the two processes which share the token.) A reply needs

only 1 bit. In the case of direct replies, the total number of bits that should be charged to  $p_i$  is  $D(\log N + 1)$ .

Using the technique of relaying requests, a request must carry information as for which interactions the requesting process is aggressive, because a process may be aggressive for any number of interactions in its interaction set. This information needs at least 1 bit and in the worst case  $D$  bits. So, the number of bits that a request carries will range from  $(\log N + 1)$  to  $(\log N + D)$ . In the worst case, a request will be relayed up to  $D$  times (including the original request sent by  $p_i$ ) implying that the total number of bits charged to  $p_i$  is larger than  $D(\log N + 1)$ .

An alternative is to allow a passive process to send a solicitation message. The primary difference between a solicitation and a request is that a process may send multiple solicitation messages (to different processes) simultaneously. Each solicitation may induce a request message. When more than one request arrives, a process must deny all but one of them. In the worst case, a process may simultaneously receive  $D$  requests and have to deny  $D - 1$  of them resulting in  $3D - 1$  messages (including the  $D$  solicitation messages), which is worse than the preceding bound. Furthermore, if we assume that a process may send at most one solicitation message at any time, the algorithm is essentially the same as one which allows both processes in an interaction to be simultaneously aggressive. Once again, this modification can not improve the message complexity of the worst case to be better than  $2D$ .

### 3.1.4.3 Shortening Delay Chains

In Algorithm A, we used edge-coloring to achieve the  $(D + 1)$  upper bound on the length of a delay chain. This appears to improve the performance of the algorithm only if the maximum degree is relatively small compared to the size of the interaction graph, for instance, if the interaction graph is regular (each node has same number of neighbors) and sparse.

However, even if the graph is dense or complete, in which case  $(D + 1) = n$ , the performance may be improved by using a specific edge-coloring. Notice that the delay chain considered in Theorem 6 is formed when the id of the incoming request at a process is greater than the id of its pending request. If the edge-coloring is computed to minimize the average length of paths with decreasing edge id's, the response time can be improved. The authors of [Gra73] consider a problem of ordering the edges of a graph such that the maximum length of an increasing (or decreasing) simple path is as small as possible. Their simple

construction provides a  $\frac{3}{4}n$  upper bound. A better bound asymptotically near  $\frac{1}{2}n$  is proved in [Cal84]. Their constructions can be used to improve the response time of Algorithm A in the case of dense interaction graphs.

#### 3.1.4.4 Average Case

The bounds in Theorem 6 demonstrate that the worst-case message cost of Algorithm A is proportional to the maximum degree of the interaction graph while its response time is quadratic in the maximum degree. We argue that the average-case performance is considerably superior.

Assuming an approximately even distribution of tokens, on the average, each process holds  $\frac{D}{2}$  tokens. Furthermore, a process delays an incoming request *only if* it has a pending request with a *smaller* id. Hence, unlike the assumption in Theorem 6, the length of the delay chain introduced by a request will usually be much smaller than its id. According to the algorithm, the replies propagating along a delay chain will alternately be *yes* and *no* (if a process that has delayed a request receives a *no* in response to its pending request, it commits to the delayed request); this implies that it is impossible for every process along a delay chain to experience the worst case. From these observations, we conclude that the worst case is very unlikely to occur and we expect that the *average* message cost and response time of the algorithm will be only a small factor of their worst-case bounds. Simulation experiments have supported this hypothesis — we present the results from the study of simple interaction graphs. Although the results of the experiment are hardly conclusive, they corroborate the preceding analysis.

We consider interaction graphs with 10 nodes and five different topologies: ring, 4-regular, 6-regular, 8-regular, and complete. The maximum degrees of the preceding graphs are 2, 4, 6, 8, and 9, respectively. A ring is defined by connecting each node to exactly two other nodes in the graph. The 4-regular, 6-regular, and 8-regular graphs are constructed from a ring by connecting every pair of nodes which are within 2, 3, and 4 hops away, respectively. A process is assumed to be connected to each of its communicating partners by a direct link; the message transmission time for each link is assumed to be exactly 1 unit of time. We model the transition of a process from active to idle by a Bernoulli trial with expected time  $a_t$  units. We found that the outcomes seem not very sensitive to the change of  $a_t$ ; however, we did not investigate their relationship further. The experimental results with  $a_t = 1$  are plotted in Figure 3.3. The worst-case message cost and response time are as defined in Section 1.2.5.

As seen from the figures, the experimental average-case metrics are consider-

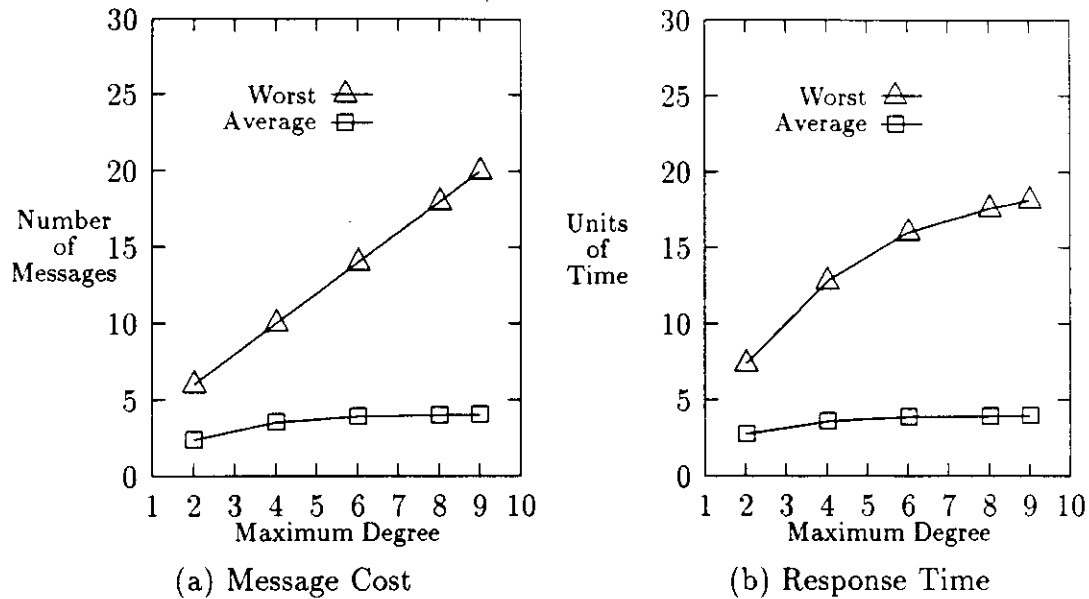


Figure 3.3: Experimental performance

ably better than the experimental worst-case metrics. In fact, the experimental worst-case response time, which as seen in Figure 3.3 (b) is almost linear, is considerably better than the theoretical quadratic bound derived in Theorem 6. However, the experimental worst-case count of the number of messages sent by a waiting process was found to be close to the theoretical bound.

## 3.2 Algorithm B

### 3.2.1 Process Interactions and Dining Philosophers

The dining philosophers problem [Dij78] is typically stated as follows [Cha84]: A philosopher is placed at each node of a finite undirected graph, called the conflict graph; each edge of the graph is associated with a unique fork. Two philosophers are neighbors if an edge exists between them. A philosopher is in one of three states: *thinking*, *hungry*, or *eating*. A thinking philosopher may autonomously become hungry. A hungry philosopher can eat only when it holds all forks associated with the edges incident to it (which implies neighboring philosophers do not eat simultaneously). A solution to this problem, henceforth referred to as the *diners problem*, is required to ensure that a hungry philosopher will eventually



*eat if every eating philosopher becomes thinking within a finite time.*

A solution to the diners problem can be embedded in an algorithm for the process interaction problem [Cha88, Bag89a, Cho92]. For binary interactions, this embedding yields an algorithm that is less efficient than Algorithm A. Furthermore, the embedding does not automatically satisfy fairness and modifications to achieve SPF are not straightforward. Existing algorithms for the interaction problem that utilize solutions to the diners problem usually adopt the following transformation: The members of an interaction cooperatively play the role of a philosopher such that each philosopher manages exactly one interaction. Thus two philosophers are neighbors and share a fork if and only if the corresponding interactions conflict (i.e. they have a common member). In order to use a solution to the diners problem, transitions from thinking to hungry and from eating to thinking for a process must be defined such that the eating period of any process is finite. A thinking philosopher becomes hungry when all members of the corresponding interaction are *waiting* (in the same sense as a process is *waiting* in Algorithm A, i.e. it is idle, it has not committed to any request, and it is not participating in any interaction). The embedded diners algorithm guarantees that a hungry philosopher will eventually eat. When a philosopher starts eating, if all members of the corresponding interaction are still waiting, the interaction may be started and the philosopher goes back to thinking; otherwise, the philosopher goes to thinking state without starting the interaction. This ensures that the eating period of a philosopher is finite.

We use a simple counterexample to show that the preceding transformation does not satisfy SPF for the case of binary interactions. Consider a system with three processes  $p_i$ ,  $p_j$ , and  $p_k$  and two interactions  $\{i,j\}$  and  $\{j,k\}$ , where philosophers X and Y correspond to the respective interactions; the configuration of the system is shown in Figure 3.4.

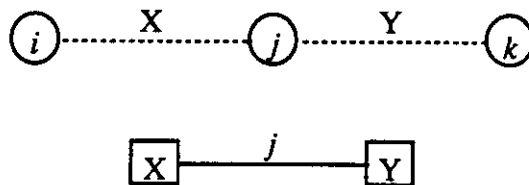


Figure 3.4: The interaction graph and the conflict graph of a problem instance

At some point in the computation, all three processes become waiting and both philosophers are hungry. Without loss of generality, assume that X becomes eating and consequently starts interaction  $\{i,j\}$ . However, when Y becomes eating,  $p_j$  is no longer waiting and Y goes back to thinking without starting interac-

tion  $\{j,k\}$ . Subsequently,  $p_i$  and  $p_j$  simultaneously become waiting and the cycle continues such that  $p_k$  is infinitely often ready to participate in interaction  $\{j,k\}$  but always remains waiting.

### 3.2.2 Alternative Transformation

We propose an alternative transformation that embeds a solution to the diners problem and satisfies SPF for the case of binary interactions. Two processes are neighbors if an interaction is defined between them. Similar to Algorithm A, neighboring processes share a single token which is initially assigned to one of the processes. Each process maintains a queue of the tokens that it holds. A process may request another process for interaction by sending the corresponding token as a request; an incoming token will be appended to the end of the token queue in the receiving process. *Let each process in the binary interaction problem also be a philosopher.* Thus a process, in addition to the active and idle states defined earlier, also has *dining* states that correspond to the thinking, hungry, and eating states of a philosopher.

The rules for sending and replying to requests are as follows: A process may send a request only if it is eating and tokens must be selected in the FIFO order defined by its token queue. On receiving a token, a process commits to the corresponding interaction if it is also *waiting* (i.e. it is idle, has not committed to a request, and is not participating in any interaction); otherwise, it denies the request. In either case, a process cannot delay an incoming request, but must reply to a request immediately on receiving it. When a process commits to a request, the corresponding interaction can be started.

Once again we must define the appropriate transitions for the dining states of a process and show that its eating period is finite. A process goes from thinking to hungry if it is waiting and its token queue is not empty. The diners algorithm guarantees that a hungry process will eventually eat. An eating process goes to thinking if its token queue becomes empty or it becomes not waiting (possibly because it has committed to some interaction while hungry). We prove subsequently that the eating period of a process is finite.

The alternative transformation is more formally described in the following state transition rules for the scheduler of each process  $p_i$ . We use a variable `dstate` to record the dining state of the process. Henceforth, we will abbreviate “`dstate = thinking`” as *thinking*; similarly for *hungry* and *eating*. Initially, all processes are thinking and `pend` is set to *null*.

Rule D1: *thinking and waiting and  $\neg$ empty(token.q).*

1.  $dstate := hungry$ .
- Rule D2: *eating and waiting and  $\neg pend$  and  $\neg empty(token\_q)$ .*
1. Remove the first token  $\{i,j\}$  from  $token\_q$  and send it to  $p_j$ .
  2.  $pend := j$ .
- Rule D3: *eating and  $\neg waiting$ . /\* Prior to becoming eating, hungry  $p_i$  has committed to some request. \*/*
1.  $dstate := thinking$ .
- Rule D4: *eating and  $\neg pend$  and  $empty(token\_q)$ .*
1.  $dstate := thinking$ .
- Rule D5: On receiving a token  $\{i,j\}$  from  $p_j$ .
1. *if waiting*  
     *then* send a *yes* message to  $p_j$ ;  $commit := j$ .  
     *else* send a *no* message to  $p_j$ .
  2. Append token  $\{i,j\}$  to the end of  $token\_q$ .
- Rule D6: On receiving a *yes* message.
1.  $flag := \{i,pend\}$ . /\* interaction  $\{i,pend\}$  is started \*/
  2.  $pend := null$ .
  3.  $dstate := thinking$ .
- Rule D7: On receiving a *no* message.
1.  $pend := null$ .

Note that Rules D3, D4, and D6(3) can be combined and further simplified. However, the present format is adopted for simplicity in the exposition. Again, we have omitted the rules that deal with termination of interactions as in Algorithm A. We refer to the derived algorithm as Algorithm B.

### 3.2.2.1 Correctness

It is straightforward to show that Algorithm B will satisfy the synchronization and mutual exclusion requirements and the proofs are omitted for brevity. The fairness of Algorithm B can be proved in an analogous way as for Algorithm A. In fact, the proof is much simpler.

**Lemma 7** *The eating period of a process is finite.*

*Proof.* An eating process becomes thinking on the execution of Rule D3, D4, or D6. We show that one of the preceding rules must eventually be executed for an eating process. An eating  $p_i$  is either waiting or not waiting; in the latter case Rule D3 is enabled and the result follows. Assume  $p_i$  is waiting.

If  $\text{token}_q$  is empty when  $p_i$  becomes eating, then Rule D4 is the only enabled rule of  $p_i$  and must eventually be executed, causing  $p_i$  to become thinking. Assume that  $\text{token}_q$  is not empty.  $p_i$  will send the first token from  $\text{token}_q$  to some neighbor, say  $p_j$  (Rule D2). On receiving the token,  $p_j$  must immediately deny or commit to the request (Rule D5). We assume that any request from  $p_i$  is denied; otherwise, Rule D6 will be executed and  $p_i$  becomes thinking as required. After receiving the *no* message from  $p_j$  and resetting  $\text{pend}$  to *null* (Rule D7),  $p_i$  will send the next token if  $\text{token}_q$  is not empty (Rule D2). As the embedded diners algorithm ensures that neighbors do not eat simultaneously and only eating processes can send tokens (Rule D2),  $p_i$  will not receive additional tokens as long as it is eating. It follows that the number of tokens in  $\text{token}_q$  will decrease monotonically and eventually become zero, at which point Rule D4 is enabled and must eventually be executed, causing  $p_i$  to become thinking. *End of Proof.*

**Lemma 8** *If a process is continuously waiting, eventually its  $\text{token}_q$  must become empty and thereafter remain empty.*

*Proof.* Suppose  $p_i$  becomes waiting at some state  $s$  and remains waiting at any state subsequent to  $s$ . If  $\text{token}_q$  is empty at  $s$ , then  $p_i$  can never subsequently receive a token. Because if it does, it must commit to the request (Rule D5) and violate the assumption that it is continuously waiting.

Assume  $\text{token}_q$  is not empty at  $s$ . If  $p_i$  is thinking, it will eventually become hungry (Rule D1) and subsequently eat (the embedded diners algorithm). An eating process eventually becomes thinking (Lemma 7) either when it becomes not waiting (Rules D3 and D6) or its  $\text{token}_q$  becomes empty (Rule D4). The first alternative violates the assumption of the lemma. It follows that  $p_i$  eventually exhausts its tokens. Once again, to ensure that  $p_i$  remains continuously waiting, it may never subsequently receive a token (Rule D5). *End of Proof.*

**Theorem 7** *Algorithm B satisfies SPF.*

*Proof.* As in Theorem 5, we need to show that if any of the neighbors of  $p_i$ , say  $p_j$ , becomes *idle* infinitely often, then  $p_j$  will eventually send token  $\{i,j\}$  to  $p_i$ . Again,  $p_j$  must participate in some interaction infinitely often; otherwise, from

Lemma 8,  $\text{token\_q}_j$  also will eventually remain empty, which is not possible. Each time  $p_j$  participates in some interaction, say  $\{j,k\}$ , there is at least one exchange of the corresponding token between  $p_j$  and  $p_k$  (Rule D5 and D6). As the token queues are FIFO, the token for interaction  $\{j,k\}$  will eventually reside in  $\text{token\_q}_j$  and its position is behind that of the token for interaction  $\{i,j\}$ . Before  $p_j$  again participates in interaction  $\{j,k\}$ ,  $p_j$  must have sent the token for interaction  $\{i,j\}$  to  $p_i$ , which will commit to the request if still waiting (Rule D5), contradicting the assumption. *End of Proof.*

### 3.2.2.2 Performance

We analyze the worst case performance of Algorithm B in terms of the complexity of the embedded diners algorithm. For the diners problem, the message cost is defined as the worst-case count of number of messages sent and received by a hungry process and its response time is the elapsed time in the worst case from the time a process becomes hungry until it becomes eating.

**Theorem 8** *Given a diners algorithm with message cost  $O(M)$  and response time  $O(T \times \tau)$ , where  $\tau$  denotes the duration of an eating period, the message cost of Algorithm B is  $O(M)$  and its response time is  $O(TD)$ .*

*Proof.* A process may hold at most  $D$  tokens. From the proof of Lemma 7, a waiting process that is eating will send at most  $D$  tokens until it becomes thinking. If the process is still waiting when it becomes thinking, its token queue must be empty and it will commit to the first request (token) subsequently received. It has been shown that  $O(M) \geq O(D)$  [Cha86], so the total message cost remains  $O(M)$ .

To compute the response time, suppose interaction  $\{i,j\}$  is enabled in some state. Assume both  $p_i$  and  $p_j$  are waiting in this state; otherwise, at least one of them will participate in some interaction within constant units of time. As requests are sent by an eating process in a sequential manner (Rule D2) and an incoming request is never delayed by the requested process (Rule D5), the duration of an eating period will be  $O(D)$  and hence the actual response time of the diners algorithm will be  $O(TD)$ . From the proof of Lemma 8, within  $O(TD) + O(D)$  ( $= O(TD)$ ) units of time (i.e. the response time of the diners algorithm plus the duration of an eating period),  $p_i$  will become not waiting or exhaust its tokens and so will  $p_j$ . It is not possible for both  $\text{token\_q}_i$  and  $\text{token\_q}_j$  to become (and remain) empty, so at least one of them will participate in some

interaction within  $O(TD)$  units of time. The response time of Algorithm B hence is  $O(TD)$  as stated. *End of Proof.*

### 3.2.3 Further Improvement

In Algorithm B, tokens are stored in a FIFO queue and sent in a sequential order to achieve SPF. The same goal can be achieved in a more efficient, though quite complicated, way:

Each process maintains a *circular* array that stores the id's of all its neighbors; a moving pointer indicates which position is the current head of the circular array. When a process becomes eating, it first sends (if still necessary) the token that it shares with the neighbor at or nearest to the head of the array; recall that the process may hold only some of the tokens that it shares with its neighbors. If it receives a negative reply to the first request, the process will send all of the rest tokens simultaneously. It then *confirms* whichever positive reply received first and *cancels* all subsequent positive replies.

Before going to thinking, an eating process always advances its pointer to the next position with one exception: If the eating process becomes thinking without sending any token (because it has been requested by other process while being hungry and its positive reply was confirmed), then the pointer remains at its current position. This prevents a neighbor from being ignored repeatedly.

To ensure the nice property as stated in Lemma 8 (i.e. a continuously waiting process will eventually run out of tokens), a process that has been requested for an interaction *returns* the corresponding token as a positive reply. An eating process that has received at least one positive reply is guaranteed to be participating in some interaction eventually; hence, the process may keep those tokens that are returned as positive replies with no risk of sending them again.

Finally, a process that has sent a positive reply (by returning the corresponding token) to some eating process delays at most one subsequent request (while denying **all others**) that was sent as the very first request by some other eating process. **After** receiving a confirmation or cancellation for the positive reply, the process **takes** the obvious action. It is important to note that the delay takes only a constant time, as a confirmation or cancellation will be received in a constant time.

The resulting algorithm is referred to as Algorithm B<sup>+</sup>. Theorems 9 and 10 can be proved in a similar way as the corresponding theorems of Algorithm B. Note that, since a process sends all tokens except the first one simultaneously and

each request is responded within a constant time, the eating period of a process is constant. -

**Theorem 9** *Algorithm  $B^+$  satisfies SPF.*

**Theorem 10** *Given a diners algorithm with message cost  $O(M)$  and response time  $O(T \times \tau)$ , where  $\tau$  denotes the duration of an eating period, the message cost of Algorithm  $B^+$  is  $O(M)$  and its response time is  $O(T)$ .*

### 3.3 Generalized Algorithms

We have been considering the binary interaction problem, where at most one interaction is defined between a pair of processes. The problem can be generalized such that (a) more than one interactions may be defined between two processes and (b) when a process becomes idle, it may be willing to participate in one of the subset of interactions of which it is a member. Two processes in CSP, for example, can synchronize via one of several matched pairs of communication commands and these commands may be further constrained by some boolean expressions.

Our algorithms can easily be extended to handle these generalizations. In particular, Algorithm A can be extended as follows: We treat each interaction between two processes as a different interaction and associate a unique token with each of these interactions. The interaction graph becomes a multi-graph, where parallel edges are allowed. Fortunately, a proper edge-coloring of the graph still suffices to prevent the algorithm from deadlock. When selecting a token, a waiting process simply skip the tokens corresponding to those interactions in which it is not willing to participate. Furthermore, if it receives the request for an interaction in which it is not willing to participate, it denies the request.

## CHAPTER 4

### Fault-Tolerance

We extend Algorithm A to satisfy the modified problem requirements (Section 1.2.4) in the presence of detectable failures. In particular, the extension guarantees SPF if there are only a finite number of failures. The technique is applicable to Algorithm B (and the adopted dining philosophers algorithm). To cope with undetectable failures, Algorithm B adopts an existing dining philosophers algorithm to achieve fairness and constant failure locality. We also show that, at the cost of increasing response time, it is possible to further reduce the failure locality.

