On Fairness and Multiparty Interaction

Coping with Finiteness and Conspiracies

David Ruiz

University of Sevilla

Doctoral Dissertation

September 30, 2003

First published in October 2003 by
The Distributed Group
ETSI Informática
Avda. de la Reina Mercedes s/n
Sevilla, 41012. SPAIN

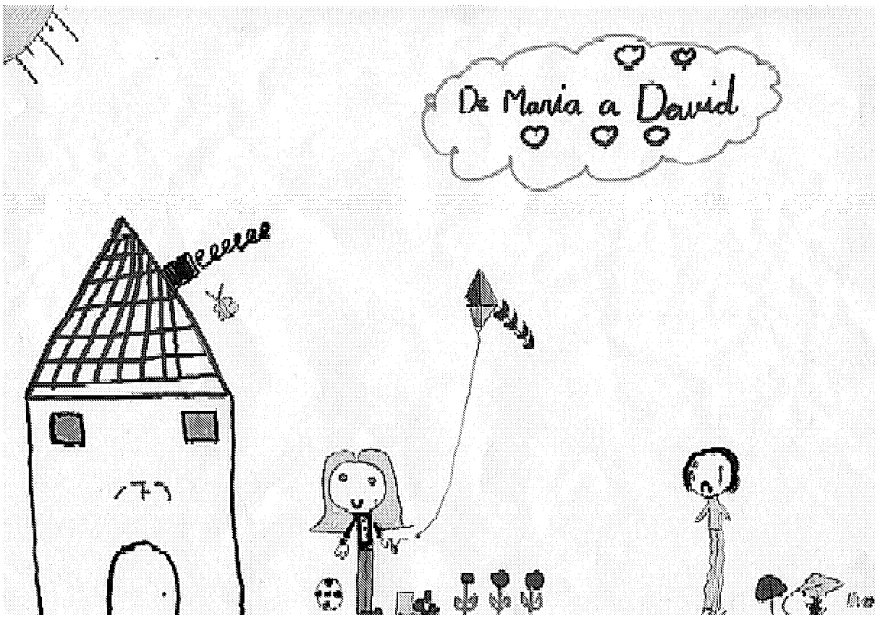
Copyright © MMIII The Distributed Group
http://www.tdg-seville.info
contact@tdg-seville.info

In keeping with the traditional purpose of furthering science, education and research, it is the policy of the publisher, whenever possible, to permit non-commercial use and redistribution of the information contained in the documents whose copyright they own. You however are *not allowed* to take money for the distribution or use of these results except for a nominal charge for photocopying, sending copies, or whichever means you use redistribute them. The results in this document have been tested carefully, but they are not guaranteed for any particular purpose. The publisher or the holder of the copyright do not offer any warranties or representations, nor do they accept any liabilities with respect to them.

Trademarks: DCOM, .NET, J♯, WINDOWS, WINDOWS XP PROFESSIONAL, and XLANG are registered trademarks of Microsoft Corporation. ADA is a registered trademark of the Government of The United States of America. i860 is a registered trademark of Intel, Inc. JAVA registered trademark of Sun Microsystems, Inc. OCCAM and TRANSPUTER are registered trademarks of Inmos, Inc. UNIX is a registered trademark of The Open Group. MEIKO is a registered trademark of Meiko Limited.

Classification (ACM 1998): D.1.3 Concurrent Programming: distributed and parallel programming; D.2.4 Software/Program Verification: correctness proofs; D.2.5 Testing and Debugging: tracing; D.3.3 Language Constructs and Features: co-routines, concurrency; D.3.1 Formal Definitions and Theory: semantics; D.4.1 Process Management: concurrency, scheduling, synchronisation; F.1.2 Modes of Computation: parallelism and concurrency, alternation and non-determi-nism; F.3.1 Specifying and Verifying and Reasoning about Programs: logics of programs; F.3.2 Semantics of Programming Languages: Operational semantics; F.3.3 Studies of Program Constructs: control primitives.

Support: Partially supported by the Spanish Ministry of Science and Technology under grants TIC97-0593-C05-05, TIC-2000-1106-C02-01, FIT-150100-2001-78, TIC2003-02737-C02-01, and Castilla-La Mancha Local Government under grant PCB-02-001.



Fairness by Maria Ruiz, aged six.

Fairness by Sara Ruiz, aged seven.

To my loving gordi.

Contents

Acknowledgements ..... **xi**

Abstract ..... **xiii**

Resumen ..... **xv**

I Preface

1 Introduction **3**
1.1 Research context 4
1.1.1 Concurrent systems 4
1.1.2 Interaction models 5
1.1.3 Multiparty interactions 6
1.1.4 Fairness 7
1.2 Summary of contributions 8
1.3 Structure of this dissertation 10

II Background information

2 Multiparty interactions **13**
2.1 Introduction 14
2.2 A taxonomy of multiparty interaction models 16
2.3 Semantics 17
2.4 Some examples 19
2.4.1 The dining philosophers 19

2.4.2 The leader election . . . 21
2.4.3 The OE SORT algorithm . . . 23
2.4.4 The ALOHA protocol . . . 25
2.5 Summary . . . 27

3 Fairness . . . 29
3.1 Introduction . . . 30
3.2 Safety and liveness properties . . . 31
3.3 Taxonomies of fairness notions . . . 33
3.3.1 Best's taxonomy . . . 34
3.3.2 Francez's taxonomy . . . 35
3.3.3 Alur and Henzinger's taxonomy . . . 37
3.3.4 Examples of fair and unfair executions . . . 38
3.4 Frameworks to define fairness . . . 40
3.4.1 A transitional framework . . . 40
3.4.2 A framework based on temporal logic of actions . . . 41
3.5 Controversy on fairness . . . 41
3.5.1 Is fairness a safety or a liveness property? . . . 42
3.5.2 Can fairness be ignored? . . . 43
3.5.3 Are fair implementations complete implementations? . . . 45
3.6 Summary . . . 49

4 Implementing fair multipartiness . . . 51
4.1 Introduction . . . 52
4.2 Implementing economy-based fairness . . . 52
4.3 Implementing strong fairness . . . 53
4.3.1 Impossibility results . . . 54
4.3.2 A straightforward scheduler . . . 55
4.3.3 An incremental scheduler . . . 56
4.3.4 A randomised scheduler . . . 56
4.4 Implementing ∞-fairness and hyperfairness . . . 57
4.5 Implementing k-bounded and finitary fairness . . . 58
4.6 Summary . . . 60

III Our contribution

5 Motivation . . . 65
5.1 Introduction . . . 66

5.2 Fair finiteness 66
5.3 Conspiracies 68
5.4 Analysis of current solutions 69
5.4.1 Economy-based fairness 69
5.4.2 Strong, weak and unconditional fairness 70
5.4.3 ∞-fairness and hyperfairness 70
5.4.4 k-bounded and finitary fairness 71
5.5 Discussion 74
5.6 Summary 74

6 Our framework to support multiparty interactions 77
6.1 Introduction 78
6.2 A formal characterisation 80
6.2.1 Assumptions 80
6.2.2 Core definitions 81
6.2.3 Miscellaneous implementable sets 84
6.2.4 Miscellaneous non-implementable predicates 84
6.2.5 Safety properties 85
6.3 Realising the benefits 86
6.4 Summary 88

7 A new fairness notion 89
7.1 Introduction 90
7.2 k-conspiracy-free fairness 90
7.3 Correctness 93
7.4 Summary 97

8 Implementing our framework and our notion 99
8.1 Introduction 100
8.2 Implementing the theoretical framework 100
8.2.1 Configurations 100
8.2.2 Predicates and functions 101
8.2.3 Semantics 102
8.2.4 Correctness of the implementation 103
8.2.5 Preservation of safety properties 106
8.2.6 Fairness requirements for the implementation 108
8.3 Implementing k-conspiracy-free fairness 108
8.3.1 Configurations 109
8.3.2 Semantics 109

8.3.3 Correctness of predicates and functions 111
8.3.4 Correctness and completeness of the algorithm 112
8.4 Summary 114

9 Empirical evaluation 115
9.1 Introduction 116
9.2 Timings 117
9.2.1 Timings as the number of participants increases 118
9.2.2 Timings as the execution length increases 122
9.3 Addressing conspiracies 125
9.4 Tuning k in the dining philosophers problem 129
9.4.1 An analytical characterisation 132
9.4.2 A simple method to select k 134
9.5 Summary 136