#### 4.1 Detectable Failures

We assume that a faulty process behaves in the detectable fail-stop model [Sch82] according to the following assumptions. **Failure Assumption:** When a process fails, its variables are reset to their initial values and a failure message is broadcast to each of its neighbors (this can easily be implemented using the probe facility defined in [Sch82], which is used by a process to detect the failure of another process). A failed process may never restart. If it does restart, the **Restart Assumption** states that a designated rule is executed, by which the process sets some of its local variables to appropriate values and it also sends an *awake* message to each of its neighbors (the designated rule corresponds to the restart protocol postulated in [Sch82]). On receiving an awake message, a process responds by sending an *ack* message to the restarted process.

Assume that messages, including failure and awake messages, are delivered in FIFO order; this assumption is subsequently relaxed. Note that although messages are assumed to be delivered correctly, a message (including a failure message) may still be lost. This occurs, for instance, when a process fails upon receiving the message. We show how Algorithm A described in Section 3.1 can be extended to satisfy the modified problem requirements. We first consider the case of single failures and subsequently extend the solution to cope with multiple failures.



### 4.1.1 Single Failures

We initially assume that at most one process fails at a time and the minimum time between successive failures is sufficiently large such that a restarted process receives an ack message for each of the awake messages it sent.

A new variable called `proper` — a boolean array which is initially *true* — is added to each process. If a process receives a failure message from  $p_j$ , it sets `proper[j]` to false; also, if a process fails and subsequently restarts, it initializes `proper` to false. Subsequently, `proper[j]` is reset to true either if the process receives an awake message from  $p_j$  or if it receives an ack message from  $p_j$  in response to its awake message. A process sends a request to some  $p_j$  only if `proper[j]` is true, which ensures that it will eventually receive either a reply or a failure message from  $p_j$ . Failures need special handling in the following three contexts:

- outdated requests: A request sent by  $p_i$  to  $p_j$  is said to be outdated if either  $p_i$  or  $p_j$  fails (and possibly restarts) after the request is sent and before a reply is received by  $p_i$ .
- outdated commitment:  $p_i$  or  $p_j$  fails while executing interaction  $\{i,j\}$ .
- token loss: a process that currently holds the token for an interaction fails.

We first consider outdated requests. Under the single failure assumption, either the requesting process, say  $p_i$ , or the requested process, say  $p_j$ , has failed. Assume  $p_j$  fails. Eventually,  $p_i$  will receive a failure message from  $p_j$ , which is simply interpreted as a *no* response to its request. If  $p_j$  restarts, it will send an awake to  $p_i$  and wait for the corresponding ack. The FIFO assumption guarantees that outdated messages from  $p_i$  (including outdated requests) must be received prior to the ack and can simply be discarded by  $p_j$ . On the other hand, if  $p_i$  fails,  $p_j$  will receive the request and then the failure message from  $p_i$ . As detailed in the extended algorithm,  $p_j$  takes appropriate corrective action on receiving the failure message: (a) if it had denied the request, no further action is necessary, (b) if it had delayed the request,  $p_j$  resets `delay` to *null*, and (c) if it had committed to the request,  $p_j$  resets `commit` to *null*. Also note that if  $p_i$  restarts, it will also discard **any** message it receives from  $p_j$  prior to receiving an ack in response to its awake message (during the period while `proper[j]` is false).

The problem of outdated commitments is handled similarly. If  $p_i$  is participating in interaction  $\{i,j\}$  and it receives a failure message from  $p_j$ , it simply assumes that the interaction has been completed.

Finally, to prevent token loss, the token assignment step is modified. Instead of initially assigning the token for interaction  $\{i,j\}$  to either  $p_i$  or  $p_j$ , two copies

of the token are created and each member is initialized with a copy. As a token can be lost only if its owner fails, it will be regenerated when the process restarts and is re-initialized. To preserve the performance advantage of having a single token for each interaction, we stipulate that (a) if a process acquires two tokens for the same interaction it simply discards one and (b) if  $p_i$  requests interaction  $\{i,j\}$  and receives a token for the same interaction from  $p_j$ , it discards the token if  $i < j$ . The strategy is illustrated in Figure 4.1.

The modified algorithm is shown below, where the modified parts are underlined for clarity. Assuming no failures, the performance of the modified algorithm is the same as that of Algorithm A derived in Theorem 6.

#### VARIABLES:

$flag_i$ : Interaction  $\{i,j\}$  is started if  $flag_i = \{i,j\}$  or  $flag_j = \{i,j\}$ .  
proper: a boolean array.  
 $token\_q$ : a queue of tokens. Each token is an unordered pair of process id's.  
 $ino[\{i,j\}]$ : id of interaction  $\{i,j\}$ .  
 $pend_i$ : If  $p_i$  has a pending request, say  $\{i,j\}$ ,  $pend_i$  is set to  $j$ .  
 $delay_i$ : id of the process, if any, whose request has been delayed by  $p_i$ .  
 $commit_i$ : ( $commit_i = j$ ) indicates that  $p_i$  has committed to a request from  $p_j$ .

#### INITIALIZATION:

1. Edge-color the interaction graph with positive integers  $\leq (D + 1)$ . Assign  $ino[\{i,j\}]$  the color (id) of edge  $\{i,j\}$ .
2. For each interaction, assign a copy of the associated token to each member.
3. All tokens initially assigned to a process are stored (in an arbitrary order) in its  $token\_q$ .
4. Variables  $flag$ ,  $pend$ ,  $delay$ , and  $commit$  of each process are initialized to null and proper to true.

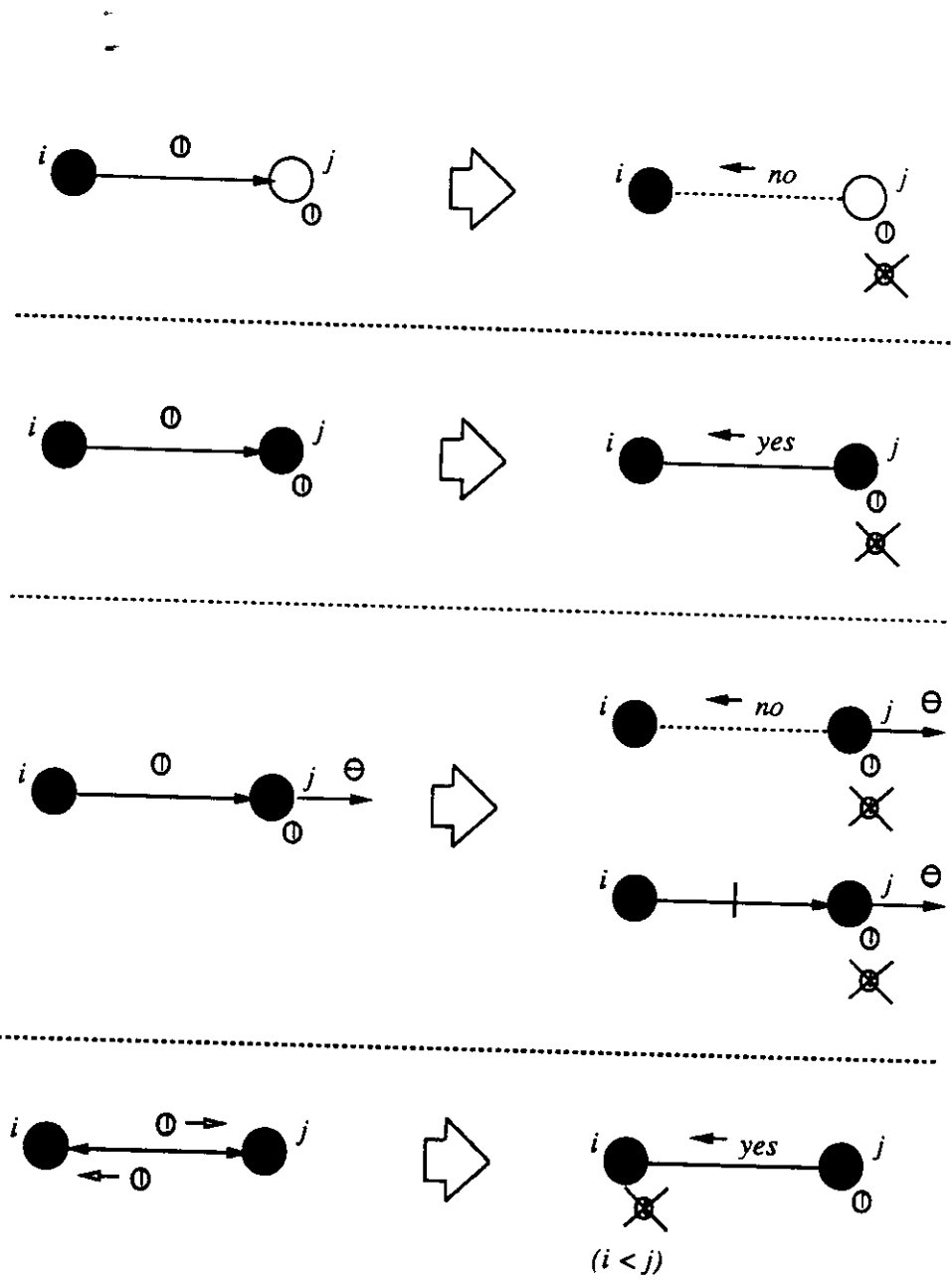
#### TRANSITION RULES:

(Again, *waiting* is implemented by "(state = *idle*) and  $\neg commit$  and  $\neg flag$ .")

Rule 1: *waiting* and  $\neg pend$  and  $\neg empty(token\_q)$ .

/\*  $token\_q$  is considered empty if for every token  $\{i,k\}$  in  $token\_q$ ,  $proper[k]$  is *false*. \*/

1. **token-selection:** For the first time after the process becomes waiting, the token at the head of  $token\_q$  is selected; subsequent tokens, if needed, are selected from  $token\_q$  in decreasing order of their id's. Skip any token  $\{i,k\}$ , where  $proper[k]$  is *false*.



In each case, an extra copy of token is deleted.

Figure 4.1: How an extra copy of token is removed

2. Remove the selected token  $\{i,j\}$  from  $\text{token\_q}$  and send it to  $p_j$ .
3.  $\text{pend} := j$ .

Rule 2: On receiving a request (token  $\{i,j\}$ ) from  $p_j$ .

if proper[j] and (pend  $\neq$  j) then

1. if waiting and  $\neg$ pend  
then send a *yes* message to  $p_j$ ;  $\text{commit} := j$ .
2. if waiting and pend  
then if  $\neg$ delay and ( $\text{ino}\{\{i,j\}\} \geq \text{ino}\{\{i,\text{pend}\}\}$ )  
then  $\text{delay} := j$   
else send a *no* message to  $p_j$ .
3. if  $\neg$ waiting then send a *no* message to  $p_j$ .
4. Append token  $\{i,j\}$  to the end of  $\text{token\_q}$ , if it is not an extra copy.

if proper[j] and (pend = j) then /\* Processes send tokens to each other.  
\*/

1. if (i > pend)  
then send a *yes* message to  $p_{\text{pend}}$ ;  
commit := pend;  
append the token to the end of token\_q;  
pend := null.  
else do nothing. /\* Discard token and wait for a *yes* message. \*/

if  $\neg$ proper[j] then

1. Append the token to the end of token\_q, if it is not an extra copy.

Rule 3: On receiving a *yes* message.

if proper[j] then

1.  $\text{flag} := \{i,\text{pend}\}$ . /\* Interaction  $\{i,\text{pend}\}$  is started. \*/
2. if delay then send a *no* message to  $p_{\text{delay}}$ .
3.  $\text{pend} := \text{null}$ ;  $\text{delay} := \text{null}$ .

if  $\neg$ proper[j] then

1. Do nothing. /\* Ignore the outdated reply. \*/

Rule 4: On receiving a *no* message.

if proper[j] then

1. if delay then send a *yes* message to  $p_{\text{delay}}$ ;  $\text{commit} := \text{delay}$ .
2.  $\text{pend} := \text{null}$ ;  $\text{delay} := \text{null}$ .

if  $\neg$ proper[j] then

1. Do nothing. /\* Ignore the outdated reply. \*/

Rule 5: On receiving a *failure* message from  $p_j$ .

1.  $\text{proper}[j] := \text{false}$ .
2. **if** ( $\text{pend} = j$ ) **then**  $\text{pend} := \text{null}$ . /\* Interpret the *failure* message as a *no* message. \*/
3. **if** ( $\text{delay} = j$ ) **then**  $\text{delay} := \text{null}$ .
4. **if** ( $\text{commit} = j$ ) **then**  $\text{commit} := \text{null}$ .
5. **if** ( $\text{flag} = \{i, j\}$ ) **then**  $\text{flag} := \text{null}$ .

Rule 6: On restart. /\* The designated rule for restart. \*/

1.  $\text{proper} := \text{false}$ .
2. Broadcast an *awake* message to each neighbor.

Rule 7: On receiving an *awake* message from  $p_j$ .

1.  $\text{proper}[j] := \text{true}$ .
2. Send an *ack* message to  $p_j$ .

Rule 8: On receiving an *ack* message from  $p_j$ .

1.  $\text{proper}[j] := \text{true}$ .

(Again, a host process terminates an interaction by resetting its **flag** to *null*; its scheduler detects that the **flag** has been reset and notifies the other scheduler by means of an appropriate message, which resets its **commit** to *null*.)

#### 4.1.2 Multiple Failures

Assume that a process may fail at any time and may fail repeatedly. Multiple failures imply that a failure message may itself be lost, making it harder to identify outdated messages.

Suppose  $p_i$  fails after receiving a failure message from  $p_j$ . On restart,  $p_i$  will not “remember” that  $p_j$  has failed. However, the *awake* and corresponding *ack* messages can be used to reconstruct a consistent view. Assume  $p_i$  restarts while  $p_j$  is still failed. The absence of an *ack* message from  $p_j$  will cause  $\text{proper}[j]$  to remain **false**. This will prevent  $p_i$  from sending a request to  $p_j$  or from responding to an **outdated** message from  $p_j$ . However, it is possible for a process to receive an **outdated ack** (for instance, if a failed process restarts, sends *awake* messages, fails before receiving all the corresponding *ack* messages, and again restarts and sends *awake* messages) without it being detectable as an outdated message. Every *ack* message received by a process is handled as explained in Rule 8. Processing an outdated *ack* creates only a temporary inconsistency because an outdated *ack* must be followed by a failure message, which will appropriately reset the state of the recipient process. Similarly, an outdated failure message cannot be

detected and is instead handled exactly as in Rule 5. Once again processing an outdated failure message causes only a temporary inconsistency, because a subsequent awake message will eventually be received.

The loss of an awake or ack message can be handled in a similar way. We thus conclude that multiple failures do not need additional modifications to be introduced in the algorithm.

### Remark

The extended algorithm does not use any notion of “incarnation number,” which is assumed in [Sch82]. If messages may be delivered out of order, the incarnation numbers are sufficient to recover the order of messages to serve our purpose. For example, when  $p_i$  sends a message to  $p_j$ , it appends its own incarnation number together with, what is in its knowledge, the highest incarnation number for  $p_j$ . Using the pair of incarnation numbers in each incoming message, a process may easily identify an outdated message and update incarnation numbers for its neighbors.

## 4.2 Undetectable Failures

In the undetectable fail-stop model [Fis85], a failed process does not execute additional computation steps and its local variables maintain their last assigned values.

### 4.2.1 Algorithms with Small Failure Locality

Although Algorithm A satisfies fairness and is very efficient, its failure locality is not satisfactory. As the longest delay chain that can be formed in the algorithm has a length of  $D + 1$ , it follows that Algorithm A has a failure locality of  $D + 1$ .

In [Cho92], Choy and Singh describe an algorithm for the diners problem that has a constant failure locality of 4. The message cost and response time of their algorithm are  $O(D^2)$  and  $O(D^2 \times \tau)$  respectively, where  $\tau$  is the maximum eating time of a process.

Adopting the diners algorithm in [Cho92], Algorithm B satisfies fairness and achieves a failure locality of 5. The increment of 1 in failure locality is introduced by the fact that an eating process has to wait for the replies to its requests from its neighbors. From Theorem 8, the message cost of Algorithm B is  $O(D^2)$  and its response time will be  $O(D^3)$ .

Analogously, Algorithm  $B^+$  satisfies fairness and achieves a failure locality of 6. The failure locality increases by 2, because an eating process may have to wait for the reply to its first request from the requested neighbor that happens to be waiting for a confirmation from another eating process. From Theorem 10, the message cost of Algorithm  $B^+$  is  $O(D^2)$  and its response time will also be  $O(D^2)$ .

#### 4.2.2 Minimizing Failure Locality

In this section, we propose a dining philosophers algorithm that achieves the smallest possible failure locality of 2; this algorithm has also been presented in [Tsa93a]. The algorithm can be adopted by Algorithm B to achieve fairness and a failure locality of 3. It should be noted that a reduction of 1 on the failure locality may translate into a reduction of a factor of  $D$  on the number of processes that may be affected by a failed process.

Our algorithm combines the idea of a dynamic priority scheme as in [Cha84] with the use of a preemptable fork collecting strategy as in [Cho92]. Surprisingly, if no failures actually occur, the response time of our algorithm remains asymptotically as good as that of [Cha84], which is  $O(n)$  with  $n$  being the total number of processes; in the presence of failures, the response time degrades to  $O(n^2)$ . The response time compares favorably with those of existing algorithms with a worse failure locality of three.

Besides its application in solving the process interaction problem with undetectable process failures, the proposed algorithm is interesting in its own right. In Table 4.1, the algorithm is compared with existing algorithms and the lower bounds for the dining philosophers problem. In the table,  $c$  is the number of colors required to color the resources such that the resources needed by a process have different colors,  $\delta$  is the maximum degree of the conflict graph,  $n$  is the total number of processes, and  $U$  is the minimum number of process id's such that two neighboring processes have different id's. No specific time bound for the first algorithm of Styer and Peterson is available. A subscript  $f$  indicates that the time bound is obtained with the effect of failures taken into consideration.

##### 4.2.2.1 Informal Description

The ideas of our diners algorithm are as follows:

- **Dynamic Priority Scheme:** A directed acyclic graph that results from orienting the edges of the conflict graph is maintained cooperatively by the processes. The process at the start-node of a directed edge has lower

Algorithms	Response Time	Failure Locality
Lynch 80 [Lyn80]	$O(c^\delta)$	$O(\delta)$
Chandy & Misra 84 [Cha84]	$O(n)$	$O(n)$
Styer & Peterson 88 [Sty88]	Exponential	3
Styer & Peterson 88 [Sty88]	$O(\delta^{\log \delta + 1})$	$O(\log \delta)$
Awerbuch & Saks 90 [Awe90]	$O(\delta^2 \log U)$	$O(\delta)$
Choy & Singh 92 [Cho92]	$O(\delta^{\delta+2})_f$	3
Choy & Singh 92 [Cho92]	$O(\delta^2)_f$	4
Proposed algorithm	$O(n), O(n^2)_f$	2
Lower bounds	$O(\delta)$	2

Table 4.1: Comparison of algorithms and the lower bounds for the dining philosophers problem

priority than the process at the end-node. Initially, an edge between two neighbors is directed from the process with smaller id to the one with larger id; as every process has an unique id, this orientation guarantees that the directed graph is acyclic initially. Each time when a process becomes eating, it reverses all its incoming edges such that it has lower priority than each of its neighbors; the redirection results in another acyclic graph.

- **Preemptable Fork Collecting Strategy:** A hungry process tries to first collect all the forks shared with the neighbors that have higher priority; these forks and neighbors are referred to as *higher* forks and neighbors, respectively. After it has successfully done so, the process starts to collect the *lower* forks that it shares with lower neighbors. A hungry process delays a request from a lower neighbor only if it holds all the higher forks; otherwise, it grants the request immediately (the corresponding lower fork is “preempted”). It grants any request from a higher neighbor and consequently grants all the requests from lower neighbors that have been delayed (**the request from the higher neighbor causes the preemption of those lower forks**).

The dynamic priority scheme ensures that, if a process remains hungry “long enough” while some of its higher neighbors repeatedly causing preemption, then it will eventually has higher priority than those neighbors such that no further preemption is possible. The fork collecting strategy guarantees that the length of any “chain of delayed requests” is at most two. This is because a hungry



process *delays* a request only if it is waiting for the replies to its requests to some lower *neighbors*, each of which may in turn delay the request only if it is eating. It is interesting to note that, with the fork collecting strategy, the necessity of acyclicity of the directed graph maintained by the algorithm is to prevent livelock rather than deadlock as in [Cha84] among processes on a cycle.

#### 4.2.2.2 The Algorithm

To maintain the directed graph, each process  $p_i$  has a boolean array  $\text{higher}_i$  with the id's of its neighbors as index. A *true* value of  $\text{higher}_i[j]$  indicates that the neighbor  $p_j$  of  $p_i$  has higher priority than  $p_i$ ;  $\text{higher}_i[j]$  is set to *false* by  $p_i$  upon receiving a *switch* message from  $p_j$ , which indicates that  $p_j$  has started to eat. It is possible that  $p_i$  “thinks”  $p_j$  has higher priority while a *switch* message is in transit from  $p_j$  to  $p_i$ ; this “conservative thought” does not affect the correctness of the algorithm.

We postulate that each process has a variable *dstate* indicating its dining state. We will abbreviate “*dstate* = *thinking*” as *thinking*; similarly for *hungry* and *eating*. Initially, all processes are assumed to be thinking. The code for a process  $p_i$  is illustrated below.

##### VARIABLES:

(Every array is indexed by the id's of the neighbors of  $p_i$ . The subscript  $i$  of a variable is omitted if no confusion may rise.)

*dstate*: The dining state of  $p_i$ : *thinking*, *hungry*, or *eating*.

*higher*: Boolean array. A neighbor  $p_j$  has higher priority if  $\text{higher}[j]$  is *true*.

*fork*: Boolean array.  $p_i$  holds  $\text{fork}_{i,j}$  if  $\text{fork}[j]$  is *true*.

*token*: Boolean array.  $p_i$  holds the token for requesting  $\text{fork}_{i,j}$  if  $\text{token}[j]$  is *true*.

*collecting*: Boolean.  $p_i$  is collecting forks if *collecting* is *true*.

*delay*: Boolean array. A request from  $p_j$  has been delayed if  $\text{delay}[j]$  is *true*.

##### INITIALIZATION:

1. For each neighbor  $p_j$ , where  $j > i$ ,  $\text{higher}_i[j]$  is initialized to *true*,  $\text{fork}_i[j]$  to *false*, and  $\text{token}_i[j]$  to *true*.
2. For each neighbor  $p_j$ , where  $j < i$ ,  $\text{higher}_i[j]$  is initialized to *false*,  $\text{fork}_i[j]$  to *true*, and  $\text{token}_i[j]$  to *false*.
3. Variables *collecting* and *delay* are initialized to *false* (and *dstate* to *thinking*).

##### ACTIONS:

(For the ease of presentation, we assume that each time when a process goes

from *thinking* to *hungry*, some “internal message” is generated so that it can be received by the algorithm to observe the transition; analogously for the transition from *eating* to *hungry*. Each time after a process becomes eating, Action D2 will always be enabled and executed before Action D1, due to FIFO message delivery.)

D1: On observing the transition to *hungry*.

1. **if**  $\forall k(\text{fork}[k])$   
**then**  $\text{dstate} := \text{eating}$ ; /\* Delayed requests have been handled by D2.  
 \*/  
     **for each**  $k$  such that  $\neg \text{higher}[k]$  **do**:  
         send a *switch* to  $p_k$ ;  $\text{higher}[k] := \text{true}$ .
2. **if**  $\neg \forall k(\text{fork}[k])$  **and**  $\forall k(\text{higher}[k] \Rightarrow \text{fork}[k])$   
**then**  $\text{collecting} := \text{true}$ ;  
     **for each**  $k$  such that  $\neg \text{higher}[k]$  **and**  $\neg \text{fork}[k]$  **and**  $\text{token}[k]$  **do**:  
         send a *request* to  $p_k$ ;  $\text{token}[k] := \text{false}$ .
3. **if**  $\neg \forall k(\text{fork}[k])$  **and**  $\neg \forall k(\text{higher}[k] \Rightarrow \text{fork}[k])$   
**then**  $\text{collecting} := \text{true}$ ;  
     **for each**  $k$  such that  $\text{higher}[k]$  **and**  $\neg \text{fork}[k]$  **and**  $\text{token}[k]$  **do**:  
         send a *request* to  $p_k$ ;  $\text{token}[k] := \text{false}$ .

D2: On observing the transition to *thinking*.

**For each**  $k$  such that  $\text{delay}[k]$  **do**:

1. send  $\text{fork}_{i,k}$  to  $p_k$ ;  $\text{fork}[k] := \text{false}$ .
2.  $\text{delay}[k] := \text{false}$ .

D3: On receiving a *request* from  $p_j$ .

1.  $\text{token}[j] := \text{true}$ ;
2. **if**  $\neg \text{eating}$  **and**  $\neg \text{collecting}$   
**then** send  $\text{fork}_{i,j}$  to  $p_j$ ;  $\text{fork}[j] := \text{false}$ .
3. **if**  $\text{eating}$  **or** ( $\text{collecting}$  **and**  $\neg \text{higher}[j]$  **and**  $\forall k(\text{higher}[k] \Rightarrow \text{fork}[k])$ )  
**then**  $\text{delay}[j] := \text{true}$ .
4. **if**  $\text{collecting}$  **and**  $\neg \text{higher}[j]$  **and**  $\neg \forall k(\text{higher}[k] \Rightarrow \text{fork}[k])$   
**then** send  $\text{fork}_{i,j}$  to  $p_j$ ;  $\text{fork}[j] := \text{false}$ .
5. **if**  $\text{collecting}$  **and**  $\text{higher}[j]$   
**then** send  $\text{fork}_{i,j}$  to  $p_j$ ;  $\text{fork}[j] := \text{false}$ ;  
     send a *request* to  $p_j$ ;  $\text{token}[j] := \text{false}$ ;  
     **for each**  $k$  such that  $\text{delay}[k]$  **do**:  
         send  $\text{fork}_{i,k}$  to  $p_k$ ;  $\text{fork}[k] := \text{false}$ .

D4: On receiving  $\text{fork}_{i,j}$ .

1.  $\text{fork}[j] := \text{true}$ ;

2. **if**  $\forall k(\text{fork}[k])$   
**then**  $\text{dstate} := \text{eating}$ ;  $\text{collecting} := \text{false}$ ;  
for each  $k$  such that  $\neg \text{higher}[k]$  do:  
send a *switch* to  $p_k$ ;  $\text{higher}[k] := \text{true}$ .
3. **if**  $\neg \forall k(\text{fork}[k])$  and  $\forall k(\text{higher}[k] \Rightarrow \text{fork}[k])$   
**then** for each  $k$  such that  $\neg \text{higher}[k]$  and  $\neg \text{fork}[k]$  and  $\text{token}[k]$  do:  
send a *request* to  $p_k$ ;  $\text{token}[k] := \text{false}$ .

D5: On receiving a *switch* from  $p_j$ .

$\text{higher}[j] := \text{false}$ .

#### 4.2.2.3 Correctness Proof

It is straightforward to show that the algorithm satisfies the safety requirement. To prove the liveness property, we show that the algorithm has a response time of  $O(n)$ , if no failures actually occur, and in the presence of failures, a hungry process will eat within  $O(n^2)$  units of time if no processes within two hops away from the process fail. We use  $\nu$  to denote the bound on message transmission time.

At any state  $s$  of an execution, the algorithm maintains a directed graph  $G(s)$  with the conflict graph as the underlying undirected graph, where the direction of an edge between two neighboring processes is determined as follows:

The edge between  $p_i$  and  $p_j$  is directed from  $p_i$  to  $p_j$  if and only if (a)  $\text{higher}_j[i]$  is *false* or (b) both  $\text{higher}_i[j]$  and  $\text{higher}_j[i]$  are *true* and there is a *switch* message in transit from  $p_i$  to  $p_j$  (upon receiving the *switch* message,  $p_j$  will set  $\text{higher}_j[i]$  to *false*).

It can be shown that for any pair of neighbors  $p_i$  and  $p_j$ , at most one of  $\text{higher}_i[j]$  and  $\text{higher}_j[i]$  is *false*. Moreover, both  $\text{higher}_i[j]$  and  $\text{higher}_j[i]$  are *true* if and only if a *switch* message is in transit between  $p_i$  and  $p_j$ . Therefore, the direction of an edge between any two neighbors is well defined. From the initialization and Actions D1 (Step 1) and D4 (Step 2) of the algorithm, it is clear that  $G(s)$  is *acyclic* at any state  $s$  in an execution of the system.

Let  $H_i(s)$  denote the set of processes that are reachable from  $p_i$  with respect to  $G(s)$ ; a node is *reachable* from another in a directed graph if (a) there exists a directed path from the latter to the former or (b) the two nodes are the same node.  $H_i(s)$  contains  $p_i$ , the higher neighbors of  $p_i$ , the higher neighbors of the higher neighbors of  $p_i$ , and so on. Let  $|H_i(s)|$  denote the number of processes in  $H_i(s)$ ; analogously for other sets of processes.

**Lemma 9** *If  $p_i$  does not become eating at any state between state  $s'$  and some future state  $s''$  inclusively, then  $H_i(s'') \subseteq H_i(s')$  and hence  $|H_i(s'')| \leq |H_i(s')|$ ; additionally, if some other process in  $H_i(s')$  meanwhile becomes eating, then  $|H_i(s'')| < |H_i(s')|$ .*

*Proof.* When a process  $p_j$ ,  $j \neq i$ , becomes eating and reverses all its incoming edges, no processes are made reachable from  $p_i$  due to the redirection of those edges incident to  $p_j$ . It follows that  $H_i(s'') \subseteq H_i(s')$ . If  $p_j$  happens to be in  $H_i(s')$ , then it is no longer reachable from  $p_i$  and hence  $|H_i(s'')| < |H_i(s')|$ . *End of Proof.*

For a process  $p_i$  that is *hungry* at some state  $s$ , we define  $F_i(s)$ , a subset of  $H_i(s)$ , recursively as follows:

1.  $p_i$  itself is in  $F_i(s)$ .
2. If  $p_j$  is in  $F_i(s)$ ,  $p_k$  is hungry, and  $p_k$  is a higher neighbor of  $p_j$ , then  $p_k$  is also in  $F_i(s)$ .

We define the *level* of a process in  $F_i(s)$  to be the smallest positive integer  $k$  such that each higher neighbor of the process is either not in  $F_i(s)$  or at a level less than  $k$ . Particularly, a process none of whose higher neighbors is in  $F_i(s)$  is at level 1. It is clear that  $p_i$  is at a level less than or equal to  $|F_i(s)|$ .

#### 4.2.2.4 Without Failures

**Lemma 10** *If a hungry process holds all higher forks, then within  $(2\nu + \tau)$  units of time the process starts to eat unless it receives a request from some higher neighbor.*

*Proof.* Since the hungry process holds all higher forks, it will send a request for each lower fork that it does not hold. It takes  $\nu$  units of time for the request to be delivered, possibly  $\tau$  units for the lower neighbor to finish eating, and  $\nu$  units for the fork to be delivered. Unless it receives a request from some higher neighbor and has to give up the corresponding higher fork, within  $(2\nu + \tau)$  units of time the process will hold all needed forks and start to eat. *End of Proof.*

**Lemma 11** *If  $p_i$  is hungry at state  $s'$ , then within  $(4\nu + \tau)$  units of time the system is at another state  $s''$  such that one of the following three cases occurs:*

1.  $p_i$  starts to eat.

2. *Some process other than  $p_i$  in  $F_i(s')$  starts to eat, implying that  $|H_i(s'')| < |H_i(s')|$ .*
3. *A higher neighbor of some process in  $F_i(s')$  becomes hungry such that  $|H_i(s'')| \leq |H_i(s')|$  and  $|F_i(s'')| > |F_i(s')|$ .*

*Proof.* Consider a process in  $F_i(s')$  at level 1, i.e. the process none of whose higher neighbors is in  $F_i(s')$ ;  $p_i$  may happen to be this process. Unless some of its higher neighbors becomes hungry, it takes at most  $2\nu$  units of time (which is the transmission time of a request plus that of a fork) for the process to collect all the higher forks and from Lemma 10, within another  $(2\nu + \tau)$  units of time the process will start to eat. Note that the considered process is also in  $H_i(s')$ , which contains  $F_i(s')$ . If the process is not  $p_i$ , then from Lemma 9 it follows that  $|H_i(s'')| < |H_i(s')|$ .

If some of the higher neighbors does become hungry within  $(4\nu + \tau)$  units of time but no processes in  $F_i(s)$  eat, then it is clear that  $|H_i(s'')| \leq |H_i(s')|$ , from Lemma 9, and  $|F_i(s'')| > |F_i(s')|$ , from the definition of  $F_i$ . *End of Proof.*

**Theorem 11** *If no failures occur, the algorithm satisfies the problem requirements and has a response time of  $O(n)$ , or more precisely  $O(n(\nu + \tau))$ .*

*Proof.* Suppose  $p_i$  becomes hungry at some state  $s$ . From repeated applications of Lemma 11, it follows that  $p_i$  will start to eat within  $(M + N + 1)(4\nu + \tau)$  units of time, where  $M$  and  $N$  are the numbers of processes in  $H_i(s)$  that become hungry and eating respectively after state  $s$  and before  $p_i$  eats. As  $|H_i(s)|$  is bounded by  $n$ , the algorithm has a response time of  $O(n(\nu + \tau))$ , or  $O(n)$ . *End of Proof.*

#### 4.2.2.5 With Failures

As will greatly simplify the analysis, we first examine how a failed process may affect each of its *lower* neighbors (as defined at the state when the failure occurs). There are two possibilities:

1. *The lower neighbor will hold and keep the shared fork forever within  $\nu$  units of time after the failure.* This occurs if (a) the neighbor currently holds the fork and there is no request from the failed process that is in transit to or delayed by the neighbor or (b) the fork is in transit to the neighbor and there is no request from the failed process following the fork.

2. *The lower neighbor will lose the fork forever within  $(\nu + \tau)$  units of time.* This occurs if (a) the failed process holds the fork upon failing, (b) the fork is in transit to the failed process, or (c) the neighbor subsequently relinquishes the fork (possibly after it eats for  $\tau$  units of time) in response to a request made by the failed process prior to the failure. According to Action D3 of the algorithm, the neighbor will thereafter grant any request unless itself also fails.

In the first case, the edge (and the corresponding fork) between the lower neighbor and the failed process can be regarded as non-existing; while in the second, since the neighbor will grant any request unless itself also fails, the neighbor and all the edges incident to it (as well as the corresponding forks) can be regarded as non-existing. We hence make the following claim:

Suppose a process  $p_i$  becomes hungry at state  $s$  and no processes within two hops away from  $p_i$  will fail. In no risk of underestimating the asymptotic response time for  $p_i$ , we may assume that failures occur only on the processes outside of  $H_i(s)$ .

Regarding the effect of a failed process on its higher neighbors, we observe that, if a process, particularly one that is in  $H_i(s)$ , loses a lower fork forever due to the failure of some lower neighbor, it will behave like a normal process except that it will never eat. In particular, when holding all higher forks, the process will attempt to collect all lower forks and in the meantime delay any request from a lower neighbor.

**Lemma 12** *If a hungry process holds all the higher forks and no processes within one hop away will fail, then within  $(2\nu + \tau)$  units of time the process starts to eat unless it receives a request from a higher neighbor.*

*Proof.* The lower neighbors of the hungry process behave as in Lemma 10, as they are one hop away and assumed not to fail. *End of Proof.*

**Lemma 13** *If  $p_i$  is hungry at state  $s'$  and no processes within two hops away from  $p_i$  will fail, then within  $((|F_i(s')| + 1)\nu + 2\nu + \tau)$  units of time the system is at another state  $s''$  such that one of the following three cases occurs:*

1.  $p_i$  starts to eat.

2. Some process other than  $p_i$  in  $F_i(s')$  starts to eat, implying that  $|H_i(s'')| < |H_i(s')|$ .
3. A higher neighbor of some process in  $F_i(s')$  becomes hungry such that that  $|H_i(s'')| \leq |H_i(s')|$  and  $|F_i(s'')| > |F_i(s')|$ .

*Proof.* Assume no higher neighbors of the processes in  $F_i(s')$  become hungry within  $((|F_i(s')| + 1)\nu + 2\nu + \tau)$  units of time from state  $s'$ . Recall that failures can be assumed to occur only on processes outside of  $H_i(s')$ , each of which may be lower neighbors of some processes in  $H_i(s')$  and/or in  $F_i(s')$ .

Within  $(1 + 1)\nu$  units of time each process in  $F_i(s')$  at level 1 should hold all its higher forks; Lemma 12 is applicable unless each process has a failed lower neighbor outside of  $H_i(s')$ . In any case, each process at level 1 will acquire all the lower forks from the processes at level 2 unless some process at level 2 has become eating, implying Case 1 or 2 has occurred. Within  $(1 + 2)\nu$  units of time each process at level 2 should give up its lower forks to the processes at level 3 (unless it has become eating). Within  $(1 + 3)\nu$  units of time each process at level 3 should hold all its higher forks. Reasoning along this line, within  $(1 + |F_i(s')|)\nu$  units of time Lemma 12 will be applicable to either  $p_i$  which is at a level  $\leq |F_i(s')|$  or some of its higher neighbors at the immediate preceding level; note that *no* processes within *one* hop away from either process may fail according to the hypothesis of the lemma. Within  $((|F_i(s')| + 1)\nu + 2\nu + \tau)$  units of time from state  $s'$ , one of the two processes starts to eat or some other process in  $F_i(s')$  has become eating. The rest is analogous to the proof of Lemma 11. *End of Proof.*

**Theorem 12** *The algorithm satisfies the problem requirements with a failure locality of 2 and has a response time  $O(n^2)$ , or more precisely  $O(n^2\nu + n\tau)$ .*

*Proof.* Suppose  $p_i$  becomes hungry at state  $s$  and no processes within two hops away from  $p_i$  will fail. At any state  $s'$  between state  $s$  and the state when  $p_i$  starts to eat,  $F_i(s') \subseteq H_i(s)$  and hence  $|F_i(s')| \leq |H_i(s)|$  from Lemma 9 and the definition of  $F_i(s')$ . From repeated applications of Lemma 13, it follows that  $p_i$  will **start to eat** within  $(M + N + 1)((|H_i(s)| + 1)\nu + 2\nu + \tau)$  units of time, where  $M$  and  $N$  are the numbers of processes in  $H_i(s)$  that become hungry and eating respectively after state  $s$  and before  $p_i$  eats. As  $|H_i(s)|$  is bounded by  $n$ , the algorithm has a failure locality of 2 and a response time of  $O(n^2\nu + n\tau)$ , or  $O(n^2)$ . *End of Proof.*

### 4.3 Remarks

The response time of the extended Algorithm A for detectable failures is bounded above by  $D^2 + 5D$ , while those of Algorithms B and B<sup>+</sup> (using the dining philosophers algorithm in [Cho92]) for undetectable failures are  $O(D^3)$  and  $O(D^2)$ , respectively. The three algorithms are real-time [Rei84a] in that their worst-case (message costs and) response times are independent of the total number of processes in the system; rather they depend only on the maximum number of interactions of which some process is a common member. To the best of our knowledge, these are the first real-time algorithms for this problem which satisfy strong fairness and can cope with process failures. Some of the results in this and the preceding chapters have also been reported in [Tsa92b, Tsa93b].

Table 4.2 shows how our algorithms compare with existing algorithms for binary interactions. The response times of some existing algorithms have not been analyzed, in which cases we provide the lowest possible bounds. The algorithm in [Bag89b] may achieve the indicated response time and failure locality if it adopts an assignment of interaction id's similar to that of our first algorithm.

Algorithms	Message Cost	Response Time	Strong Fairness	Failure Locality
[Sch82]	$O(n)$	$\geq O(n)$	No	$O(n)$
[Buc83]	$O(D)$	$\geq O(n)$	No	$O(n)$
[Sis84]	$O(D)$	$O(D^2L)$	Yes	$O(D)$
[Ram87b]	$O(D)$	$\geq O(n)$	No	$O(n)$
[Bag89b]	$O(D)$	$[O(D^2)]$	No	$[O(D)]$
[Cho92]	$O(D)$	$O(D^2)$	No	$O(D)$
[Cho92]	$O(D^2)$	$O(D^2)$	No	4
Algorithm A	$2D + 2$	$D^2 + 5D$	Yes	$O(D)$
Algorithm B	$O(D^2)$	$O(D^3)$	Yes	5
Algorithm B <sup>+</sup>	$O(D^2)$	$O(D^2)$	Yes	6

Table 4.2: Comparison of algorithms for binary interactions



## CHAPTER 5

### Fairness and UNITY

We give a brief introduction to the UNITY formalism, including its program model and logic. We then show that strong fairness properties can be expressed in UNITY; in fact, we shall prove a general result on the expressive power of conditional UNITY properties, from which the preceding result follows. Existing results about the expressive and deductive powers of UNITY logic are also summarized. Finally, we prove that UNITY logic is relatively complete for proving strong fairness properties. These new results have also been reported in [Tsa93c, Tsa92a].

#### 5.1 An Introduction to UNITY

##### 5.1.1 Program and Execution Model

A UNITY program essentially consists of a finite set of variables, an initial predicate that specifies the initial values of the variables, and a finite set of conditional multiple assignment statements. The execution (computation) of a program starts from a state that satisfies the initial predicate and goes on forever. In each step, an assignment statement is selected nondeterministically for execution subject to the fair selection constraint that each statement is selected infinitely many times. The execution of an assignment statement always terminates and its effect on the current program state is deterministic.

We fix a program  $F$  for the rest of the exposition unless otherwise stated. Let  $R$  denote an arbitrary execution of  $F$  and  $R_i$ ,  $i \geq 0$ , denote the  $i$ -th component of  $R$ . Each  $R_i$  is a pair of  $R_i.state$  and  $R_i.label$ , where  $R_i.state$  is the state of the program before the  $i$ -th step (steps are numbered from 0) and  $R_i.label$  is the name of the statement selected for execution on the  $i$ -th step.  $p[R_i]$  is true if predicate  $p$  holds at  $R_i.state$ ; in particular,  $Init[R_0]$  is true, where  $Init$  denotes the initial predicate of  $F$ .

### 5.1.2 Programming Logic

UNITY logic is based on Hoare triples, namely assertions of the form  $\{p\} s \{q\}$  [Hoa69], which asserts that the execution of statement  $s$  at any state where  $p$  holds will result in a state where  $q$  holds. In the logic, a property of program  $F$  is mainly expressed in the form of “ $p$  unless  $q$  in  $F$ ”, “invariant  $p$  in  $F$ ”, “ $p$  ensures  $q$  in  $F$ ”, or “ $p \mapsto q$  in  $F$ ”. The “in  $F$ ” part is usually omitted, when the program name is known from the context. The logic relations are defined as follows:

$$p \text{ unless } q \text{ in } F \equiv \langle \forall s : s \text{ in } F :: \{p \wedge \neg q\} s \{p \vee q\} \rangle$$

$$\text{invariant } p \text{ in } F \equiv (Init \Rightarrow p) \wedge (p \text{ unless } \text{false} \text{ in } F)$$

$$p \text{ ensures } q \text{ in } F \equiv (p \text{ unless } q \text{ in } F) \wedge (\exists s : s \text{ in } F :: \{p \wedge \neg q\} s \{q\})$$

Another useful form of property “stable  $p$ ” is defined as “ $p$  unless false”.

Program  $F$  has property  $p \mapsto q$  if and only if the property can be derived by a finite number of applications of the following inference rules:

$$\text{(promotion)} \quad \frac{p \text{ ensures } q}{p \mapsto q}$$

$$\text{(transitivity)} \quad \frac{p \mapsto q, q \mapsto r}{p \mapsto r}$$

$$\text{(disjunction)} \text{ For any set } W, \quad \frac{\langle \forall m : m \in W :: p(m) \mapsto q \rangle}{\langle \exists m : m \in W :: p(m) \rangle \mapsto q}$$

As seen convenient for our purpose, we give names to the first conjunct in the definition of **invariant** and the second conjunct in the definition of **ensures**:

$$\text{initially } p \text{ in } F \equiv Init \Rightarrow p$$

$$p \text{ est } q \text{ in } F \equiv \langle \exists s : s \text{ in } F :: \{SI \wedge p \wedge \neg q\} s \{q\} \rangle$$

Hence,

$$\text{invariant } p \equiv (\text{initially } p) \wedge (p \text{ unless } \text{false})$$

$$p \text{ ensures } q \equiv (p \text{ unless } q) \wedge (p \text{ est } q)$$

#### Substitution Axiom

*If  $p = q$  is an invariant of a program,  $p$  can replace  $q$  in all properties of the same program. In particular, any invariant can replace true and vice versa.*

To illustrate the use of these various forms of properties, some examples are in order:

1. “ $x \leq k$  unless  $x < k$ ” says that the value of  $x$  is non-increasing.
2. “invariant  $x \geq 0$ ” says that the value of  $x$  is non-negative at any point of a computation, or simply,  $x$  is always non-negative.
3. “ $x > 0 \mapsto x = 0$ ” says that if  $x$  is positive at any point of a computation then it will eventually become 0 at a later point. However, it does not say that  $x$  will not become negative before it becomes 0 (which may be enforced by adding “ $x > 0$  unless  $x = 0$ ”).