IV Final remarks

10 Conclusions and future work 141

V Appendices

A Strong k-fairness 145

B The IP language 147
B.1 An overall picture 147
B.2 The syntax in a nutshell 148
B.3 Simple examples 149
B.4 Interaction-driven programs 151

C Mathematical notes 153
C.1 Plotkin's method 153
C.2 Büchi automata 154
C.3 Framework notation 155
C.4 Mathematical notation 155

Bibliography 159

Index 171

List of Figures

1.1 Multiparty and biparty interactions 6

2.1 A taxonomy of languages that support multiparty interactions 18
2.2 A sample multiparty-based system 18
2.3 A static view of the dining philosophers problem 21
2.4 A static view of the leader election problem 22
2.5 A static view of the OE SORT algorithm 23
2.6 An execution of the OE SORT algorithm 25
2.7 A static view of the SLOTTED ALOHA protocol 26

3.1 Relationships amongst different fairness levels 34
3.2 Best's taxonomy of fairness 35
3.3 Francez's taxonomy of fairness 36
3.4 Alur and Henzinger's taxonomy of fairness 38
3.5 Relationships amongst some sample strong, finitary, and bounded executions 42
3.6 Distribution of lunches with both fair and unfair schedulers (I) 46
3.7 Distribution of lunches with both fair and unfair schedulers (II) 47
3.8 Distribution of lunches with Francez and Forman's scheduler using a bad random number generator 48

4.1 Joung and Smolka's impossibility results 55
4.2 Finitary transformation: original state chart and transition system . . . 60
4.3 Finitary transformation: resulting transition system 61

5.1 A sample execution to illustrate fair finiteness 67
5.2 A sample execution to illustrate conspiracies 68

5.3 A program with conspiracies and its corresponding state chart 72
5.4 Standard transition system for the program in Figure§5.3, and its weakly finitary-fair transformation 73

6.1 A mental map of multiparty-based systems 79

7.1 A couple of fully-connected systems 91

9.1 Execution time as the number of participants increases 119
9.2 TB's execution time as the number of participants increases 121
9.3 Selection time as the number of participants increases 121
9.4 Execution time as the execution length increases 123
9.5 Selection time as the execution length increases 123
9.6 TB's selection time as the execution length increases 125
9.7 A scenario to force conspiracies 126
9.8 Histograms when conspiracies are forced 127
9.9 Distribution of lunches when conspiracies are forced 130
9.10 Distribution of lunches using different random number generator . . 131
9.11 Tuning k-conspiracy-free fairness (I) 137
9.12 Tuning k-conspiracy-free fairness (II) 138

A.1 Practical comparison between k-conspiracy-free fairness and strong k-fairness 146

C.1 Büchi automaton to recognise $(S_0 + S_1)^* S_2^\infty$ 155

List of Tables

2.1 A taxonomy of languages for multiparty interactions 16

3.1 Examples of safety properties 31
3.2 Examples of liveness properties 31
3.3 Sample safety properties and their corresponding liveness properties 32
3.4 Differences amongst classical, finitary and bounded notions 42

5.1 Comparison with related notions 75

9.1 Execution time as the number of participants increases 119
9.2 Execution time tendency as the number of participants increases . . . 120
9.3 Execution times as the execution length increases 124
9.4 Execution time tendency as the execution length increases 124
9.5 Histograms when conspiracies are forced 128
9.6 Execution time when conspiracies are forced 128
9.7 A portion of the standard χ^2 table 133
9.8 Expected number of lunches with several $\mathtt{Think}$ functions 135
9.9 Finding an adequate value for k if $\mathtt{Think}(i) = (i-6)^2 + 1$ 135

C.1 Notation used to define our theoretical framework 156
C.2 Predicates used to define our theoretical framework 157
C.3 Functions used to define our theoretical framework 157
C.4 Mathematical notation 158

List of Programs

2.1 An IP_{CORE} solution to the dining philosophers problem 20
2.2 An IP_{CORE} solution to the leader election problem 22
2.3 An IP_{CORE} implementation of the OE SORT algorithm 24
2.4 The SLOTTED ALOHA protocol 26

3.1 Fairness to ensure termination and eventual response to a request for service 30
3.2 Sample programs to show fair and unfair executions 39
3.3 Dijkstra's example to show that fairness is a void property 43
3.4 Schneider and Lamport's counterexample 44

4.1 Tsay and Bagrodia's counterexample 54
4.2 Hyperfair transformation: original program 58
4.3 Hyperfair transformation: hyperfair program 59

5.1 A program with inherent conspiracies 69

9.1 A conspiracy-resistant version of the dining philosophers problems in interaction-driven normal form 117
9.2 A program to force conspiracies 125
9.3 A sample program to show how to tune k 131

Acknowledgements

Acknowledgement of one another's faults
is the highest duty imposed by our love of truth.

Ambrose G. Bierce, 1842–1914
Newspaper columnist

I am in no doubt that the best moment during the development of your *Philosophiæ Doctor* Thesis and the writing of your dissertation is when you finish them both. Not only because you reach the end of a hard way, but also because you can have a few minutes to relax, look back, and put your gratitude to many people in black ink.

Many individuals, friends and colleagues have been instrumental in making this dissertation a reality. Pivotal in this role was my research advisor, Dr. Rafael Corchuelo, since I would not have been able to finish my work without his help. Since he was the advisor of my master thesis, he has always trusted me and has always had the suitable, right words in every situation.

The help of a good research team is of uttermost importance. I have been lucky since my thesis work has been developed in the bosom of The Distributed Group, who have given me the encouragement and support I have needed since the beginning. Chiefly Dr. Miguel Toro, for his cheerful willingness to proof-read and criticise this document; without his support, I could not have begun this work. I shall never forget the arguments I had with Dr. José A. Pérez and Dr. Antonio Ruiz, or the breakfasts with José L. Arjona, Octavio Martín, and Joaquín Peña, which made my long working hours more bearable. I would also like to mention David Benavides, who joined the group recently.

Last, but not least, I thank my family and my girlfriend for putting up with me and tolerating my frequent bad mood, chiefly during the last few months when I have been so absorbed in the writing and polishing of this dissertation.

Abstract

You will have only an opportunity
to make a first impression.

Popular saying

If the solution to a problem requires three or more software artefacts to collaborate, there are some situations in which primitives such as remote procedure call, *rendez-vous* or message passing are not the most adequate tool from a conceptual point of view. The reason is that they require the programer to describe the interactions as biparty synchronisation and communication protocols in which the level of abstraction is often not appropriate to understand and reason about the properties of the problem.

The members of The Distributed Group have been working on multiparty interaction models since 1997. These models provide the programer with adequate mechanisms to describe complex interactions from a conceptual point of view, and they have devised several efficient implementations that are based on traditional synchronisation and communication primitives. In Corchuelo's thesis [35], it was presented a symbolic constraint-based model that was applied to the construction of early prototypes during the requirements elicitation and analysis stages; later, in Pérez's thesis [108], this model was refined and an efficient implementation was devised; finally, in Ruiz-Cortés's thesis [127], some aspects related to the quality of service were studied. Currently, in Peña's thesis [105], these models are being applied to multiagent systems; furthermore, in Martín's thesis [96], some results are being applied to the field of automatic negotiation amongst multiple parties.

There are many hot research topics in these interaction models, and our goal in this thesis has been to study notions that can guarantee that every interaction that can be executed sufficiently often is executed sufficiently often eventually. This allows to avoid undesirable behaviours due to the latency

of the communication network or the different speeds at which the processors work. Unfortunately, current notions suffer from a couple of drawbacks known as fair finiteness and conspiracies. They both constitute a major obstacle from a practical point of view since they prevent programmers from having precise control of the execution of their programs; furthermore, they motivate the need for a new notion to solve them.

In this dissertation, we present a new fairness notion to which we refer to as k-conspiracy-free fairness. Its main advantage is that it disallows the executions of a program that suffer from the aforementioned anomalies. We have also devised an efficient framework to implement it. Our experimental results support these claims, and the detailed study of other authors' work prove that our proposal constitutes a novel, original contribution.