### Notation

It is a UNITY convention that free variables of a property, e.g.  $k$  in the first example, are assumed to be *universally quantified*.

### 5.1.3 Theorems

A UNITY theorem is in the form of an inference rule “ $\frac{\Delta}{Q}$ ” ( $\Delta$  may be empty), asserting that property  $Q$  given in the conclusion can be deduced from the set of properties  $\Delta$  given in the premise;  $\Delta$  and  $Q$  may contain properties that are simply predicates on program states<sup>1</sup>. For instance, below is the famous PSP (Progress-Safety-Progress) Theorem:

$$\frac{p \mapsto q, r \text{ unless } b}{(p \wedge r) \mapsto (q \wedge r) \vee b}$$

The above theorem is in fact a theorem scheme and can be instantiated for different programs. A theorem is said to be “a theorem of program  $F$ ” if the properties in its premise and conclusion are intended to be properties of a program named  $F$ .

We list a number of theorems which will be used in subsequent sections.

#### Reflexivity and Antireflexivity of *unless*

$$p \text{ unless } p \qquad \neg p \text{ unless } p$$

#### Consequence Weakening of *unless*

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

---

<sup>1</sup>In fact, “initially  $p$ ” is simply a predicate.

**Conjunction of *unless***

$$\frac{p \text{ unless } q, p' \text{ ensures } q'}{(p \wedge p') \text{ ensures } (p \wedge q') \vee (p' \wedge q) \vee (q \wedge q')}$$

**Simple Conjunction of *unless***

$$\frac{p \text{ unless } q, p' \text{ unless } q'}{(p \wedge p') \text{ unless } (q \vee q')}$$

**Disjunction of *unless***

$$\frac{p \text{ unless } q, p' \text{ ensures } q'}{(p \vee p') \text{ ensures } (\neg p \wedge q') \vee (\neg p' \wedge q) \vee (q \wedge q')}$$

**Simple Disjunction of *unless***

$$\frac{p \text{ unless } q, p' \text{ unless } q'}{(p \vee p') \text{ unless } (q \vee q')}$$

A corollary of the preceding two theorems is “*true unless p*”.

**Implication Theorem<sup>2</sup> of  $\mapsto$**

$$\frac{\text{invariant } p \Rightarrow q}{p \mapsto q}$$

**Finite Disjunction of  $\mapsto$**

$$\frac{p \mapsto q, p' \mapsto q'}{p \vee p' \mapsto q \vee q'}$$

**Cancellation Theorem of  $\mapsto$**

$$\frac{p \mapsto q \vee b, b \mapsto r}{p \mapsto q \vee r}$$

Let  $M$  denote a (ranking) function from program states to a well-founded set under relation  $\prec$ .

**Induction Principle of  $\mapsto$**

$$\frac{p \wedge (M = m) \mapsto (p \wedge (M \prec m)) \vee q}{p \mapsto q}$$

---

<sup>2</sup>We have replaced the premise “ $p \Rightarrow q$ ” in the original theorem by “invariant  $p \Rightarrow q$ ”, extending the applicability of the theorem.

### 5.1.4 Program Composition

We are interested in the most common type of parallel composition called *union*. The union of two programs  $F$  and  $G$  is denoted by  $F\parallel G$ . The set of variables (statements) of  $F\parallel G$  is the union, as the name suggests, of the sets of variables (statements) of  $F$  and  $G$ . For convenience, each component program of a composite program is called a *module*. Variables belonging to more than one modules are termed *shared* variables. One important constraint on such kind of program composition is that each shared variable should be declared and initialized “consistently” by the sharing modules. In a computation of  $F\parallel G$ , each statement of  $F$  or  $G$  must be selected infinitely often. A computation of  $F$  is no longer a computation of  $F\parallel G$ , since the statements of  $G$  are not selected; analogously for  $G$ . A predicate is a *local predicate* of  $F$  if it mentions only variables that can be modified by  $F$  alone.

The following four theorems on composing properties of individual programs will be useful in the next chapter:

#### Union Theorems

$$\frac{a \text{ unless } b \text{ in } F, \text{ stable } a \text{ in } G}{a \text{ unless } b \text{ in } F\parallel G.}$$

$$\frac{\text{invariant } a \text{ in } F, \text{ stable } a \text{ in } G}{\text{invariant } a \text{ in } F\parallel G}$$

$$\frac{a \text{ ensures } b \text{ in } F, \text{ stable } a \text{ in } G}{a \text{ ensures } b \text{ in } F\parallel G}$$

If any of the following properties holds in  $F$ , where  $p$  is a local predicate of  $F$ , then it also holds in  $F\parallel G$  for any  $G$ :  $p$  unless  $q$ ,  $p$  ensures  $q$ , and invariant  $p$ .

#### Remark

As pointed out in the original work of UNITY [Cha88], for the above theorems to work, **any** invariant that is used in applying substitution axiom to deduce a property of one module should also be proved to be an invariant of the other module.

### 5.1.5 Conditional Properties

*Conditional* properties are another form of UNITY properties. Properties in terms of the relations defined in Section 5.1.2 are *unconditional* properties; for

convenience, we also treat predicates as unconditional properties. A conditional property consists of a *hypothesis* and a *conclusion*, each of which is a set of unconditional properties. A program  $F$  is said to have a conditional property if, *given the hypothesis as a premise, the conclusion can be proved from  $F$* , i.e. every property in the conclusion can be deduced from the properties in the hypothesis plus those directly provable from  $F$ .

Assume that program  $F$  has a conditional property “Hypothesis:  $\Delta$  Conclusion:  $Q$ ”. It follows from definition that “ $\frac{\Delta \cup \bar{\Delta}}{Q}$ ” is a theorem of  $F$ , for some  $\bar{\Delta}$  which is a set of properties provable from  $F$ .

## 5.2 Expressiveness of UNITY

For the purpose of subsequent sections, we adopt the notion of strongest invariant [San91a, San91b] to incorporate the power of the substitution axiom into the definitions of the UNITY logic relations.

The strongest invariant (of program  $F$ ), denoted as  $SI$ , is defined as the strongest solution of  $X$  in the following equation:

$$(Init \Rightarrow X) \wedge \langle \forall s : s \text{ in } F :: \{X\}s\{X\} \rangle.$$

$SI$  characterizes the set of reachable states, i.e. the program states that may appear in some execution of  $F$ . (Consequently,  $SI[R_i]$ ,  $i \geq 0$ , is true for any execution  $R$  of  $F$ .)

Based on  $SI$ , the fundamental logic relations are defined as follows:

$$p \text{ unless } q \text{ in } F \equiv \langle \forall s : s \text{ in } F :: \{SI \wedge p \wedge \neg q\}s\{p \vee q\} \rangle$$

$$\text{initially } p \text{ in } F \equiv Init \Rightarrow p$$

$$\text{invariant } p \text{ in } F \equiv (\text{initially } p \text{ in } F) \wedge (p \text{ unless } false \text{ in } F) \quad (\equiv SI \Rightarrow p)$$

$$p \text{ est } q \text{ in } F \equiv \langle \exists s : s \text{ in } F :: \{SI \wedge p \wedge \neg q\}s\{q\} \rangle$$

$$p \text{ ensures } q \text{ in } F \equiv (p \text{ unless } q \text{ in } F) \wedge (p \text{ est } q \text{ in } F)$$

“ $p \mapsto q$ ” is as defined in Section 5.1.2.

Now, in stead of substitution axiom we have *substitution theorem*; in particular, we have the following special case:

**Substitution Theorem (Specialized)** [San91a]

$SI$  can be replaced by  $true$  and  $true$  by  $SI$  in any property.

### 5.2.1 Unconditional Properties

We define a set of operational properties with respect to an arbitrary execution  $R$  of program  $F^3$ :

$$R \models p \text{ unless}^L q \equiv \langle \forall i : i \geq 0 :: (p \wedge \neg q)[R_i] \Rightarrow (p \vee q)[R_{i+1}] \rangle$$

$$R \models \text{initially}^L p \equiv p[R_0]$$

$$R \models \text{invariant}^L p \equiv (R \models \text{initially}^L p) \wedge (R \models p \text{ unless}^L \text{false}) \quad (\equiv \langle \forall i : i \geq 0 :: p[R_i] \rangle)$$

$$R \models p \text{ est}^L q \equiv \langle \exists s : s \text{ in } F :: \langle \forall i : i \geq 0 :: ((p \wedge \neg q)[R_i] \wedge R_i.\text{label} = s) \Rightarrow q[R_{i+1}] \rangle \rangle$$

$$R \models p \text{ ensures}^L q \equiv (R \models p \text{ unless}^L q) \wedge (R \models p \text{ est}^L q)$$

$$R \models p \mapsto^L q \equiv \langle \forall i : i \geq 0 :: p[R_i] \Rightarrow \langle \exists j : j \geq i :: q[R_j] \rangle \rangle.$$

The following lemma states the operational implication of unconditional properties and hence the expressive power of the fundamental logic relations. (a), (c), and (f) in the lemma are known results; in fact, their converses are also true [Jut89, Kna92, San91a]. These three results are restated in Lemma 16 using linear time temporal logic notation.

**Lemma 14** (a) *If “ $p$  unless  $q$ ” is provable from  $F$ , then for any execution  $R$  of  $F$ ,  $R \models p$  unless<sup>L</sup>  $q$ .*

(b) *If “initially  $p$ ” is provable from  $F$ , then for any execution  $R$  of  $F$ ,  $R \models$  initially<sup>L</sup>  $p$ .*

(c) *If “invariant  $p$ ” is provable from  $F$ , then for any execution  $R$  of  $F$ ,  $R \models$  invariant<sup>L</sup>  $p$ .*

(d) *If “ $p$  est  $q$ ” is provable from  $F$ , then for any execution  $R$  of  $F$ ,  $R \models p$  est<sup>L</sup>  $q$ .*

(e) *If “ $p$  ensures  $q$ ” is provable from  $F$ , then for any execution  $R$  of  $F$ ,  $R \models p$  ensures<sup>L</sup>  $q$ .*

(f) *If “ $p \mapsto q$ ” is provable from  $F$ , then for any execution  $R$  of  $F$ ,  $R \models p \mapsto^L q$ .*

*Proof.* The proofs of (a), (c), and (f) can be found in [Jut89, Kna92, San91a]. We shall prove (d); (b) is trivial and (e) follows from (a) and (d).

---

<sup>3</sup>We have defined “ $p$  unless<sup>L</sup>  $q$ ”, “invariant<sup>L</sup>  $p$ ”, and “ $p \mapsto^L q$ ” in such a way that they are semantically equivalent to linear time temporal properties: “ $\Box(p \Rightarrow (p \mathcal{W} q))$ ”, “ $\Box p$ ”, and “ $\Box(p \Rightarrow \Diamond q)$ ” respectively.

$$\begin{aligned}
& p \text{ est } q \text{ in } F \\
\equiv & \langle \exists s : s \text{ in } F :: \{SI \wedge p \wedge \neg q\} s \{q\} \rangle \\
& \text{(We shall omit the range of an quantified variable, which is known} \\
& \text{from the context.)} \\
\Rightarrow & \langle \exists s :: \langle \forall R, i :: ((SI \wedge p \wedge \neg q)[R_i] \wedge R_i.\text{label} = s) \Rightarrow q[R_{i+1}] \rangle \rangle \\
\Rightarrow & \langle \exists s :: \langle \forall R, i :: ((p \wedge \neg q)[R_i] \wedge R_i.\text{label} = s) \Rightarrow q[R_{i+1}] \rangle \rangle \\
\Rightarrow & \langle \forall R :: \langle \exists s :: \langle \forall i :: ((p \wedge \neg q)[R_i] \wedge R_i.\text{label} = s) \Rightarrow q[R_{i+1}] \rangle \rangle \rangle \\
\equiv & \langle \forall R :: R \models p \text{ est}^L q \rangle
\end{aligned}$$

*End of Proof.*

The operational properties implied by “ $p$  unless  $q$ ” and “ $p \mapsto q$ ” are illustrated in Figures 5.1 and 5.2, respectively.

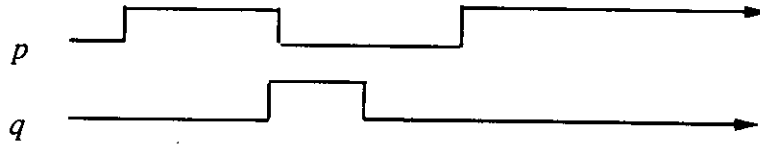


Figure 5.1: The operational implication of “ $p$  unless  $q$ ”

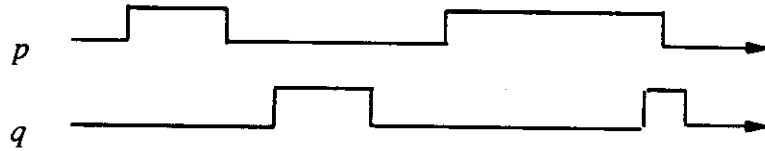


Figure 5.2: The operational implication of “ $p \mapsto q$ ”

The implied operational property may be considered as being “specified” by the corresponding unconditional UNITY property. Of particular importance, “ $p \mapsto^L q$ ” specifies exactly the same property on an execution as that by the temporal logic formula “ $\Box(p \Rightarrow \Diamond q)$ ”, which perhaps is the most common form of liveness property. Consequently, properties of the form “ $\Box \Diamond p$ ” (= “ $\Box(\text{true} \Rightarrow \Diamond p)$ ”) can be expressed by “ $\text{true} \mapsto p$ ”. With the assistance of an auxiliary predicate, properties of the form “ $\Diamond \Box p$ ” can be expressed by “ $\text{true} \mapsto p \wedge b$ , stable  $p \wedge b$ ”, where  $b$  is an auxiliary predicate [Mis90].

### 5.2.2 Conditional Properties

No result analogous to Lemma 14 for conditional properties has been provided in the literature. The hypothesis and the conclusion of a conditional property, in



general, may contain properties of *different* programs (which facilitates composition of program properties). As we are interested in the operational implication of conditional properties, we restrict our attention to those involving properties of a *same* program.

An operational implication that has been suggested for “Hypothesis:  $\Delta_1$  Conclusion:  $\Delta_2$ ” is as follows: The operational properties corresponding to  $\Delta_1$  are satisfied by all executions of the program implies those corresponding to  $\Delta_2$  are also satisfied by all executions; or equivalently, either the operational properties corresponding to  $\Delta_1$  are not satisfied by some execution of the program or those corresponding to  $\Delta_2$  are satisfied by all executions. This implication is very weak and does not reflect the expressive power of conditional properties.

In this section we deduce a stronger operational implication from “Hypothesis:  $\Delta_1$  Conclusion:  $\Delta_2$ ”, asserting that “for every execution of the program, if the operational properties corresponding to  $\Delta_1$  are satisfied by the execution then those corresponding to  $\Delta_2$  are also satisfied by the execution.” This result extends the class of operational properties that can be specified in UNITY. In order to prove the stronger result, we must first investigate the soundness of UNITY theorems (and inference rules). A UNITY theorem in the form of “ $\frac{\Delta}{Q}$ ” states that property  $Q$  can be deduced from the properties in  $\Delta$  ( $\Delta$  and  $Q$  may contain properties that are simply predicates). Assume that a program has conditional property “Hypothesis:  $\Delta$  Conclusion:  $Q$ ”. It follows from definition that there is a set of properties  $\Delta'$  directly provable from the program such that  $\frac{\Delta \cup \Delta'}{Q}$  is a theorem.

We prove the following strong soundness result: If  $\frac{\Delta}{Q}$  is a theorem, then for every execution of the program in question, the operational properties corresponding to  $\Delta$  are satisfied by the execution implies the one corresponding to  $Q$  is also satisfied by the execution. From this soundness result, the stronger operational implication of conditional properties follows immediately.

### 5.2.2.1 Proving UNITY Theorems

To prove a UNITY theorem, one resorts to the following:

1. the underlying calculus for predicates on program states,
2. the definitions of *unless*, *initially*, *invariant*, *est*, and *ensures*,
3. the definition (and hence the three inference rules) of  $\mapsto$ , and
4. the inference rules for Hoare triples:

$$\begin{array}{c}
\frac{}{\{p\} s \{true\}} \\
\frac{}{\{false\} s \{q\}} \\
\frac{}{\{p\} s \{false\}} \\
\neg p \\
\frac{\langle \forall j :: \{p_j\} s \{q_j\} \rangle}{\{\langle \forall j :: p_j \rangle\} s \{\langle \forall j :: q_j \rangle\}} \\
\frac{\langle \forall j :: \{p_j\} s \{q_j\} \rangle}{\{\langle \exists j :: p_j \rangle\} s \{\langle \exists j :: q_j \rangle\}} \\
\frac{p' \Rightarrow p, \{p\} s \{q\}, q \Rightarrow q'}{\{p'\} s \{q'\}}
\end{array}$$

In each step of the proof, a property is deduced from a number of others, each of which is either included in the premise or deduced in a previous step; some property may be deduced from an empty set of properties, including a logically valid predicate and a property that can be derived using the first two rules for Hoare triples. When  $\mapsto$  properties appear in the premise, it is often necessary to do a structural induction on the length of the proof of a  $\mapsto$  property; sometimes more than one levels of induction are required. In the base case and the induction steps of the inductive proof, one in turn proves various simpler theorems. A proved theorem may be applied as an inference rule in the proof of another theorem; the application is viewed as a “macro” proof step, which can be replaced by the proof of the theorem applied.

For convenience, a theorem is said to be *simple* if it can be proved without using the aforementioned structural induction. The Conjunction Theorem of *unless* and *ensures*, the Implication Theorem<sup>4</sup>, and the Cancellation Theorem of  $\mapsto$  shown below in this order are all simple theorems.

$$\frac{p \text{ unless } q, p' \text{ ensures } q'}{(p \wedge p') \text{ ensures } (p \wedge q') \vee (p' \wedge q) \vee (q \wedge q')}$$

$$\frac{\text{invariant } p \Rightarrow q}{p \mapsto q} \\
\frac{p \mapsto q \vee b, b \mapsto r}{p \mapsto q \vee r}$$

---

<sup>4</sup>Again, we have replaced the premise “ $p \Rightarrow q$ ” in the original Implication Theorem by “**invariant**  $p \Rightarrow q$ ”.

Particularly, the Conjunction Theorem of *unless* and *ensures* can be proved using the definitions of *unless* and *ensures* and the conjunction, pre-condition strengthening, and post-condition weakening rules for Hoare triples; while the Cancellation Theorem can be proved using the Implication Theorem (which is assumed to be a proved theorem) and the transitivity and disjunction rules for  $\mapsto$ .

The PSP Theorem shown again below, on the other hand, has to be proved using structural induction:

$$\frac{p \mapsto q, r \text{ unless } b}{(p \wedge r) \mapsto (q \wedge r) \vee b}$$

In the base case of its proof, one typically will apply the Conjunction Theorem of *unless* and *ensures*; while in the induction steps, one may apply the Cancellation Theorem.

### 5.2.2.2 Strong Soundness of UNITY Theorems

Following the inference rules for  $\mapsto$ , we propose analogous rules for  $\mapsto^L$ :

$$\begin{array}{l} \text{(promotion)} \quad \frac{p \text{ ensures}^L q}{p \mapsto^L q} \\ \text{(transitivity)} \quad \frac{p \mapsto^L q, q \mapsto^L r}{p \mapsto^L r} \\ \text{(disjunction)} \text{ For any set } W, \quad \frac{\langle \forall m : m \in W :: p(m) \mapsto^L q \rangle}{\langle \exists m : m \in W :: p(m) \rangle \mapsto^L q} \end{array}$$

The soundness of the preceding inference rules for  $\mapsto^L$  can be verified using the definitions of *ensures*<sup>L</sup> and  $\mapsto^L$  and the fairness constraint in the program execution model.

Below are the operational versions of the Implication Theorem and the Cancellation Theorem, which can be proved from the definitions of *invariant*<sup>L</sup> and *ensures*<sup>L</sup> and the inference rules for  $\mapsto^L$ . These results will be needed in the proof of Lemma 15.

$$\frac{\text{invariant}^L p \Rightarrow q}{p \mapsto^L q}$$

$$\frac{p \mapsto^L q \vee b, b \mapsto^L r}{p \mapsto^L q \vee r}$$

**Lemma 15** *For any execution  $R$  of  $F$ , if  $R \models p \mapsto^L q$ , then  $p \mapsto^L q$  can be deduced using the promotion, transitivity, and disjunction rules for  $\mapsto^L$ .*

*Proof.* We postulate an auxiliary program variable that is initialized to 0 and is incremented by 1 in each step of an execution of  $F$ . Given an execution  $R$  of  $F$  such that  $R \models p \mapsto^L q$ , we can construct from  $R$  a function  $\delta$ , mapping from program states to positive integers, such that  $\delta$  is defined only when  $p \wedge \neg q$  holds and it counts the number of steps to the nearest future state where  $q$  holds. Hence,  $\delta$  satisfies the following two conditions:

$$\begin{aligned} &\langle \forall i : i \geq 0 :: ((p \wedge \neg q) \Rightarrow \langle \exists \alpha :: (\delta = \alpha) \rangle)[R_i] \rangle \\ &\langle \forall i : i \geq 0 :: (\delta = \alpha)[R_i] \Rightarrow (q \vee (\delta < \alpha))[R_{i+1}] \rangle \end{aligned}$$

The rest of the proof is similar to that of the relative completeness of UNITY in [San91a]. We need to show that, from the preceding two conditions of  $\delta$  we can deduce  $p \mapsto^L q$  using the three inference rules for  $\mapsto^L$ .

$$\begin{aligned} &\langle \forall \alpha :: (\delta = \alpha) \text{ ensures}^L (q \vee (\delta < \alpha)) \rangle && \text{, from the second condition.} \\ &\langle \forall \alpha :: (\delta = \alpha) \mapsto^L q \vee (\delta < \alpha) \rangle && \text{, promotion on the above.} \\ &\langle \forall \alpha :: (\delta = \alpha) \mapsto^L q \rangle && \text{, induction on the above (proved} \\ & && \text{separately below).} \\ &\langle \exists \alpha :: (\delta = \alpha) \rangle \mapsto^L q && \text{, disjunction on the above.} \quad (1) \\ \text{invariant } (p \wedge \neg q) \Rightarrow \langle \exists \alpha :: (\delta = \alpha) \rangle && \text{, rewriting of the first condition.} \\ p \wedge \neg q \mapsto^L \langle \exists \alpha :: (\delta = \alpha) \rangle && \text{, Implication Thm. on the above.} \\ p \wedge \neg q \mapsto^L q && \text{, transitivity on the above and (1).} \\ p \wedge q \mapsto^L q && \text{, Implication Theorem on} \\ && \text{invariant } (p \wedge q) \Rightarrow q. \\ p \mapsto^L q && \text{, disjunction on the above two.} \end{aligned}$$

The detail of the inductive proof step is as follows:

Base case:  $(\delta = 1) \mapsto^L q$ . (From the second condition and that the minimum value of  $\delta$  is 1.)