Note on the term "fairness": According to the Cambridge Advanced Learner's Dictionary, the term "fairness" is defined as follows: "the quality of treating people equally or in a way that is right or reasonable". The Merriam Webster Dictionary gathers this definition, too, but goes into deeper details: "conforming with the established rules, consonant with merit or importance". The Spanish for both definitions is "*justicia*" (*/xustjθja/*) since according to the Royal Spanish Academy's Dictionary this term is defined as follows: "one of the four fundamental virtues that incline people to give each one what they deserve".

In this dissertation, the term "fairness" should not be misinterpreted. It does not mean that every interaction must have the same opportunities to be executed as the rest. On the contrary, they must have the *right* opportunities according to the features that are inherent to the program under consideration. Obviously, if the logic of a program enables an interaction only after one million other interactions have been executed, it is impossible for that interaction to execute more frequently since this would contradict the rules and would not be neither correct nor reasonable. On the contrary, it would be unfair not to execute that interaction at all or very little times with respect to the theoretical number of times it should have been executed.

Resumen

Sólo tendrás una oportunidad
de dar una primera impresión.

Dicho popular

Cuando la resolución de un problema requiere que tres o más artefactos software colaboren, hay situaciones en que las llamadas a procedimientos remotos, el *rendez-vous* o el paso de mensajes no resultan las primitivas más adecuadas desde un punto de vista conceptual. El motivo principal es que obligan a describir las interacciones en forma de protocolos de sincronización y comunicación bipartitos en los que, a menudo, se pierde el nivel de abstracción necesario para entender y razonar sobre las propiedades del problema.

En el grupo de investigación denominado The Distributed Group, llevamos trabajando desde 1997 en modelos de interacción entre múltiples participantes. Estos modelos proporcionan los mecanismos necesarios para describir interacciones complejas desde un punto de vista conceptual y, además, se han desarrollado implementaciones eficientes de los mismos basadas en las primitivas de sincronización y comunicación tradicionales. Inicialmente, en la tesis doctoral de Corchuelo [35], se presentó un modelo basado en restricciones simbólicas que se aplicó con éxito a la construcción de prototipos tempranos durante la fase de elicitación y análisis de requisitos; posteriormente, en la tesis doctoral de Pérez [108], este modelo se refinó hasta un punto en el que ya era posible una implementación eficiente preparada para la etapa de producción; finalmente, en la tesis de Ruiz-Cortés [127], se estudiaron aspectos relacionados con la calidad de servicio que ofrecen los artefactos que participan en una interacción. Actualmente, en la tesis de Peña [105] se está estudiando cómo aplicar estos modelos en sistemas multiagentes y en la de Martín [96] se están extrapolando algunos conceptos al campo de la negociación automática.

Entre los muchos problemas de investigación que existen en estos modelos de interacción, en esta tesis nos hemos centrado en el estudio de nociones

para garantizar que toda interacción que puede ser ejecutada suficientemente a menudo termina siendo ejecutada suficientemente a menudo con el tiempo. De esta forma se pueden impedir comportamientos indeseables provocados por la latencia de la red de comunicaciones o la diferencia de velocidad entre los distintos procesadores involucrados. Por desgracia, las nociones existentes sufren de un par de problemas conocidos como finitud justa y conspiraciones. Ambos constituyen un gran obstáculo desde el punto de vista práctico, ya que impiden al programador tener un control preciso sobre la ejecución de sus programas, y motivan, por lo tanto, la necesidad de contar con una nueva noción que losresuelva.

En esta tesis doctoral presentamos una nueva noción de selección justa a la que hemos denominado "k-conspiracy-free fairness". Su ventaja fundamental respecto a otras es que restringe el conjunto de posibles ejecuciones de un programa a aquéllas que no presentan ninguna de las dos anomalías anteriores. Además, hemos diseñado un framework que permite implementarla de una manera muy eficiente. Nuestros resultados experimentales corroboran estas afirmaciones y el estudio detallado del trabajo relacionado demuestra que se trata de una aportación original y novedosa.