Induction step:

$$\begin{aligned} &\langle \forall \beta : \beta < \alpha :: (\delta = \beta) \mapsto^L q \rangle && \text{, hypothesis.} \\ &\langle \exists \beta : \beta < \alpha :: (\delta = \beta) \rangle \mapsto^L q && \text{, disjunction on the above.} \\ &\langle \forall \alpha :: (\delta = \alpha) \mapsto^L q \vee (\delta < \alpha) \rangle && \text{, given.} \\ &\langle \forall \alpha :: (\delta = \alpha) \mapsto^L q \rangle && \text{, Cancellation Theorem on the above two.} \\ & && \text{End of Proof.} \end{aligned}$$

We use  $Q^L$  to denote the operational property that corresponds to an unconditional UNITY property  $Q$  in the sense of Lemma 14; analogously, for a set of

properties. For convenience, the corresponding “operational property”  $Q^L$  of a predicate  $Q$  denotes  $Q$  itself. “ $R \models \Delta^L$ ”, where  $\Delta^L$  is a set of properties, denotes that “for each  $Q^L$  in  $\Delta^L$ ,  $R \models Q^L$ ”. We stipulate that “ $R \models Q^L$ ” actually means “ $Q^L$ ” if  $Q$  is simply a predicate.

**Theorem 13 (Strong Soundness)** *If  $\frac{\Delta}{Q}$  is a theorem of  $F$ , then for any execution  $R$  of  $F$ ,  $R \models \Delta^L$  implies  $R \models Q^L$ .*

*Proof.* From the assumption, there exists a proof of  $\frac{\Delta}{Q}$  in UNITY. To prove that  $R \models \Delta^L$  implies  $R \models Q^L$ , we simply imitate step by step a given UNITY proof of  $\frac{\Delta}{Q}$ . We start with a set of operational properties  $\Delta^L$  that correspond to the properties in  $\Delta$ ; in each proof step, an operational property is deduced that corresponds to the property deduced in the corresponding step of the given UNITY proof. It remains to show that, for each inference rule applied in the UNITY proof, the corresponding rule for the imitating proof indeed exists and is sound. We observe immediately that the underlying calculus for predicates on program states used in the UNITY proof is equally applicable to the imitating proof.

For the case of simple theorems (as defined in Section 5.2.2.1), we only need to (a) replace the applications of promotion, transitivity, and disjunction rules for  $\mapsto$  by those of the corresponding inference rules for  $\mapsto^L$  and (b) the application of an inference rule for Hoare triples by that of its operational counterpart shown below:

$$\begin{array}{c}
\frac{}{\{p\} s \{true\}} \qquad \frac{}{p[R_i] \Rightarrow true[R_{i+1}]} \\
\frac{}{\{false\} s \{q\}} \qquad \frac{}{false[R_i] \Rightarrow q[R_{i+1}]} \\
\frac{}{\{p\} s \{false\}} \qquad \frac{}{p[R_i] \Rightarrow false[R_{i+1}]} \\
\qquad \qquad \qquad \neg p \qquad \qquad \qquad \neg p[R_i] \\
\frac{\langle \forall j :: \{p_j\} s \{q_j\} \rangle}{\{\langle \forall j :: p_j \rangle\} s \{\langle \forall j :: q_j \rangle\}} \qquad \frac{\langle \forall j :: (p_j)[R_i] \Rightarrow (q_j)[R_{i+1}] \rangle}{(\langle \forall j :: p_j \rangle)[R_i] \Rightarrow (\langle \forall j :: q_j \rangle)[R_{i+1}]} \\
\frac{\langle \forall j :: \{p_j\} s \{q_j\} \rangle}{\{\langle \exists j :: p_j \rangle\} s \{\langle \exists j :: q_j \rangle\}} \qquad \frac{\langle \forall j :: (p_j)[R_i] \Rightarrow (q_j)[R_{i+1}] \rangle}{(\langle \exists j :: p_j \rangle)[R_i] \Rightarrow (\langle \exists j :: q_j \rangle)[R_{i+1}]} \\
\frac{p' \Rightarrow p, \{p\} s \{q\}, q \Rightarrow q'}{\{p'\} s \{q'\}} \qquad \frac{p' \Rightarrow p, p[R_i] \Rightarrow q[R_{i+1}], q \Rightarrow q'}{p'[R_i] \Rightarrow q'[R_{i+1}]}
\end{array}$$

It is clear that the inference rules in the second column are sound; the inference rules for  $\mapsto^L$  are also sound as indicated previously.

For theorems that are not simple, the structural inductions on  $\mapsto$  properties in the UNITY proof can be translated into inductions on the corresponding  $\mapsto^L$  properties thanks to Lemma 15. *End of Proof.*

### 5.2.2.3 Operational Implication of Conditional Properties

**Theorem 14** *If “Hypothesis:  $\Delta_1$  Conclusion:  $\Delta_2$ ” is provable from  $F$ , then for any execution  $R$  of  $F$ ,  $R \models \Delta_1^L$  implies  $R \models \Delta_2^L$ .*

*Proof.* We consider each property in  $\Delta_2$  separately. Fix a property  $Q$  in  $\Delta_2$ .

The assumption of the theorem implies that there is a set of properties  $\bar{\Delta}_1$  provable from  $F$  such that  $\frac{\Delta_1 \cup \bar{\Delta}_1}{Q}$  is a theorem of  $F$ . From the Strong Soundness Theorem, it follows that  $R \models \Delta_1^L \cup \bar{\Delta}_1^L$  implies  $R \models Q^L$ . As properties in  $\bar{\Delta}_1$  are provable from  $F$ , for any execution  $R$  of  $F$ ,  $R \models \bar{\Delta}_1^L$  (Lemma 14). Consequently, for any execution  $R$  of  $F$ ,  $R \models \Delta_1^L$  implies  $R \models Q^L$ .

Putting the result for each property in  $\Delta_2$  together, we conclude that for any execution  $R$  of  $F$ ,  $R \models \Delta_1^L$  implies  $R \models \Delta_2^L$ . *End of Proof.*

The operational properties corresponding to *unless*, *initially*, *invariant*, and  $\mapsto$  properties (excluding *est*) are expressible in the usual linear temporal logic notation, so are their boolean combinations. The following concluding theorem, which is an immediate corollary of Theorem 14, identifies the class of operational properties that can be specified in UNITY. Note that unconditional properties can be treated as conditional properties with empty hypotheses.

**Theorem 15** *Suppose  $P_i$ ,  $1 \leq i \leq m$ , and  $Q_i$ ,  $1 \leq i \leq n$ , are *unless*, *initially*, *invariant*, or  $\mapsto$  properties. If “Hypothesis:  $P_1, P_2, \dots, P_m$  Conclusion:  $Q_1, Q_2, \dots, Q_n$ ” is provable from  $F$ , then for any execution  $R$  of  $F$ , “ $R \models (P_1^L \wedge P_2^L \wedge \dots \wedge P_m^L) \Rightarrow (Q_1^L \wedge Q_2^L \wedge \dots \wedge Q_n^L)$ ”.*

Together with the results on unconditional properties, it follows that a strong fairness property “ $\Box \Diamond p \Rightarrow \Box \Diamond q$ ” can be expressed by “Hypothesis: *true*  $\mapsto p$  Conclusion: *true*  $\mapsto q$ ”.

## 5.3 Deducing Fairness Properties in UNITY

An algorithm that satisfies a strong fairness property may be described in the UNITY program model which assumes a weaker notion of fairness. The availabil-

one can deduce in temporal logic “ $(\Box\Diamond p \vee \Box\Diamond q) \Rightarrow Q$ ”. As “ $(\Box\Diamond p \vee \Box\Diamond q)$ ” = “ $\Box\Diamond(p \vee q)$ ”  $\tau$  “ $(\Box\Diamond p \vee \Box\Diamond q) \Rightarrow Q$ ” is equivalent to “ $\Box\Diamond(p \vee q) \Rightarrow Q$ ”. Recall that “ $\Box\Diamond(p \vee q)$ ” can be expressed as “ $true \mapsto p \vee q$ ” in UNITY. Hence, we propose the following rule:

### Disjunctivity of Infinitely Often

$$\frac{\text{Hypothesis: } true \mapsto p \quad \text{Conclusion: } Q, \quad \text{Hypothesis: } true \mapsto q \quad \text{Conclusion: } Q}{\text{Hypothesis: } true \mapsto p \vee q \quad \text{Conclusion: } Q}$$

The rule is formulated in its present form due to the fact that there is no notion of *disjunction* of properties in UNITY.

In temporal logic, one can prove “ $\Box\Diamond p \Rightarrow \Box\Diamond q$ ” by showing that “ $\Box\Diamond p \wedge \Diamond\Box\neg q \Rightarrow false$ ”. Recall that “ $\Diamond\Box\neg q$ ” can be expressed as “ $true \mapsto \neg q \wedge b$ , **stable**  $\neg q \wedge b$ ”, where  $b$  is some auxiliary predicate. We therefore propose a second rule:

### Proof by Contradiction

$$\frac{\text{Hypothesis: } \Delta, true \mapsto \neg q \wedge b, \text{ **stable** } \neg q \wedge b \quad \text{Conclusion: **initially false**}}{\text{Hypothesis: } \Delta \quad \text{Conclusion: } true \mapsto q}$$

The above two inference rules are not provable in UNITY but can be justified by Theorems 14 and 16 in last chapter.

### Notation

For brevity, we shall use temporal logic notations to abbreviate certain UNITY properties.

“ $\Box\Diamond p$ ” abbreviates “ $true \mapsto p$ ”; “ $\Diamond\Box p$ ” abbreviates “ $true \mapsto p \wedge b$ , **stable**  $p \wedge b$ ”, where  $b$  is some auxiliary predicate.

“ $\Box\Diamond p \Rightarrow \Box\Diamond q$ ” abbreviates “Hypothesis:  $true \mapsto p$  Conclusion:  $true \mapsto q$ ”. The alternative form “ $\Diamond\Box p \Rightarrow \Diamond\Box q$ ” abbreviates “Hypothesis:  $true \mapsto p \wedge b$ , **stable**  $p \wedge b$  Conclusion:  $true \mapsto q \wedge b'$ , **stable**  $q \wedge b'$ ”. These abbreviations extend to **conditional** properties where the hypothesis and conclusion contain more than **one** properties.

## 6.2 Specification of the Problem

We adapt the UNITY format of problem specification [Cha88]. In particular, we express a conditional property simply as an unconditional property *when the hypothesis of the conditional property is empty (or true)*.

Analogous to Section 2.3, let  $USER$  refer to a program which contains a set of asynchronous processes and  $OS$  refer to the scheduler that implements synchronizations among the asynchronous processes in  $USER$ . The composite program  $USER \parallel OS$  is referred to as  $\mathcal{P}$ .

We assume processes in  $USER$  are numbered from 1 through  $n$  and the  $i$ -th process is denoted by  $user_i$ ; analogously for processes in  $OS$ .  $p_i \equiv user_i \parallel os_i$  denotes the  $i$ -th process in  $\mathcal{P}$ . Again, we shall refer to a process in  $USER$  as a **user**, a process in  $OS$  as an **os**, and a process in  $\mathcal{P}$  as a **process**. A binary interaction between  $user_i$  and  $user_j$  is represented by  $\{i, j\}$ .  $\mathcal{I}$  is the fixed set of all binary interactions defined among users; each element of  $\mathcal{I}$  is a two-element subset of  $\{1, 2, \dots, n\}$ . Two different interactions are said to be *conflicting* if they have one common member. The set of all interactions of which a **user** (or loosely, a **process**) is a member is referred to as the *interaction set* of the user (process).

Each **user** and the corresponding **os** share two variables: **state** and **flag**. **state** may assume the value *active* or *idle*. Interaction  $\{i, j\}$  is started if one of its members, say  $p_i$ , sets  $flag_i$  to  $\{i, j\}$  and is terminated if **flag** is set back to *null* for both  $p_i$  and  $p_j$ . We say a process is *engaged* if some interaction in its interaction set is started. The relationship of users, os's and their compositions is shown in Figure 6.1.

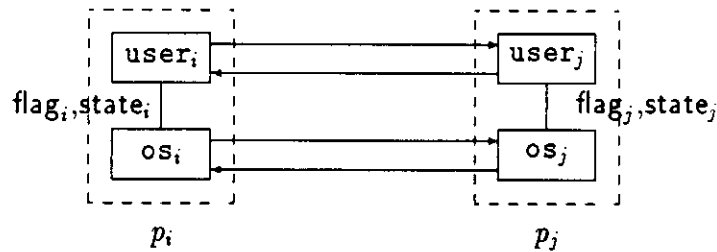


Figure 6.1: Compositions of users and os's

We define some frequently used predicates:

$$idle_i \equiv (state_i = idle), \text{ analogously for } active_i \quad (d1)$$

$$enable^{\{i,j\}} \equiv (idle_i \wedge idle_j) \quad (d2)$$

$$start^{\{i,j\}} \equiv (flag_i = \{i, j\} \vee flag_j = \{i, j\}) \quad (d3)$$

$$engaged_i \equiv [\exists j : \{i, j\} \in \mathcal{I} :: start^{\{i,j\}}] \quad (d4)$$

As usual, all properties are assumed to be universally quantified over all values of their free variables (e.g., process indexes usually appear as free variables). For brevity, we shall hereafter omit phrases such as " $\{i, j\} \in \mathcal{I}$ " when no confusion may rise.



$$\begin{aligned}
&\cong p \wedge SI \wedge \neg q \wedge (M = m) \mapsto (M \prec m) \vee q \\
&\quad , \text{Lemma 16.} \\
&= p \wedge \neg q \wedge (M = m) \mapsto (M \prec m) \vee q \\
&\quad , \text{substitution theorem.}
\end{aligned}$$

The third premise can be further rewritten as  $p \wedge (M = m) \mapsto (M \prec m) \vee q$  due to the following results:

$$\begin{aligned}
p \wedge \neg q \wedge (M = m) \mapsto (M \prec m) \vee q &\quad , \text{ given.} \\
p \wedge q \wedge (M = m) \mapsto q &\quad , \text{ implication theorem on } q \wedge \dots \Rightarrow q. \\
p \wedge (M = m) \mapsto (M \prec m) \vee q &\quad , \text{ finite disjunction on the above two.} \\
\\
p \wedge (M = m) \mapsto (M \prec m) \vee q &\quad , \text{ given.} \\
p \wedge \neg q \wedge (M = m) \mapsto p \wedge (M = m) &\quad , p \wedge \neg q \wedge (M = m) \Rightarrow p \wedge (M = m). \\
p \wedge \neg q \wedge (M = m) \mapsto (M \prec m) \vee q &\quad , \text{ transitivity on the above two.}
\end{aligned}$$

We now proceed to the second step, i.e. to prove the following inference rule:  
SF-UNITY

$$\frac{M \preceq m \text{ unless } q \quad p \wedge (M = m) \mapsto (M \prec m) \vee q}{\text{Hypothesis: } true \mapsto p \quad \text{Conclusion: } true \mapsto q}$$

$$\begin{aligned}
M \preceq k \text{ unless } q &\quad , \text{ the first premise.} \\
M \prec m \text{ unless } q &\quad , \text{ simple disjunction over all } k \prec m. \\
true \mapsto p &\quad , \text{ the hypothesis in the conclusion.} \\
M \prec m \mapsto (p \wedge (M \prec m)) \vee q &\quad , \text{ PSP theorem on the above two.} \\
p \wedge (M = m) \mapsto (M \prec m) \vee q &\quad , \text{ the second premise.} \\
p \wedge (M = m) \mapsto (p \wedge (M \prec m)) \vee q &\quad , \text{ cancellation on the above two.} \\
p \mapsto q &\quad , \text{ induction on the above.} \\
true \mapsto q &\quad , \text{ transitivity on } true \mapsto p \text{ and the} \\
&\quad \text{above.}
\end{aligned}$$

*End of Proof.*

In the process of proving Theorem 16, we obtain the inference rule SF-UNITY for deducing strong fairness properties in UNITY.

Operationally, SF-UNITY can be reasoned as follows: The value of  $M$  is non-increasing in the absence of  $q$ . An occurrence of  $p$  will either decrease the value

of  $M$  or establish  $q$ . Due to well-foundedness of  $M$ , enough occurrences of  $p$  will eventually establish  $q$ ; and infinitely many occurrences of  $p$  will guarantee infinitely many occurrences of  $q$ .

SF-UNITY is sound, which follows from its proof in Theorem 16. It is relatively complete, from the proof of Theorem 16 and the relative completeness of the specialized B-REAC. We summarize by the following theorem:

**Theorem 17** *Rule SF-UNITY is sound and relatively complete for proving strong fairness properties of a UNITY program.*

### Remark

A referee of one of our submitted papers suggested that the first premise of SF-UNITY can be weakened such that the following is also a valid inference rule:

$$\frac{\begin{array}{l} M \preceq m \text{ unless } (p \wedge (M \preceq m)) \vee q \\ p \wedge (M = m) \mapsto (M \prec m) \vee q \end{array}}{\text{Hypothesis: } true \mapsto p \quad \text{Conclusion: } true \mapsto q}$$

The proof of this inference rule is almost identical to that of SF-UNITY. The modified rule allows the ranking function  $M$  to be chosen such that its values may increase as long as  $p$  holds. In contrast, SF-UNITY disallows this behavior; however, this does not prevent SF-UNITY from being relatively complete<sup>13</sup>.

### 5.3.3 Specialized Rules

The inference rule SF-UNITY covers three useful special cases:

$$\frac{p \mapsto q}{\text{Hypothesis: } true \mapsto p \quad \text{Conclusion: } true \mapsto q}$$

Define a function  $M$  from program states to  $\{0, 1\}$  such that  $M = 1$  if and only if  $q$  is true.

---

<sup>13</sup>The completeness proof of B-REAC in [Man91] (page 121) constructs a ranking function such that a program state, where  $p$  holds while  $q$  does not, is mapped to a maximal (not maximum) value in the range of  $M$ . So, when  $p$  is true while  $q$  is not, the value of  $M$  cannot increase unless  $q$  becomes true.

(B  $\Rightarrow$  A) Our strategy consists of two steps: The first is to show that there exists a relatively complete inference rule in MP, say  $R$ , for  $\Box\Diamond p \Rightarrow \Box\Diamond q$  such that each premise of  $R$  has an equivalent unconditional property in UNITY. The second step is to show that the equivalent properties indeed allow us to deduce “Hypothesis:  $true \mapsto p$  Conclusion:  $true \mapsto q$ ”. Consequently, for any property of the form “ $\Box\Diamond p \Rightarrow \Box\Diamond q$ ” that is provable in MP, we can find a set of unconditional properties in UNITY which can be used to deduce “Hypothesis:  $true \mapsto p$  Conclusion:  $true \mapsto q$ ”.

The following inference rule is defined for MP:

B-REAC

$$\frac{\begin{array}{l} \Box(r \Rightarrow (\varphi \vee q)) \\ \{\varphi \wedge (M = m)\}F\{(\varphi \wedge (M \preceq m)) \vee q\} \\ \Box[(p \wedge \varphi \wedge (M = m)) \Rightarrow \Diamond((M \prec m) \vee q)] \end{array}}{\Box[(r \wedge \Box\Diamond p) \Rightarrow \Diamond q]}$$

where  $\varphi$  is an auxiliary predicate and  $M$  is a ranking function from program states to a well-founded set under relation  $\prec$ <sup>10</sup>. B-REAC is sound and relatively complete for deducing properties of the form “ $\Box[(r \wedge \Box\Diamond p) \Rightarrow \Diamond q]$ ”. We simply let  $r$  be  $true$  to obtain a sound and relatively complete inference rule for  $\Box(\Box\Diamond p \Rightarrow \Diamond q)$ , which is equivalent to  $\Box\Diamond p \Rightarrow \Box\Diamond q$ . We can rewrite the second premise as  $\{\varphi \wedge (M \preceq m)\}F\{(\varphi \wedge (M \preceq m)) \vee q\}$  due to the following results:

$$\begin{array}{l} \{\varphi \wedge (M = k)\}F\{(\varphi \wedge (M \preceq k)) \vee q\} \quad , \text{ given.} \\ \{\varphi \wedge (M \preceq m)\}F\{(\varphi \wedge (M \preceq m)) \vee q\} \quad , \text{ disjunction over all } k, k \preceq m. \end{array}$$

$$\begin{array}{l} \{\varphi \wedge (M \preceq m)\}F\{(\varphi \wedge (M \preceq m)) \vee q\} \quad , \text{ given.} \\ \{\varphi \wedge (M = m)\}F\{(\varphi \wedge (M \preceq m)) \vee q\} \quad , \text{ strengthening the precondition.} \end{array}$$

The specialized inference rule is as follows:

$$\frac{\begin{array}{l} \Box(\varphi \vee q) \\ \{\varphi \wedge (M \preceq m)\}F\{(\varphi \wedge (M \preceq m)) \vee q\} \\ \Box[(p \wedge \varphi \wedge (M = m)) \Rightarrow \Diamond((M \prec m) \vee q)] \end{array}}{\Box\Diamond p \Rightarrow \Box\Diamond q}$$

Although the first and third premises have equivalent expressions in UNITY, the second does not. We seek to restate the second premise in the form “ $\{SI \wedge a \wedge \neg b\}F\{a \vee b\}$ ”. The completeness proof of B-REAC (Corollary 7.1 in [Man91]),

<sup>10</sup>The domain of  $M$  in general has to be finite sequences of states. As we will restrict  $p, q, r$  to be predicates on program states, it suffices for  $M$  to be a function with program states as its domain [Man91].

with  $r$  replaced by  $true$ , shows that it suffices to choose  $\varphi$  such that  $\varphi$  holds at a program state if and only if the state is reachable and it satisfies  $\neg q$ . i.e.  $\varphi = (SI \wedge \neg q)$ <sup>11</sup>. We can substitute  $SI \wedge \neg q$  for  $\varphi$  in each premise of the inference rule while preserving its soundness and relative completeness<sup>12</sup>. After substitution, the inference rule becomes as follows:

$$\frac{\begin{array}{l} \Box((SI \wedge \neg q) \vee q) \\ \{(M \preceq m) \wedge SI \wedge \neg q\} F \{((M \preceq m) \wedge SI \wedge \neg q) \vee q\} \\ \Box[(p \wedge SI \wedge \neg q \wedge (M = m)) \Rightarrow \Diamond((M \prec m) \vee q)] \end{array}}{\Box \Diamond p \Rightarrow \Box \Diamond q}$$

Each of the premises can be rewritten such that it corresponds to a UNITY property, as shown below:

$$\begin{aligned} & \Box((SI \wedge \neg q) \vee q) \quad , \text{ the first premise.} \\ = & \Box(SI \vee q) \quad , \text{ predicate calculus.} \\ \cong & \text{invariant } SI \vee q \quad , \text{ Lemma 16.} \\ = & SI \Rightarrow SI \vee q \quad , \text{ the definition of invariant.} \\ = & true \quad , \text{ predicate calculus.} \end{aligned}$$

The first premise can be dropped.

$$\begin{aligned} & \{(M \preceq m) \wedge SI \wedge \neg q\} F \{((M \preceq m) \wedge SI \wedge \neg q) \vee q\} \\ & \quad , \text{ the second premise.} \\ = & \{SI \wedge (M \preceq m) \wedge SI \wedge \neg q\} F \{((M \preceq m) \wedge SI) \vee q\} \\ & \quad , \text{ predicate calculus.} \\ = & (M \preceq m) \wedge SI \text{ unless } q \\ & \quad , \text{ definition of unless.} \\ = & M \preceq m \text{ unless } q \\ & \quad , \text{ substitution theorem.} \end{aligned}$$

$$\begin{aligned} & \Box[(p \wedge SI \wedge \neg q \wedge (M = m)) \Rightarrow \Diamond((M \prec m) \vee q)] \\ & \quad , \text{ the third premise.} \end{aligned}$$

<sup>11</sup>The very definition of  $\varphi$  chosen in the proof is as follows:  $\varphi$  is true at a program state  $s$  if and only if there exists a reachable state  $s'$  satisfying  $r$  and a  $q$ -free segment (a finite subsequence of some computation, each state of which satisfies  $\neg q$ ) leading from  $s'$  to  $s$ . As we have taken  $r$  to be true, the definition collapses down to  $SI \wedge \neg q$ .

<sup>12</sup>A consequence of replacing  $\varphi$  by  $SI \wedge \neg q$  is that the choice of the ranking function will be more restricted.

ity of an inference rule for deducing strong fairness properties in UNITY would facilitate the verification of such an algorithm.

We prove, for a given UNITY program, the equivalence between the provability of conditional property “Hypothesis:  $true \mapsto p$  Conclusion:  $true \mapsto q$ ” in UNITY logic and that of “ $\Box \Diamond p \Rightarrow \Box \Diamond q$ ” in Manna and Pnueli’s temporal logic (referred to as MP in this section) [Man91]. Part of our proof is based on a relatively complete inference rule in MP for deducing strong fairness properties. In the process, we obtain a relatively complete inference rule for proving strong fairness properties in UNITY.

### 5.3.1 UNITY and Temporal Logic

The basic model for programs in MP (which is, again, Manna and Pnueli’s temporal logic) is that of *fair transition system*. As a UNITY program is essentially a fair transition system with weak fairness assumption<sup>5</sup>, MP is suitable for proving properties (about the set of computations) of a UNITY program. MP is sound and relatively complete for deducing properties that can be specified by linear time temporal logic formulas with temporal operators such as  $\Box$ ,  $\Diamond$ , and  $\mathcal{W}$  (weak until)<sup>6</sup>.

We restate the known results about the soundness and relatively completeness results about UNITY in Lemma 16<sup>7</sup>. We say a temporal formula is *valid* (with respect to a program) if it is satisfied by every execution of the program.

**Lemma 16** *For a given UNITY program,*

(A) “ $p$  unless  $q$ ” is provable if and only if “ $\Box(p \Rightarrow (p \mathcal{W} q))$ ” is valid (or, equivalently, is provable in MP),

(B) “invariant  $q$ ” is provable if and only if “ $\Box q$ ” is valid (or provable in MP), and

(C) “ $p \mapsto q$ ” is provable if and only if “ $\Box(p \Rightarrow \Diamond q)$ ” is valid (or provable in MP).

---

<sup>5</sup>Each assignment statement of a UNITY program defines a state transition that is always enabled. The fairness constraint in the UNITY program execution model corresponds to the weak fairness (justice) assumption in the fair transition system.

<sup>6</sup>The definitions of  $\Box$  and  $\Diamond$  can be found in Section 2.2.2. For our purpose,  $\mathcal{W}$  is used only in properties of the form “ $p \Rightarrow (p \mathcal{W} q)$ ” which is exactly the same as “ $p$  Unless  $q$ ” introduced in the same section.

<sup>7</sup>The syntax of the language used in UNITY to express predicates on program states is not given in [Cha88]. As in MP, for the relative completeness to hold, one typically needs to assume that the language is capable of stating facts about the integers and expressing solutions to fixpoint equations. More discussion on this issue may be found in [Man91, Rao91].

The corresponding properties between UNITY and MP in the above lemma can be considered *equivalent*. We write “ $p \mapsto q \cong \Box(p \Rightarrow \Diamond q)$ ” to denote this relationship between the two properties; analogously for the other two pairs of properties. A useful special case of (C) is that  $true \mapsto p \cong \Box \Diamond p$ .

We write “ $\Delta \vdash Q$ ” (for both UNITY and MP) to denote that  $Q$  can be deduced using the list of properties in  $\Delta$  as premises. Given a UNITY property  $P$ , let  $\hat{P}$  denote the equivalent property in MP (in the sense of Lemma 16). As shown in Section 5.2, Lemma 16 and the soundness of the UNITY proof system have the following implications:

Suppose  $P_1, P_2, \dots, P_n, Q$  are *unless*, *invariant*, or  $\mapsto$  properties.  $P_1, P_2, \dots, P_n \vdash Q$  implies that  $\hat{P}_1 \wedge \hat{P}_2 \wedge \dots \wedge \hat{P}_n \Rightarrow \hat{Q}$  is valid. Observe that  $\hat{P}_1 \wedge \hat{P}_2 \wedge \dots \wedge \hat{P}_n \Rightarrow \hat{Q}$  is a temporal formula in MP. As the formula is valid, it follows that  $\vdash \hat{P}_1 \wedge \hat{P}_2 \wedge \dots \wedge \hat{P}_n \Rightarrow \hat{Q}$  due to the relative completeness of MP, which itself implies  $\hat{P}_1, \hat{P}_2, \dots, \hat{P}_n \vdash \hat{Q}$ <sup>8</sup>. The above analysis is summarized by the following lemma:

**Lemma 17** *Let  $P_1, P_2, \dots, P_n, Q$  be properties that are expressed in terms of unless, invariant, and  $\mapsto$ . If  $P_1, P_2, \dots, P_n \vdash Q$  then  $\hat{P}_1, \hat{P}_2, \dots, \hat{P}_n \vdash \hat{Q}$ .*

### 5.3.2 Relative Completeness

**Theorem 16** *For a given UNITY program, the following two statements are equivalent:*

- (A) “Hypothesis:  $true \mapsto p$  Conclusion:  $true \mapsto q$ ” is provable.
- (B) “ $\Box \Diamond p \Rightarrow \Box \Diamond q$ ” is provable in MP (which is equivalent to that  $\Box \Diamond p \Rightarrow \Box \Diamond q$  is valid).

*Proof.* (A  $\Rightarrow$  B)<sup>9</sup> Statement A says that there is a proof in UNITY of  $true \mapsto q$ , where  $true \mapsto p$  is assumed. Note that each property in the proof is unconditional. It is clear that we can extract from the proof a list of properties  $P_1, P_2, \dots, P_n$ , each of which is an *invariant*, *unless*, or  $\mapsto$  property proved from the given program, such that  $true \mapsto p, P_1, P_2, \dots, P_n \vdash true \mapsto q$ . From Lemma 16,  $\hat{P}_1, \hat{P}_2, \dots, \hat{P}_n$  are provable in MP from the given program; and, from Lemma 17, it follows that  $\Box \Diamond p, \hat{P}_1, \hat{P}_2, \dots, \hat{P}_n \vdash \Box \Diamond q$ , which implies  $\hat{P}_1, \hat{P}_2, \dots, \hat{P}_n \vdash \Box \Diamond p \Rightarrow \Box \Diamond q$ .

<sup>8</sup>MP respects the generalized deduction theorem:  $\Delta \vdash P \Rightarrow Q$  if and only if  $\Delta, P \vdash Q$ .

<sup>9</sup>In fact, a much simpler proof can be obtained by applying Theorem 14 and the relative completeness of MP

$p \wedge q \mapsto \neg q \vee q$	, implication theorem on $p \wedge q \Rightarrow true$ .
$p \wedge (M = 1) \mapsto (M < 1) \vee q$	, from the above and definition of $M$ . (1)
$p \wedge \neg q \mapsto q$	, $p \wedge \neg q \Rightarrow p$ and the premise $p \mapsto q$ .
$p \wedge (M = 0) \mapsto (M < 0) \vee q$	, from the above and definition of $M$ .
$p \wedge (M = m) \mapsto (M < m) \vee q$	, from the above and (1). (2)
$(M \leq 1) \text{ unless } q$	, from $true \text{ unless } q$ .
$(M \leq 0) \text{ unless } q$	, from $\neg q \text{ unless } q$ .
$(M \leq m) \text{ unless } q$	, from the above two.
The inference rule is valid	, from the above, (2), and SF-UNITY.

$$\frac{true \mapsto \neg p \vee q, \neg p \text{ unless } q}{\text{Hypothesis: } true \mapsto p \quad \text{Conclusion: } true \mapsto q}$$

For the proof, define a function  $M$  again with range  $\{0, 1\}$  such that  $M = 1$  if and only if  $p$  is *true*. The rest is similar to the previous inference rule. Operationally, the first premise implies that there are infinitely many occurrences of either  $\neg p$  or  $q$ ; only the first case needs to be considered. The hypothesis says that there are infinitely many occurrences of  $p$ . It follows that  $p$  changes from false to true for infinitely many times, which will trigger as many occurrences of  $q$  thanks to the second premise.

$$\frac{p \mapsto \neg p, \neg p \text{ unless } q}{\text{Hypothesis: } true \mapsto p \quad \text{Conclusion: } true \mapsto q}$$

$p \mapsto \neg p \vee q$	, from the first premise and $\neg p \Rightarrow \neg p \vee q$ .
$\neg p \mapsto \neg p \vee q$	, implication theorem on $\neg p \Rightarrow \neg p \vee q$ .
$true \mapsto \neg p \vee q$	, disjunction on the above two.

The validity of the last special case therefore follows from the second special case.

# CHAPTER 6

## UNITY Proof of Fair Algorithms

To illustrate the use of the results from last chapter in the formal specification and verification of reactive systems with strong fairness properties, we give a formal specification of the binary interaction problem with SPF and prove the correctness of a variation of Algorithm A. Other fair algorithms can be proved in a similar way.

### 6.1 Preparation

#### 6.1.1 Modeling Distributed Systems

We model a distributed system with message-passing by a UNITY program in a manner analogous to that of Section 2.2.3:

The UNITY program consists of component programs that are functionally divided into two categories: *processes* which do significant computations and *channels* which simply relay messages. Distinct processes have disjoint sets of variables and so do distinct channels; variables may be shared only between a process and a channel. A sender process may send a message to a receiver process by depositing the message in a message queue shared by the sender and a channel; the channel then delivers the message by removing the message and depositing it in another message queue shared by the channel and the receiver process.

#### 6.1.2 Plausible Inference Rules

Though we are equipped with a relatively complete inference rule SF-UNITY derived in Section 5.3 for proving strong fairness properties, the ranking function in the rule may be very complicated for certain fairness properties. In accordance with the results in Sections 5.2 and 5.3, we shall propose two inference rules to facilitate the proof of fairness properties.

In an usual logic system, given " $P_1 \Rightarrow Q$ " and " $P_2 \Rightarrow Q$ ", one should be able to deduce " $(P_1 \vee P_2) \Rightarrow Q$ ". In particular, from " $\Box \Diamond p \Rightarrow Q$ " and " $\Box \Diamond q \Rightarrow Q$ ",



### 6.2.1 Specification of *USER* (the Given)

This part specifies the behavior of the *USER* program at its interface with the *OS* and also specifies some properties that are guaranteed when *USER* is composed with the *OS*.

For each *user*, the variable *state* is initialized to *active* and *flag* to *null*. An idle *user* may become active only after some interaction in its interaction set is started, i.e. it is *engaged* — (u1). No *user* may start an interaction — (u2). Any started interaction will eventually be terminated (by some of its members) — (u3).

$$\text{idle}_i \text{ unless } (\exists j :: \text{start}^{\{i,j\}}) \text{ in } \text{USER} \quad (\text{u1})$$

$$\text{stable } (\text{flag}_i \neq \{i, j\}) \text{ in } \text{USER} \quad (\text{u2})$$

$$\text{start}^{\{i,j\}} \mapsto \neg \text{start}^{\{i,j\}} \text{ in } \mathcal{P} \quad (\text{u3})$$

Note that (u3) should have been specified as a conditional property with *true* as its hypothesis. From (u2), (u3), and two other properties (pp2) and (o1) to be defined next, one can deduce that

$$\text{engaged}_i \mapsto \neg \text{engaged}_i \text{ in } \mathcal{P} \quad (\text{u4})$$

### 6.2.2 Specification of $\mathcal{P}$ (the Composite)

This part specifies the synchronization, mutual exclusion, progress, and the additional SPF properties that must be provided by the composition of *USER* and *OS*.

The synchronization property requires that only *enabled* interactions can be started — (pp1) and the mutual exclusion property requires that conflicting interactions cannot be started simultaneously — (pp2). The progress property requires that if an interaction  $\{i, j\}$  is *enabled*, either  $p_i$  or  $p_j$  will eventually participate in some interaction — (pp3).

$$\neg \text{start}^{\{i,j\}} \text{ unless } \text{enable}^{\{i,j\}} \text{ in } \mathcal{P} \quad (\text{pp1})$$

$$\text{invariant } (j \neq k) \Rightarrow \neg(\text{start}^{\{i,j\}} \wedge \text{start}^{\{i,k\}}) \text{ in } \mathcal{P} \quad (\text{pp2})$$

$$\text{enable}^{\{i,j\}} \mapsto [\exists k :: \text{start}^{\{i,k\}} \vee \text{start}^{\{j,k\}}] \text{ in } \mathcal{P} \quad (\text{pp3})$$

#### Additional Property

The additional SPF property is specified as follows; recall that SPF subsumes (pp3).

$$\text{SPF} \equiv \Box \Diamond \text{ready}_i \Rightarrow \Box \Diamond \text{engaged}_i, \text{ where } \text{ready}_i \equiv [\exists j :: \text{enable}^{\{i,j\}}]$$

### 6.2.3 Constraints on OS (the Solution)

The only shared variables between  $user_i$  and  $os_i$  are  $state_i$  and  $flag_i$ . For each  $os$ ,  $state$  is initialized to *active* and  $flag$  to *null* (consistent with the initialization in *USER*). An  $os$  may not terminate an interaction — (o1). Moreover, an  $os$  may not change the state of a  $user$  — (o2.1) and (o2.2).

**stable** ( $flag_i = \{i, j\}$ ) in *OS* (o1)

**stable** *active<sub>i</sub>* in *OS* (o2.1)

**stable** *idle<sub>i</sub>* in *OS* (o2.2)

### 6.2.4 Simple Refinement of the Specification

According to the problem specification, an interaction, say  $\{i, j\}$ , is started when  $flag_i$  is assigned the value  $\{i, j\}$  by  $p_i$  or  $flag_j$  assigned  $\{i, j\}$  by  $p_j$ . However, neither  $p_i$  nor  $p_j$  is allowed to do so unless both processes are idle (otherwise (pp1) may be violated) and not engaged (otherwise (pp2) may be violated), which involves determination of the local states (the values of  $flag$ 's, in particular) of other processes that can be communicating partners of  $p_i$  or  $p_j$ . Since inspecting of the values of the local variables of other processes is, by definition, not allowed in the program text of a process, the states of other processes must be reflected through the values of the local variables of  $p_i$  or  $p_j$ , i.e. by some local predicate of  $p_i$  or  $p_j$ .

Hence, we postulate a local predicate  $L_i^{\{i,j\}}$  of  $os_i$  for interaction  $\{i, j\}$  and propose the following refinement of the specification, where  $L^{\{i,j\}}$  abbreviates  $L_i^{\{i,j\}} \vee L_j^{\{i,j\}}$ .

**invariant**  $L_i^{\{i,j\}} \Rightarrow enable^{\{i,j\}} \wedge \neg engaged_i \wedge \neg engaged_j$  (kp1)

$(flag_i \neq \{i, j\})$  **unless**  $L_i^{\{i,j\}}$  (kp2)

$start^{\{i,j\}} \wedge \neg L^{\{j,k\}}$  **unless**  $\neg start^{\{i,j\}} \wedge \neg L^{\{j,k\}}$ , where  $k \neq i$  (kp3)

$L_i^{\{i,j\}}$  **ensures**  $(flag_i = \{i, j\})$  (kp4)

$\Box \Diamond ready_i \Rightarrow \Box \Diamond [\exists j :: L^{\{i,j\}}]$  (kp5)

**Theorem 18** (kp1)–(kp5) *imply* (pp1), (pp2), (pp3), and SPF.

*Proof.* (kp1) and (kp2) *imply* (pp1), by consequence weakening for *unless*.

(kp4) and (kp5) *imply* SPF which subsumes (pp3), by disjunction theorem on (kp4), the fact that  $p \mapsto q$  *implies*  $\Box \Diamond p \Rightarrow \Box \Diamond q$ , and that  $\Box \Diamond p \Rightarrow \Box \Diamond q$  and  $\Box \Diamond q \Rightarrow \Box \Diamond r$  *imply*  $\Box \Diamond p \Rightarrow \Box \Diamond r$ .

It remains to show that (kp1), (kp2), and (kp3) imply (pp2); the proof utilizes a property that has so far not been explicitly stated as a UNITY property:

$\neg start^{i,j} \wedge \neg start^{k,l}$  unless  $(start^{i,j} \wedge \neg start^{k,l}) \vee (\neg start^{i,j} \wedge start^{k,l})$ ,  
 where  $\{i, j\} \neq \{k, l\}$  (kp6)

(kp6) asserts that at most one interaction is started in each computation step due to the interleaving model and that each process has one flag variable.

$\neg start^{j,k}$  unless  $L^{j,k}$

, simple conjunction on (kp2) and rewriting (change of variables).

$start^{i,j} \wedge \neg L^{j,k} \wedge \neg start^{j,k}$  unless  $\neg start^{i,j} \wedge \neg L^{j,k} \wedge \neg start^{j,k}$

, conjunction on the above and (kp3).

**invariant**  $start^{i,j} \Rightarrow \neg L^{j,k}$

, from (kp1) and  $start^{i,j} \Rightarrow engaged_j$ .

$start^{i,j} \wedge \neg start^{j,k}$  unless  $\neg start^{i,j} \wedge \neg start^{j,k}$

, substitution axiom on the above two and consequence weakening.

$\neg start^{i,j} \wedge start^{j,k}$  unless  $\neg start^{i,j} \wedge \neg start^{j,k}$

, similar to the proof of the above.

$(\neg start^{i,j} \wedge \neg start^{j,k}) \vee (start^{i,j} \wedge \neg start^{j,k}) \vee (\neg start^{i,j} \wedge start^{j,k})$

unless *false*

, disjunction on (kp6) and the above two.

$\neg(start^{i,j} \wedge start^{j,k})$  unless *false*

, rewriting of the above.

**invariant**  $\neg(start^{i,j} \wedge start^{j,k})$

, from the above and the initial condition.

*End of Proof.*

## 6.3 Verification of a Solution

### 6.3.1 The OS

We present the program of an os using Algorithm A (Section 3.1) with a simpler token selection strategy: a process always selects a token from the head of its token queue. For simplicity,  $os_i$  has a single input queue  $input_i$  for all messages sent from its neighbors; some other  $os$  sends a message to  $os_i$  by appending the message to  $input_i$ .  $os_i$  sends messages to other  $os$ 's in an analogous manner. The input queues shared among  $os$ 's can be further "decoupled" such that any pair of  $os$ 's do not share variables directly as required by the problem specification.

Variables of  $os_i$ :

(Variable subscripts are omitted, as is the declaration for some variables whose

purpose is obvious.)

**flag:** shared with **user**; see problem specification.

**token\_q:** a FIFO queue of tokens. A token is an unordered pair of process id's. Initially, each token is arbitrarily assigned to one of the processes named in the token.

**ino** $\{i,j\}$ : id of interaction  $\{i,j\}$ . Conflicting interactions are assigned different id's.

**pend:** id of the process requested for interaction. Initially *null*.

**delay:** id of the process whose request is delayed. Initially *null*.

**partner:** set to  $j$  if the process is about to participate in interaction  $\{i,j\}$ . Initially *null*.

**isstarter:** set to *true* when the process starts an interaction; reset to *false* when the termination of the interaction is detected. Initially *false*.

Statements of **os** $_i$ :

((a) A boolean variable is also used as a predicate, e.g. **partner** is *true* if and only if **partner**  $\neq$  *null*; analogously for **pend** and **delay**. (b) "Send(*request* $\{i,j\}$ )" abbreviates "Enqueue(**input** $_j$ , *request* $\{i,j\}$ )" and "Receive(*request* $\{i,j\}$ )  $\wedge \dots \rightarrow \dots$ " abbreviates " $(\text{head}(\text{input}_i) = \text{request}\{i,j\}) \wedge \dots \rightarrow \text{input}_i := \text{tail}(\text{input}_i); \dots$ "; similarly for messages *yes*, *no*, and *done*.)

```
R1: /* Requesting an interaction */
    idle  $\wedge$   $\neg$ partner  $\wedge$   $\neg$ pend  $\wedge$  (head(token_q) = {i,j})
     $\rightarrow$  token_q := tail(token_q);
    pend := j; Send(request $\{i,j\}$ );

R2: /* Refusing a request */
    Receive(request $\{i,j\}$ )  $\wedge$  ( $\neg$ idle  $\vee$  partner  $\vee$  (pend  $\wedge$  (ino[i,j] < ino[i, pend]))  $\vee$ 
    delay)
     $\rightarrow$  Enqueue(token_q, {i,j});
    Send(no $\{i,j\}$ );

R3: /* Delaying the reply to a request */
    Receive(request $\{i,j\}$ )  $\wedge$  (pend  $\wedge$  (ino[i,j]  $\geq$  ino[i, pend]))  $\wedge$   $\neg$ delay
     $\rightarrow$  delay := j; /* The token is enqueued is a later step. */

R4: /* Accepting an incoming request or a delayed request */
    R4.1: Receive(request $\{i,j\}$ )  $\wedge$  idle  $\wedge$   $\neg$ partner  $\wedge$   $\neg$ pend
         $\rightarrow$  Enqueue(token_q, {i,j});
        partner := j; Send(yes $\{i,j\}$ );
    R4.2: Receive(no $\{i,\text{pend}\}$ )  $\wedge$  delay  $\rightarrow$  pend := null;
        partner := delay; Send(yes $\{i,\text{delay}\}$ );
        Enqueue(token_q, {i,delay});
```

delay := null;

R5: /\* Relinquishing a request \*/  
 Receive( $no^{i,pend}$ )  $\wedge$   $\neg$ delay  $\rightarrow$  pend := null;

R6: /\* Starting an interaction \*/  
 R6.1: Receive( $yes^{i,pend}$ )  $\wedge$   $\neg$ delay  $\rightarrow$  isstarter := true; flag := {i,pend};  
           partner := pend; pend := null;  
 R6.2: Receive( $yes^{i,pend}$ )  $\wedge$  delay  $\rightarrow$  isstarter := true; flag := {i,pend};  
           partner := pend; pend := null;  
           Enqueue(token\_q, {i,delay});  
           Send( $no^{i,delay}$ ); delay := null;

R7: /\* Detecting the termination of an interaction \*/  
 R7.1: isstarter  $\wedge$   $\neg$ flag  $\rightarrow$  isstarter := false;  
           Send( $done^{i,partner}$ ); partner := null;  
 R7.2: Receive( $done^{i,partner}$ )  $\rightarrow$  partner := null;

### 6.3.2 Correctness of the OS

We shall omit the proofs of some properties that are straightforward. In particular, we assume the validity of such invariants as “either there is no instance of  $request^{i,j}$  in input<sub>*j*</sub>, or there is exactly one instance of  $request^{i,j}$  in input<sub>*j*</sub>”. We write “ $request^{i,j} \in \text{input}_j$ ” to denote the latter case; this convention applies to other messages and also to tokens in a token queue.