Nota sobre el término inglés "*fairness*": Según el diccionario Cambridge Advanced Learner's, el vocablo inglés "*fairness*" se define como sigue: "*la cualidad de tratar a las personas por igual o de una forma correcta o razonable*". El diccionario Merriam Webster también recoge este significado y abunda en él en su sexta acepción: "*conforme a las reglas establecidas, en consonancia con el mérito o la importancia*". Ambas definiciones encajan perfectamente en la primera acepción que el Diccionario de la Real Academia de la lengua española recoge para el vocablo justicia: "una de las cuatro virtudes cardinales que inclina a dar a cada uno lo que le corresponde o pertenece".

Por lo tanto, al referirnos a "*fairness*" en este documento, el lector no debe incurrir en la falsa idea de que toda interacción debe tener las mismas oportunidades que el resto de ser ejecutada. Al contrario, debe tener las *justas* en función a las características inherentes al programa objeto de estudio. Evidentemente, si la lógica de un programa es tal que obliga a que una interacción sólo se pueda llevarse a cabo tras un millón de ejecuciones de otra, es imposible que todas puedan tener el mismo número de oportunidades de ser ejecutadas pues eso iría en contra de las reglas establecidas en el programa y no sería ni correcto ni razonable. Lo que resultaría injusto es que pudiendo ejecutar esa interacción lo suficientemente a menudo, no resultase seleccionada nunca o en un número de ocasiones insignificante en proporción al número de veces que podría haber sido ejecutada.

Part I

Preface

Chapter 1

Introduction

There is nothing more difficult to take in hand,
more perilous to conduct, or more uncertain in its success,
than to take the lead in the introduction
of a new order of things.

Niccolo Machiavelli, 1469–1527
Italian political theorist

Not only depends the behaviour of a concurrent system on its inherent properties or logic, but also on how the interactions amongst its artefacts are implemented and performed by the underlying operating system or hardware. Many factors may have an influence, e.g., the relative speed of the processors, the latency of the network or other resources, the scheduling policy, the loading module provided by the operating system, and so on. Fairness has long been recognised as a desirable notion since it allows to abstract away from those details, which are generally too complex to be taken into account or simply unknown.

In this dissertation, we report on our work to design a new fairness notion that can be applied to concurrent systems whose artefacts synchronise and communicate by means of multiparty interactions. In this chapter, we first introduce the elements that constitute the context of our research work in Section §1.1; we then summarise our main contributions in Section §1.2; finally, we describe the structure of the dissertation in Section §1.3.

1.1 Research context

In this section, we present a bird's eye view of the main concepts we use through the rest of the dissertation. First, we define and classify concurrent systems in Section §1.1.1; our view of interaction models is presented in Section §1.1.2; we introduce the multiparty interaction models in Section §1.1.3; finally, a short introduction to fairness is shown in Section §1.1.4.

1.1.1 Concurrent systems

A computer system is usually composed of *artefacts* that can be either active, (*e.g.*, processes, tasks, threads) or passive, in which case they do not have their own activity and are used by the active artefacts (*e.g.*, instances of abstract data types, system resources, files).

A system is said to be *concurrent* if it contains more than one active artefact, and we distinguish between *competitive* and *cooperative concurrency*. The former exists when two or more active artefacts are designed separately and are not aware of each other, but use the same passive artefacts. Usually, active artefacts compete for a resource which knows nothing about the artefacts that can use it, and it can serve any clients as long as it is not busy. For instance, competitive concurrency exists in client-server systems, in databases, in operating systems in which different tasks use the same system resources (*e.g.*, heaps or printers), or in concurrent object-oriented systems in which several threads need to have access to the same object. Cooperative concurrency exists when several artefacts cooperate, *i.e.*, do some job together and are aware of this fact. They can communicate by sharing resources or explicitly, but the important point is that they have been designed together so that they can cooperate to achieve their joint goal and use each other's help and results. They synchronise their execution and can wait for information computed by other cooperating artefacts. Examples of such systems may be found in the fields of real time applications, control systems, systolic algorithms, job sharing, or leader election algorithms.

This categorisation of systems is not explicitly related to the primitives used to synchronise or exchange information. They are usually low level and, generally speaking, one can build either cooperative or competitive systems using nearly the same primitives, although some of them are more commonly used for building systems of a particular category. For instance, remote procedure calls, object method calls, shared variables and monitors are often used by active artefacts in competitive systems to have access to the same passive artefacts; nonetheless, they can be used in cooperative systems by two

clients to cooperate via servers. *Rendez-vous*, message passing or signals with known senders and receivers are more suitable for cooperative relationships.