For brevity, the arithmetic addition “+” will sometimes be applied to a series of predicates, in which case we identify the boolean value “false” with integer “0” and “true” with “1”; e.g.  $a = b + c$  means that  $a$  is true if and only if exactly one of  $b$  and  $c$  is true. We shall also make assertions about the “position” of a message or token in a queue, the head of the queue being position one. Again, we omit the phrase “in  $\mathcal{P}$ ”, when stating properties of  $\mathcal{P}$ .

It is clear that “( $head(\text{input}_i) = yes^{i,j}$ )” should play the role of  $L_i^{i,j}$ , which was assumed to be a local predicate of  $p_i$  indicating that  $p_i$  knows interaction  $\{i,j\}$  is enabled and neither  $p_i$  itself nor  $p_j$  is participating in any interaction. We thus identify the two predicates and replace  $L_i^{i,j}$  in (kp1)–(kp5) by “( $head(\text{input}_i) = yes^{i,j}$ )”.

#### 6.3.2.1 Safety Properties

**Theorem 19** ( $\text{flag}_i \neq \{i,j\}$ ) unless ( $head(\text{input}_i) = yes^{i,j}$ ). (kp2)

*Proof.*

( $\text{flag}_i \neq \{i, j\}$ ) unless ( $\text{head}(\text{input}_i) = \text{yes}^{i,j}$ ) in *OS*  
, from the definition of *unless* and the program of *OS*.

stable ( $\text{flag}_i \neq \{i, j\}$ ) in *USER*  
, from (u2).

The theorem follows

, union theorem on the above two.

*End of Proof.*

**Theorem 20**  $\text{start}^{i,j} \wedge \neg(\text{head}(\text{input}_j) = \text{yes}^{j,k} \vee \text{head}(\text{input}_k) = \text{yes}^{j,k})$   
unless  $\neg\text{start}^{i,j} \wedge \neg(\text{head}(\text{input}_j) = \text{yes}^{j,k} \vee \text{head}(\text{input}_k) = \text{yes}^{j,k})$ . where  
 $k \neq i$ . (kp3)

*Proof.* Analogous to the proof of the preceding theorem only with more invariants  
involved.

*End of Proof.*

We next prove a series of lemmas leading to Theorem 21 which states (kp1).

**Lemma 18** *The following predicates are invariants in  $\mathcal{P}$ :*

(a) ( $\text{pend}_i = j$ ) = ( $\text{request}^{i,j} \in \text{input}_j$ ) + ( $\text{delay}_j = i$ ) + ( $\text{yes}^{i,j} \in \text{input}_i$ ) +  
( $\text{no}^{i,j} \in \text{input}_i$ ) (when  $p_i$  has a pending request to  $p_j$ , either the request has not  
been received or the request has been delayed or a reply has been sent by  $p_j$ ),  
(b)  $\text{pend}_i \Rightarrow \neg\text{partner}_i$ , (c)  $\text{delay}_i \Rightarrow \text{pend}_i$ , and (d)  $\text{isstarter}_i \Rightarrow \text{partner}_i$ .

*Proof.* The proof of each invariant follows exactly the same line. We show a  
proof of (b).

initially  $\text{pend}_i \Rightarrow \neg\text{partner}_i$  in *OS*  
, from the program of *OS*.

stable  $\text{pend}_i \Rightarrow \neg\text{partner}_i$  in *OS*  
, from the definition of *stable* and the program of *OS*.

invariant  $\text{pend}_i \Rightarrow \neg\text{partner}_i$  in *OS*  
, from the definition of *invariant* and the above two.

stable  $\text{pend}_i \Rightarrow \neg\text{partner}_i$  in *USER*  
, since the predicate is local to *OS*.

invariant  $\text{pend}_i \Rightarrow \neg\text{partner}_i$   
, union theorem on the above two. *End of Proof.*

**Lemma 19** (a) invariant  $((\text{partner}_j = i) \wedge \neg\text{isstarter}_j) = (\text{yes}^{i,j} \in \text{input}_i) +$   
 $((\text{partner}_i = j) \wedge \text{isstarter}_i) + (\text{done}^{i,j} \in \text{input}_j)$ .

*As a consequence of (a) and Lemma 18:*

(b) invariant  $(\text{yes}^{i,j} \in \text{input}_i) \Rightarrow \text{pend}_i \wedge \neg\text{partner}_i \wedge ((\text{partner}_j = i) \wedge \neg\text{isstarter}_j)$ .

(c) invariant  $(\text{partner}_i = j) \wedge \text{isstarter}_i \Rightarrow (\text{partner}_j = i) \wedge \neg\text{isstarter}_j$ .

(d) invariant  $(\text{done}^{i,j} \in \text{input}_j) \Rightarrow ((\text{partner}_i \neq j) \vee \neg\text{isstarter}_i) \wedge ((\text{partner}_j =$   
 $i) \wedge \neg\text{isstarter}_i)$ .

*Proof.* Case (a) can be proved by an analogous argument as in Lemma 18. The conjunction of the predicate in (a) and all the predicates in Lemma 18 is an invariant in  $\mathcal{P}$ . Other invariants of the lemma follow from rewriting of the preceding conjunction and the substitution axiom, by substituting *true* for the predicates in Lemma 18. *End of Proof.*

**Lemma 20** invariant  $(\text{flag}_i = \{i, j\}) \Rightarrow (\text{partner}_i = j) \wedge (\text{partner}_j = i)$ .

*Proof.*

stable  $(\text{flag}_i \neq \{i, j\})$  in *USER*  
, from (u2).

stable  $(\text{partner}_i = j) \wedge \text{isstarter}_i$  in *USER*  
, since the predicate is local to *OS*.

stable  $(\text{flag}_i = \{i, j\}) \Rightarrow (\text{partner}_i = j) \wedge \text{isstarter}_i$  in *USER*  
, disjunction on the above two.

invariant  $(\text{flag}_i = \{i, j\}) \Rightarrow (\text{partner}_i = j) \wedge \text{isstarter}_i$  in *OS*  
, by an analogous argument as in Lemma 19 while using the invariants of Lemmas 18 and 19 in the application of the substitution axiom.

invariant  $(\text{flag}_i = \{i, j\}) \Rightarrow (\text{partner}_i = j) \wedge \text{isstarter}_i$  in  $\mathcal{P}$   
, union theorem the above two.

The lemma follows

, from the above and Lemma 19(c).

*End of Proof.*

**Lemma 21** invariant  $\text{pend}_i \Rightarrow \text{idle}_i \wedge \neg \text{partner}_i$ .

*Proof.*

*idle*<sub>*i*</sub> unless *partner*<sub>*i*</sub> in *USER*  
, consequence weakening on (u1) and Lemma 20.

$\neg \text{partner}_i$  unless *false* in *USER*  
,  $\neg \text{partner}_i$  is local to *OS*.

*idle*<sub>*i*</sub>  $\wedge \neg \text{partner}_i$  unless *false* in *USER*  
, conjunction on the above two.

$\neg \text{pend}_i$  unless *false* in *USER*.  
,  $\neg \text{pend}_i$  is local to *OS*.

$\neg \text{pend}_i \vee (\text{idle}_i \wedge \neg \text{partner}_i)$  unless *false* in *USER*,  
i.e. stable  $\text{pend}_i \Rightarrow \text{idle}_i \wedge \neg \text{partner}_i$  in *USER*  
, disjunction on the above two.

invariant  $\text{pend}_i \Rightarrow \text{idle}_i \wedge \neg \text{partner}_i$  in *OS*

, similar to Lemma 18.

The lemma follows

, union theorem on the above two.

*End of Proof.*

**Lemma 22 invariant**  $(yes^{i,j} \in input_i) \Rightarrow idle_j \wedge (partner_j = i) \wedge \neg partner_i$ .

*Proof.* Similar to Lemma 21.

*End of Proof.*

**Theorem 21 invariant**  $(head(input_i) = yes^{i,j}) \Rightarrow enable^{i,j} \wedge \neg engaged_i \wedge \neg engaged_j$ . (kp1)

*Proof.*

invariant  $(yes^{i,j} \in input_i) \Rightarrow (idle_i \wedge \neg partner_i) \wedge (idle_j \wedge (partner_j = i))$   
, from Lemmas 19(b), 21, and 22.

invariant  $(head(input_i) = yes^{i,j}) \Rightarrow (idle_i \wedge \neg partner_i) \wedge (idle_j \wedge (partner_j = i))$   
, from the above and  $(head(input_i) = yes^{i,j}) \Rightarrow (yes^{i,j} \in input_i)$ .

invariant  $(head(input_i) = yes^{i,j}) \Rightarrow (idle_i \wedge idle_j) \wedge (\neg partner_i) \wedge (partner_j = i)$   
, rewriting of the above.

invariant  $(head(input_i) = yes^{i,j}) \Rightarrow enable^{i,j} \wedge \neg[\exists k :: flag_i = \{i, k\} \vee flag_k = \{i, k\}] \wedge \neg[\exists k :: flag_j = \{j, k\} \vee flag_k = \{j, k\}]$   
, from the above and Lemma 20.

The theorem follows

, from the above and the definitions of  $engaged_i$  and  $engaged_j$ .

*End of Proof.*

### 6.3.2.2 Liveness Properties

**Theorem 22**  $(head(input_i) = yes^{i,j})$  ensures  $(flag_i = \{i, j\})$ . (kp4)

*Proof.*

$((partner_j = i) \wedge \neg isstarter_j) \wedge (head(input_i) = yes^{i,j})$  ensures  $(flag_i = \{i, j\})$   
in  $OS$

, from the definition of *ensures* and the program of  $OS$ .

stable  $((partner_j = i) \wedge \neg isstarter_j) \wedge (head(input_i) = yes^{i,j})$  in  $USER$

, since the predicate is local to  $OS$ .

$((partner_j = i) \wedge \neg isstarter_j) \wedge (head(input_i) = yes^{i,j})$  ensures  $(flag_i = \{i, j\})$   
, union theorem on the above two.



$(\text{head}(\text{input}_i) = \text{yes}^{\{i,j\}})$  ensures  $(\text{flag}_i = \{i,j\})$

, substitution axiom on the above and Lemma 19(b).

*End of Proof.*

We now prove a series of lemmas leading to Theorem 23 which states (kp5).

**Lemma 23** (a) invariant  $(\{i,j\} \in \text{token\_q}_i) + (\text{request}^{\{i,j\}} \in \text{input}_j) + (\{i,j\} = \{\text{delay}_j, j\}) + (\{i,j\} \in \text{token\_q}_j) + (\text{request}^{\{i,j\}} \in \text{input}_i) + (\{i,j\} = \{i, \text{delay}_i\}) = 1$

(b)  $(\{i,j\} \in \text{token\_q}_i)$  unless  $(\text{request}^{\{i,j\}} \in \text{input}_j)$ .

(c)  $(\text{request}^{\{i,j\}} \in \text{input}_i)$  unless  $(\{i,j\} \in \text{token\_q}_i) \vee (\{i,j\} = \{i, \text{delay}_i\})$ .

*Proof.* Similar to Lemma 18.

*End of Proof.*

**Lemma 24** (a)  $(\text{msg} \in \text{input}_i) \mapsto (\text{head}(\text{input}_i) = \text{msg})$  in  $\mathcal{P}$ , where  $\text{msg}$  denotes an instance of  $\text{request}^{\{i,j\}}$ ,  $\text{yes}^{\{i,j\}}$ ,  $\text{no}^{\{i,j\}}$ , or  $\text{done}^{\{i,j\}}$ .

(b)  $(\text{request}^{\{i,j\}} \in \text{input}_j) \mapsto (\{i,j\} \in \text{token\_q}_j) \vee (\{i,j\} = \{i, \text{delay}_i\})$ .

(c)  $(\text{pend}_i = j) \mapsto (\text{yes}^{\{i,j\}} \in \text{input}_i) \vee (\text{no}^{\{i,j\}} \in \text{input}_i)$ .

(d)  $\text{pend}_i \mapsto \neg \text{pend}_i$ .

(e)  $\text{delay}_i \mapsto [\exists j :: \text{yes}^{\{i,j\}} \in \text{input}_i \vee \text{yes}^{\{i,j\}} \in \text{input}_j]$ .

(f)  $\text{partner}_i \mapsto \neg \text{partner}_i \wedge \neg \text{pend}_i$ .

*Proof.* We show only the proof of (a).

Let  $\text{pos}(\text{input}_i, \text{msg})$  denote the position of  $\text{msg}$  in  $\text{input}_i$ ;  $\text{pos}(\text{input}_i, \text{msg})$  is some large enough constant if  $\text{msg} \notin \text{input}_i$ .

$(\text{msg} \in \text{input}_i) \wedge (\text{pos}(\text{input}_i, \text{msg}) = m)$  ensures

$((\text{msg} \in \text{input}_i) \wedge (\text{pos}(\text{input}_i, \text{msg}) < m)) \vee (\text{head}(\text{input}_i) = \text{msg})$

, by a similar argument as in Theorem 22 while using invariants from Lemmas 18, 19, and 23 in the applications of substitution axiom.

$(\text{msg} \in \text{input}_i) \mapsto (\text{head}(\text{input}_i) = \text{msg})$

, induction on the above.

*End of Proof.*

**Lemma 25**  $\diamond \square(\{i,j\} \notin \text{token\_q}_i) \Rightarrow \diamond \square(\{i,j\} \in \text{token\_q}_j)$ .

*Proof.* The lemma says that if from some point on in a computation token  $\{i,j\}$  is never stored in  $\text{token\_q}_i$ , then it will eventually be kept in  $\text{token\_q}_j$  forever. It can be proved from Lemmas 23, 24(b), 24(c), and 18(c). *End of Proof.*

**Lemma 26**  $\Diamond \Box [\forall j :: \neg(\text{flag}_i = \{i, j\} \vee \text{flag}_j = \{i, j\})] \Rightarrow \Diamond \Box (\neg \text{partner}_i \wedge \neg \text{delay}_i)$ .

*Proof.*

$\neg \text{partner}_i$  unless  $[\exists j :: \text{yes}^{(i,j)} \in \text{input}_j \vee \text{flag}_i = \{i, j\}]$ .  
, similar to Lemma 21.

$\Box \Diamond \text{partner}_i \Rightarrow \Box \Diamond [\exists j :: \text{yes}^{(i,j)} \in \text{input}_j \vee \text{flag}_i = \{i, j\}]$ .  
, special case of SF-UNITY on the above and Lemma 24(f).

$[\exists j :: \text{yes}^{(i,j)} \in \text{input}_j \vee \text{flag}_i = \{i, j\}] \mapsto [\exists j :: \text{flag}_i = \{i, j\} \vee \text{flag}_j = \{i, j\}]$ .  
, from Theorem 21 and Lemma 24(a).

$\Box \Diamond \text{partner}_i \Rightarrow \Box \Diamond [\exists j :: \text{flag}_i = \{i, j\} \vee \text{flag}_j = \{i, j\}]$ .  
, from the above two.

$\Diamond \Box [\forall j :: \neg(\text{flag}_i = \{i, j\} \vee \text{flag}_j = \{i, j\})] \Rightarrow \Diamond \Box (\neg \text{partner}_i)$   
, rewriting of the above.

$\Diamond \Box [\forall j :: \neg(\text{flag}_i = \{i, j\} \vee \text{flag}_j = \{i, j\})] \Rightarrow \Diamond \Box (\neg \text{delay}_i)$   
, from Lemma 24(e).

The lemma follows

, from the above two.

*End of Proof.*

**Lemma 27**  $\Diamond \Box [\forall j :: \neg(\text{flag}_i = \{i, j\} \vee \text{flag}_j = \{i, j\})] \wedge \Box \Diamond \text{idle}_i \Rightarrow \Diamond \Box \text{idle}_i$ .

*Proof.* The lemma follows from (u1).

*End of Proof.*

**Lemma 28**  $\Box \Diamond \text{idle}_i \Rightarrow \Box \Diamond (\text{idle}_i \wedge \neg \text{partner}_i \wedge \neg \text{pend}_i)$ .

*Proof.*

$\Box \Diamond \text{idle}_i \Rightarrow \Box \Diamond ((\text{idle}_i \wedge \neg \text{partner}_i \wedge \neg \text{pend}_i) \vee (\text{idle}_i \wedge \neg \text{partner}_i \wedge \text{pend}_i) \vee (\text{idle}_i \wedge \text{partner}_i))$   
, case analysis. (1)

$(\text{idle}_i \wedge \neg \text{partner}_i \wedge \text{pend}_i)$  unless  $(\text{idle}_i \wedge \neg \text{pend}_i)$   
, similar to Lemma 21.

$(\text{idle}_i \wedge \neg \text{partner}_i \wedge \text{pend}_i) \mapsto (\text{idle}_i \wedge \neg \text{pend}_i)$   
, PSP theorem on the above and Lemma 24(d).

$(\text{idle}_i \wedge \neg \text{partner}_i \wedge \text{pend}_i) \mapsto (\text{idle}_i \wedge \neg \text{partner}_i \wedge \neg \text{pend}_i) \vee (\text{idle}_i \wedge \text{partner}_i \wedge \neg \text{pend}_i)$

, rewriting of the above.

$\Box \Diamond (\text{idle}_i \wedge \neg \text{partner}_i \wedge \text{pend}_i) \Rightarrow \Box \Diamond ((\text{idle}_i \wedge \neg \text{partner}_i \wedge \neg \text{pend}_i) \vee (\text{idle}_i \wedge \text{partner}_i \wedge \neg \text{pend}_i))$   
, special case of SF-UNITY on the above. (2)

$idle_i \wedge partner_i \mapsto \neg partner_i \wedge \neg pend_i$   
 -, from Lemma 24(f).

$\Box\Diamond(idle_i \wedge partner_i) \Rightarrow \Box\Diamond((idle_i \wedge \neg partner_i \wedge \neg pend_i) \vee$   
 $(\neg idle_i \wedge \neg partner_i \wedge \neg pend_i))$ .

, special case of SF-UNITY on the above and case analysis.

$\Box\Diamond idle_i \Rightarrow \Box\Diamond((idle_i \wedge \neg partner_i \wedge \neg pend_i) \vee (\neg idle_i \wedge \neg partner_i \wedge \neg pend_i))$   
 , from (1), (2), and the above.

It remains to show that  $\Box\Diamond idle_i \wedge \Box\Diamond(\neg idle_i \wedge \neg partner_i \wedge \neg pend_i) \Rightarrow$   
 $\Box\Diamond(idle_i \wedge \neg partner_i \wedge \neg pend_i)$ .

$(\neg idle_i \wedge \neg partner_i \wedge \neg pend_i)$  unless  $(idle_i \wedge \neg partner_i \wedge \neg pend_i)$   
 , similar to Lemma 21.

$\Box\Diamond idle_i \wedge \Box\Diamond(\neg idle_i \wedge \neg partner_i \wedge \neg pend_i) \Rightarrow \Box\Diamond(idle_i \wedge \neg partner_i \wedge \neg pend_i)$   
 , from the above.

(From  $\neg p \wedge q$  unless  $p \wedge q$  it is easy to deduce that  $\Box\Diamond(\neg p \wedge q) \wedge \Box\Diamond p \Rightarrow$   
 $\Box\Diamond(p \wedge q)$ .)

*End of Proof.*

**Lemma 29**  $\Diamond\Box(idle_i \wedge \neg partner_i \wedge \neg delay_i) \Rightarrow \Diamond\Box empty(token\_q_i)$ .

*Proof.* We prove  $\Box\Diamond\neg((idle_i \wedge \neg partner_i \wedge \neg delay_i) \wedge empty(token\_q_i)) \Rightarrow \Box\Diamond\neg(idle_i$   
 $\wedge \neg partner_i \wedge \neg delay_i)$ , which is equivalent to  $\Diamond\Box(idle_i \wedge \neg partner_i \wedge \neg delay_i) \Rightarrow$   
 $\Diamond\Box((idle_i \wedge \neg partner_i \wedge \neg delay_i) \wedge empty(token\_q_i))$ , and hence the lemma.

Let  $maxid_i$  be the maximum token id among the tokens in  $token\_q_i$  plus the token  $\{i, pend_i\}$  if  $pend_i$  is not *null*.  $maxid_i$  is some small enough constant, say one less than the smallest token id among all tokens, if  $token\_q_i$  is empty and  $pend_i$  is *null*.

Let  $maxpos_i$  take on the value of the position of the token with id  $maxid_i$  in the imaginary queue formed by inserting  $\{i, pend_i\}$  in the head of  $token\_q_i$  (no token is inserted if  $pend_i$  is *null*);  $maxpos_i$  is some large enough constant, say one plus the total number of tokens, if  $maxid_i$  is the aforementioned small constant.

Define  $M_i$  to be the pair  $\langle maxid_i, maxpos_i \rangle$ .  $M_i$  is a function from program states of  $\mathcal{P}$  to a well-founded set of pairs of integers under  $<$ , where  $<$  is the usual lexicographic order.

$(idle_i \wedge \neg partner_i \wedge \neg delay_i) \wedge empty(token\_q_i)$  unless  $\neg(idle_i \wedge \neg partner_i \wedge$   
 $\neg delay_i)$

, similar to Theorem 20. (1)

$(idle_i \wedge \neg partner_i \wedge \neg delay_i \wedge pend_i) \wedge (M_i = \vec{m})$  unless  
 $(\neg(idle_i \wedge \neg partner_i \wedge \neg delay_i) \vee (M_i < \vec{m})) \vee empty(token\_q_i)$

, similar to Theorem 20.

$(idle_i \wedge \neg partner_i \wedge \neg delay_i \wedge pend_i) \wedge (M_i = \bar{m}) \mapsto$   
 $\neg(idle_i \wedge \neg partner_i \wedge \neg delay_i) \vee (M_i < \bar{m}) \vee empty(token\_q_i)$   
, PSP theorem on the above and Lemma 24(d). (2)

$(idle_i \wedge \neg partner_i \wedge \neg delay_i \wedge \neg pend_i) \wedge (M_i = \bar{m}) \mapsto \neg(idle_i \wedge \neg partner_i \wedge$   
 $\neg delay_i) \vee ((idle_i \wedge \neg partner_i \wedge \neg delay_i \wedge pend_i) \wedge (M_i = \bar{m})) \vee empty(token\_q_i)$   
, similar to the proof of the above.

$(idle_i \wedge \neg partner_i \wedge \neg delay_i \wedge \neg pend_i) \wedge (M_i = \bar{m}) \mapsto$   
 $\neg(idle_i \wedge \neg partner_i \wedge \neg delay_i) \vee (M_i < \bar{m}) \vee empty(token\_q_i)$   
, cancellation theorem on the above two.

$(idle_i \wedge \neg partner_i \wedge \neg delay_i) \wedge (M_i = \bar{m}) \mapsto \neg(idle_i \wedge \neg partner_i \wedge \neg delay_i) \vee$   
 $(M_i < \bar{m}) \vee empty(token\_q_i)$   
, disjunction on the above and (2).

$(idle_i \wedge \neg partner_i \wedge \neg delay_i) \wedge (M_i = \bar{m}) \mapsto ((idle_i \wedge \neg partner_i \wedge \neg delay_i) \wedge$   
 $(M_i < \bar{m})) \vee (\neg(idle_i \wedge \neg partner_i \wedge \neg delay_i) \vee empty(token\_q_i))$   
, rewriting of the above.

$idle_i \wedge \neg partner_i \wedge \neg delay_i \mapsto \neg(idle_i \wedge \neg partner_i \wedge \neg delay_i) \vee empty(token\_q_i)$   
, induction on the above.

$\neg(idle_i \wedge \neg partner_i \wedge \neg delay_i) \mapsto \neg(idle_i \wedge \neg partner_i \wedge \neg delay_i)$   
, implication theorem on “invariant  $p \Rightarrow p$ ”.

$true \mapsto \neg(idle_i \wedge \neg partner_i \wedge \neg delay_i) \vee empty(token\_q_i)$   
, disjunction on the above two.

$true \mapsto ((idle_i \wedge \neg partner_i \wedge \neg delay_i) \wedge empty(token\_q_i)) \vee$   
 $\neg(idle_i \wedge \neg partner_i \wedge \neg delay_i)$   
, rewriting of the above.

$\square \diamond \neg((idle_i \wedge \neg partner_i \wedge \neg delay_i) \wedge empty(token\_q_i)) \Rightarrow$   
 $\square \diamond \neg(idle_i \wedge \neg partner_i \wedge \neg delay_i)$   
, special case of SF-UNITY on the above and (1). *End of Proof.*

**Lemma 30**  $\square \diamond (idle_i \wedge \neg pend_i \wedge \neg partner_i) \Rightarrow \square \diamond (\{i, j\} \notin token\_q_i)$ .

*Proof.* Let  $pos(token\_q_i, \{i, j\})$  take on the value of the position of token  $\{i, j\}$  in  $token\_q_i$ ;  $pos(token\_q_i, \{i, j\})$  is some large constant if  $\{i, j\} \notin token\_q_i$ .

Define  $N_i^{\{i, j\}}$  to be the pair  $(pos(token\_q_i, \{i, j\}), -|token\_q_i|)$ , where  $|token\_q_i|$  is the number of tokens in  $token\_q_i$ .  $N_i^{\{i, j\}}$  has the well-founded property as  $M_i$  does in Lemma 29.

$(N_i^{\{i, j\}} \leq \bar{n})$  unless  $(\{i, j\} \notin token\_q_i)$ .

, similar to the proof in Lemma 29.

$(idle_i \wedge \neg pend_i \wedge \neg partner_i) \wedge (N_i^{\{i, j\}} = \bar{n}) \mapsto (N_i^{\{i, j\}} < \bar{n}) \vee (\{i, j\} \notin token\_q_i)$ .

, similar to the proof in Lemma 29.

The lemma follows

-, SF-UNITY on the above two.

*End of Proof.*

**Theorem 23**  $\Box\Diamond ready_i \Rightarrow \Box\Diamond[\exists j :: head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j}]$ . (kp5)

*Proof.* It suffices to show  $\Box\Diamond ready_i \wedge \neg\Box\Diamond[\exists j :: head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j}] \Rightarrow false$ , i.e.  $\Box\Diamond ready_i \wedge \Diamond\Box[\forall j :: \neg(head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j})] \Rightarrow false$ .

$[\forall j :: flag_i \neq \{i, j\} \wedge flag_j \neq \{i, j\}]$  unless  
 $[\exists j :: head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j}]$   
 , simple conjunction on Theorem 20.

$[\exists j :: flag_i = \{i, j\} \vee flag_j = \{i, j\}] \mapsto [\forall j :: flag_i \neq \{i, j\} \wedge flag_j \neq \{i, j\}]$   
 , rewriting of (u4).

$\Box\Diamond[\exists j :: flag_i = \{i, j\} \vee flag_j = \{i, j\}] \Rightarrow$   
 $\Box\Diamond[\exists j :: head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j}]$   
 , special case of SF-UNITY on the above two.

$\Diamond\Box[\forall j :: \neg(head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j})] \Rightarrow$   
 $\Diamond\Box[\forall j :: \neg(flag_i = \{i, j\} \vee flag_j = \{i, j\})]$   
 , rewriting of the above.

$\Box\Diamond ready_i \wedge \Diamond\Box[\forall j :: \neg(head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j})] \Rightarrow$   
 $[\exists k :: \Box\Diamond idle_k \wedge \Box\Diamond idle_i \wedge \Diamond\Box[\forall j :: \neg(flag_i = \{i, j\} \vee flag_j = \{i, j\})]]$   
 , from the above and the definition of  $ready_i$ .

$\Box\Diamond ready_i \wedge \Diamond\Box[\forall j :: \neg(head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j})] \Rightarrow$   
 $[(\exists k :: \Box\Diamond(idle_k \wedge \neg pend_k \wedge \neg partner_k) \wedge \Diamond\Box(idle_i \wedge \neg partner_i \wedge \neg delay_i))]$   
 , from the above and Lemmas 26, 27, and 28.

$\Box\Diamond ready_i \wedge \Diamond\Box[\forall j :: \neg(head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j})] \Rightarrow$   
 $[\exists k :: \Box\Diamond(\{i, k\} \notin token\_q_k) \wedge \Diamond\Box(\{i, k\} \notin token\_q_i)]$   
 , from the above and Lemmas 29 and 30.

$\Box\Diamond ready_i \wedge \Diamond\Box[\forall j :: \neg(head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j})] \Rightarrow$   
 $[\exists k :: \Box\Diamond(\{i, k\} \notin token\_q_k) \wedge \Diamond\Box(\{i, k\} \in token\_q_k)]$   
 , from the above and Lemma 25.

$\Box\Diamond(\{i, k\} \notin token\_q_k) \wedge \Diamond\Box(\{i, k\} \in token\_q_k) \Rightarrow false$   
 , from " $\Box\Diamond\neg p \wedge \Diamond\Box p \Rightarrow false$ ".

$\Box\Diamond ready_i \wedge \Diamond\Box[\forall j :: \neg(head(input_i) = yes^{i,j} \vee head(input_j) = yes^{i,j})] \Rightarrow false$ .

, from the above two.

*End of Proof.*

This completes the proof of the variation of Algorithm A.

## CHAPTER 7

### Conclusion

We have studied the process interaction problem with additional strong fairness and fault-tolerance constraints. The problem was considered an abstraction of the most non-trivial task in implementing languages such as CSP and IP for distributed programming, i.e. the task to schedule or coordinate the symmetric, nondeterministic, and synchronous communications among a set of processes. A scheduler was required to satisfy three basic properties, including synchronization, mutual exclusion, and progress requirements.

Strong fairness properties were added, as they are crucial for the correctness of certain programs in the concerned programming languages. We have examined strong process fairness and strong interaction fairness in a great detail. Both negative and positive results were obtained.

Fault-tolerance capability of a scheduler is important, as in a distributed architecture some processor that is executing a process may fail while other processors remain functioning; it is desirable to minimize the effect of a failed processor on the entire system. We have considered two important failure models, namely the detectable fail-stop and the undetectable fail-stop models. In each model, we presented algorithms that can cope with process failures to a certain degree.

We also explored the use of UNITY in a formal treatment of the process interaction problem. In particular, the soundness and relative completeness of UNITY logic was extended to strong fairness properties. We believe that the results will also be useful for the formal specification and verification of other reactive systems that need to exhibit certain strong fairness properties.

We now conclude with a list of main contributions of the dissertation and a number of possible directions for future research.

#### 7.1 Contributions

The main contributions of the dissertation are summarized as follows:

1. We showed that, in general, strong interaction fairness (SIF) is impossible for binary and hence for multiway interactions and strong process fairness (SPF) is impossible for multiway interactions. We demonstrated the use of branching time temporal logic in obtaining a formal proof of the impossibility results.
2. We devised an algorithm for binary interactions that satisfies SPF and has best known message cost, bounded by  $2D + 2$ , and response time, bounded by  $D^2 + 5D$ , where  $D$  is the maximum number of interactions of which some process is a common participant. The algorithm was extended to guarantee SPF in the presence of a finite number of detectable fail-stop failures.
3. We proposed a transformation of the binary interaction problem to the dining philosophers problem that ensures SPF. Adopting an existing solution to the dining philosophers problem in the proposed transformation, we derived another fair algorithm that has a constant failure locality in the presence of undetectable failures. The response time of the derived algorithm is further improved to  $O(D^2)$ , which is asymptotically as good as that of the first algorithm. We also showed that at the cost of response time the failure locality can be further reduced.
4. We proved the operational implication of conditional UNITY properties. In particular, a strong fairness property of a program can be specified in the form of “Hypothesis:  $true \mapsto p$  Conclusion:  $true \mapsto q$ ”, which implies that “ $\Box\Diamond p \Rightarrow \Box\Diamond q$ ” is true for every execution of the program. The result was applied to the specification of the process interaction problem with strong fairness.
5. We established the equivalence between the provability of conditional property “Hypothesis:  $true \mapsto p$  Conclusion:  $true \mapsto q$ ” in UNITY logic and that of “ $\Box\Diamond p \Rightarrow \Box\Diamond q$ ” in Manna and Pnueli’s temporal logic, which is known to be relatively complete. In the process, we obtained a relatively complete inference rule for proving strong fairness properties of a program in UNITY. We also proposed two plausible inference rules to facilitate the verification of strong fairness properties. The results were applied to verifying the SPF property of a variation of our first algorithm.

## 7.2 Future Research

We suggest the following possible directions for future research:

- **Lower-Bounds:** As is evident from the literature and this dissertation, there is a close relationship between the process interaction problem and the dining philosophers problem. The tight lower bound on the response time of dining philosophers algorithms has long been an open problem, so has that of algorithms for process interactions. The existence of an efficient transformation from the process interaction problem to the dining philosophers problem indicates that the lower bound for the former problem is at most as large as that for the latter. However, it is not clear if it would be possible to design an algorithm for process interactions that has a response time better than that of the best known dining philosophers algorithm.
- **Automated Verification:** In the proof of the impossibility of SIF for binary interactions, we constructed a problematic execution of any solution that meets the basic problem requirements. It might be possible to prove the impossibility result by a model checking procedure, showing that there is no program, i.e. model, for the problem specification with SIF. The main task would be to extend some existing model checking procedure to handle compositional specifications.
- **Formalization of Fault-Tolerance:** Formal specification and verification of fault-tolerant programs has become an important subject of research. One of the common approaches is to view failures as the behavior of the “environment” of the fault-tolerant program, which is a reactive system that has to tolerate the faulty behavior of its environment. This approach may be generalized to handle problems like the process interaction problem where the solution sought after is actually a reactive system. The relevant failure assumptions need to be incorporated into the original specification of the non-faulty environment.



## REFERENCES

- [Apt88] K. Apt, N. Francez, and S. Katz. "Appraising Fairness in Languages for Distributed Programming." *Distributed Computing*, 2(4):226-241, 1988.
- [Awe90] B. Awerbuch and M. Saks. "A Dining Philosophers Algorithm with Polynomial Response Time." In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pp. 65-74, 1990.
- [Bac88] R.J. Back and R. Kurki-Suonio. "Distributed Cooperation with Action Systems." *ACM TOPLAS*, 10(4):513-554, October 1988.
- [Bag89a] R.L. Bagrodia. "Process Synchronization: Design and Performance Evaluation of Distributed Algorithms." *IEEE Transactions on Software Engineering*, 15(9):1053-1065, September 1989.
- [Bag89b] R.L. Bagrodia. "Synchronization of Asynchronous Processes in CSP." *ACM TOPLAS*, 11(4):585-597, October 1989.
- [Ber76] C. Berge. *Graphs and Hypergraphs*. North-Holland, 1976.
- [Buc83] G. Buckley and A. Silberschatz. "An Effective Implementation of the Generalized Input-Output Construct of CSP." *ACM TOPLAS*, 5(2):223-235, April 1983.
- [Cal84] A.R. Calderbank, F.R.K. Chung, and D.G. Sturtevant. "Increasing Sequences with Nonzero Block Sums and Increasing Paths in Edge-Ordered Graphs." *Discrete Mathematics*, 50:15-28, 1984.
- [Cha84] K.M. Chandy and J. Misra. "The Drinking Philosophers Problem." *ACM TOPLAS*, 6(4):632-646, October 1984.
- [Cha86] K.M. Chandy and J. Misra. "How Processes Learn." *Distributed Computing*, 1(1):40-52, 1986.
- [Cha87] A. Charlesworth. "The Multiway Rendezvous." *ACM TOPLAS*, 9(3):350-366, July 1987.
- [Cha88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

- [Cha91] T. Chandra and S. Toueg. "Unreliable Failure Detectors for Asynchronous Systems." In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pp. 325–340, August 1991.
- [Cho92] M. Choy and A.K. Singh. "Efficient Fault Tolerant Algorithms for Resource Allocation in Distributed Systems." In *Proceedings of the 24th ACM Symposium on Theory of Computing*, 1992.
- [Coo78] S.A. Cook. "Soundness and Completeness of an Axiom System for Program Verification." *SIAM Journal of Computing*, 7(1):70–90, 1978.
- [Cun90] H.C. Cunningham and G.-C. Roman. "A UNITY-style Programming Logic for Shared Dataspace Programs." *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365–376, July 1990.
- [Dij78] E.W. Dijkstra. "Two Starvation Free Solutions of a General Exclusion Problem.", 1978. EWD 625, Plataanstraat 5, 5671 AL NUENEN, Netherlands.
- [Dij88] E.W. Dijkstra. "Position Paper on 'Fairness'." *ACM SIGSOFT*, 13(2):18–20, April 1988.
- [Dod82] Department of Defense, United States. *Reference Manual for the Ada Programming Language*, 1982.
- [Dol87] D. Dolev, C. Dwork, and L. Stockmeyer. "On the Minimal Synchronism Needed for Distributed Consensus." *Journal of the ACM*, 34(1):77–97, January 1987.
- [Eme89] E.A. Emerson and J. Srinivasan. "Branching Time Temporal Logic." In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *LNCS 354: Linear Time, Branching Time and Partial Order in Logic and Models for Concurrency*, pp. 123–172. Springer-Verlag, 1989.
- [Fis85] M.J. Fischer, N. Lynch, and M. Paterson. "Impossibility of Distributed Consensus with One Faulty Process." *Journal of the ACM*, 32(2):374–382, April 1985.
- [Fra86a] N. Francez. *Fairness*. Springer-Verlag, 1986.
- [Fra86b] N. Francez, B. Hailpern, and G. Taubenfeld. "Script: A Communication Abstraction Mechanism." *Science of Computer Programming*, 6(1):35–88, January 1986.

- [Fra90] N. Francez and I.R. Forman. "Interacting Processes: Coordinated Distributed Programming." Technical Report STP-226-90, MCC, 1990.
- [Ger89] R. Gerth and A. Pnueli. "Rooting UNITY." In *Proceedings of the IEEE 5th International Workshop on Software Specification and Design*, pp. 11–19, 1989.
- [Gra73] R.L. Graham and D.J. Kleitman. "Increasing Paths in Edge Ordered Graphs." *Periodica Mathematica Hungarica*, **3**(1-2):141–148, 1973.
- [Hoa69] C.A.R. Hoare. "An Axiomatic Basis for Computer Programs." *CACM*, **12**(8):576–580, 1969.
- [Hoa78] C.A.R. Hoare. "Communicating Sequential Processes." *CACM*, **21**(8):666–677, August 1978.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [Jou91] Y.-J. Joung and S.A. Smolka. "Coordinating First-Order Multiparty Interactions." In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pp. 209–220, January 1991.
- [Jut89] C.S. Jutla, E. Knapp, and J.R. Rao. "A Predicate Transformer Approach to Semantics of Parallel Programs." In *Proceedings of the 8-th Annual ACM Symposium on Principles of Distributed Computing*, pp. 249–263, 1989.
- [Kna90] E. Knapp. "An Exercise in the Formal Derivation of Parallel Programs: Maximum Flows in Graphs." *ACM TOPLAS*, **12**(2):203–223, April 1990.
- [Kna92] E. Knapp. *Refinement as a Basis for Concurrent Program Design*. PhD thesis, The University of Texas at Austin, May 1992.
- [Lam91] L. Lamport. "The Temporal Logic of Actions." Technical Report 79, SRC DEC, 1991.
- [Liu92a] Y. Liu, A.K. Singh, and R.L. Bagrodia. "A Decompositional Approach to the Design of Efficient Parallel Programs." In D. Etiemble and J.-C. Syre, editors, *LNCS 605: PARLE Parallel Architectures and Languages Europe*, pp. 21–36. Springer-Verlag, June 1992.

- [Liu92b] Z. Liu and M. Joseph. "Transformation of Programs for Fault-Tolerance." *Formal Aspects of Computing*, **3**(4):442–469, 1992.
- [Lyn80] N.A. Lynch. "Fast Allocation of Nearby Resources in a Distributed System." In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, April 1980.
- [Man91] Z. Manna and A. Pnueli. "Completing the Temporal Picture." *Theoretical Computer Science*, pp. 97–130, 1991.
- [Man92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [Mis90] J. Misra. "Auxiliary Variables." *Notes on UNITY: 15-90*, July 10 1990.
- [Par92a] J. Parrow. *Fairness Properties in Process Algebra – with Applications in Communication Protocol Verification*. PhD thesis, Uppsala University, 1992.
- [Par92b] J. Parrow and P. Sjödin. "Multiway Synchronization Verified with Coupled Simulation." In W.R. Cleaveland, editor, *LNCS 630: CONCUR '92*, pp. 518–533. Springer-Verlag, August 1992.
- [Ram87a] S. Ramesh. "A New and Efficient Implementation of Multiprocess Synchronization." In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *LNCS 259: PARLE Parallel Architecture and Languages Europe*, pp. 387–401. Springer-Verlag, June 1987.
- [Ram87b] S. Ramesh. "A New Implementation of CSP with Output Guards." In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pp. 266–273, 1987.
- [Rao91] J.R. Rao. "On a Notion of Completeness for the Leads-to." *Notes on UNITY: 24-90*, July 15 1991.
- [Rei84a] J.H. Reif and P.G. Spirakis. "Real-time Synchronization of Interprocess Communication." *ACM TOPLAS*, **6**(2):215–238, April 1984.
- [Rei84b] W. Reisig. "Partial Order Semantics versus Interleaving Semantics for CSP-like Languages and Its Impact on Fairness." In *LNCS 172: Proceedings of the 11th ICALP*, pp. 403–413, July 1984.

- [San91a] B. Sanders. "Eliminating the Substitution Axiom from UNITY Logic." *Formal Aspects of Computing*, 3(2), 1991.
- [San91b] B. Sanders. "A Predicate Transformer Approach to Knowledge and Knowledge-based Protocols." In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, 1991.
- [Sch82] F.B. Schneider. "Synchronization in Distributed Programs." *ACM TOPLAS*, 4(2):125-148, April 1982.
- [Sch83] R.D. Schlichting and F.B. Schneider. "Fail-stop Processors: An Approach to Designing Fault-Tolerant Computing Systems." *ACM Transactions on Computer Systems*, pp. 222-238, 1983.
- [Sis84] A.P. Sistla. "Distributed Algorithms for Ensuring Fair Interprocess Communication." In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pp. 266-277, 1984.
- [Sty88] E. Styer and G.L. Peterson. "Improved Algorithms for Distributed Resource Allocation." In *Proceedings of the 7th ACM Annual Symposium on Principles of Distributed Computing*, pp. 105-116. 1988.
- [Tsa92a] Y.-K. Tsay and R.L. Bagrodia. "Deducing Fairness Properties for UNITY Programs — A New Completeness Result." Submitted to ACM LOPLAS, 1992.
- [Tsa92b] Y.-K. Tsay and R.L. Bagrodia. "A Real-Time Algorithm for Fair Interprocess Synchronization." In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pp. 716-723, June 1992.
- [Tsa93a] Y.-K. Tsay and R.L. Bagrodia. "An Algorithm with Optimal Failure Locality for the Dining Philosophers Problem." Submitted for publication, 1993.
- [Tsa93b] Y.-K. Tsay and R.L. Bagrodia. "Fault-Tolerant Algorithms for Fair Interprocess Synchronization." *IEEE Transactions on Parallel and Distributed Systems*, 1993. To appear.
- [Tsa93c] Y.-K. Tsay and R.L. Bagrodia. "Operational Implication of Conditional UNITY Properties." Submitted for publication, 1993.

- [Tsa93d] Y.-K. Tsay and R.L. Bagrodia. "Some Impossibility Results in Interprocess Synchronization." *Distributed Computing*, 6(4):221-231, 1993.