In this dissertation we focus on both competitive and cooperative concurrency as long as synchronisation and communication is carried out by means of multiparty interactions.

1.1.2 Interaction models

Coordination is a cornerstone of most current activities. Malone characterised it as an emerging research area with an interdisciplinary focus, playing a key role in many diverse disciplines such as economics, operations research, organisation theory or biology [94]. It is thus not surprising that the desire for coordination has impregnated computer science. Francez and Forman, for instance, stated that the desire to support coordination is the quintessential reason for current distributed systems [58].

Unfortunately, there is not a single prevailing, general definition of *coordination*. They range from simple statements like "coordination is managing dependencies between activities" [94], to rather elaborate descriptions such as "coordination is the additional information processing performed when multiple, connected actors pursue goals that a single actor pursuing the same goals would not perform" [93].

In the field of programming languages, however, it is widely accepted that "coordination is the process of building programs by gluing together active pieces" [28]. How to glue them depends completely on the coordination model used. Roughly speaking, such a model provides the basis to express how a set of artefacts can synchronise and communicate to reach an agreement [58]. In the literature, such models are very often referred to as interaction models, and this is the term we shall use through the rest of the dissertation.

Papadopoulos and Arbab presented a detailed study of state-of-the-art *interaction!models* in Reference [104], and characterised them generically as triples of the form (E, L, M), where E represents the artefacts being coordinated, L the media used to coordinate them, and M the semantic framework to which the model adheres. An *interaction!language* is thus viewed as the linguistic embodiment of an interaction model, and they offer syntactic facilities to express synchronisation, communication, creation and termination of computational activities.

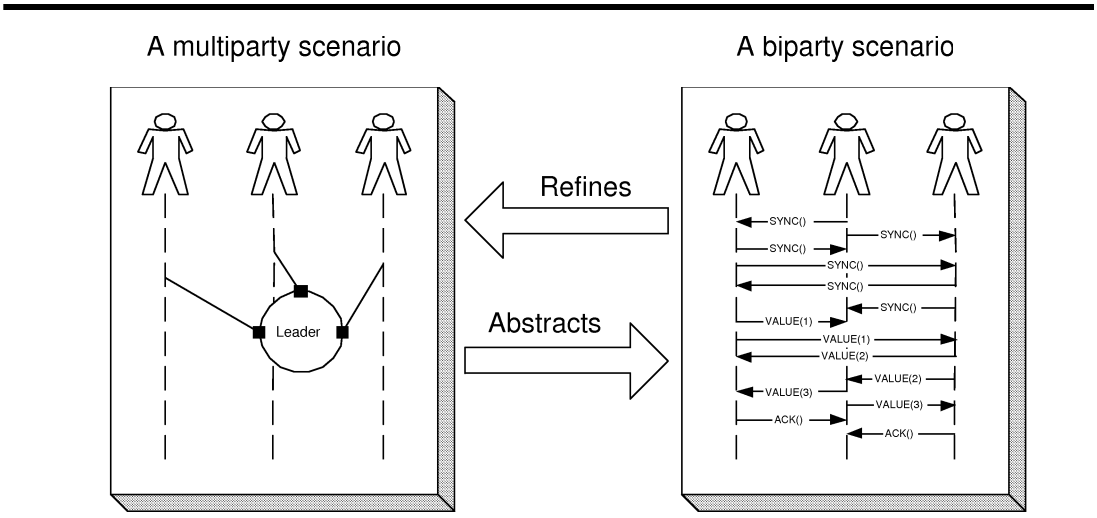


Figure 1.1: *Multiparty and biparty interactions.*

1.1.3 Multiparty interactions

Most real-world interactions are *multiparty* by nature since they require multiple parties to coordinate to reach an agreement. However, not many interaction models allow several artefacts to interact together since they provide biparty interaction primitives only, *e.g.*, *rendez-vous*, remote procedure call, message passing, message queues, or tuple spaces. Even the latest peer-to-peer systems emphasise two artefacts that exchange information.

Although these primitives are in widespread use and there are many efficient implementations, they are not the best tool we can use to model complex, real-world, multiparty interactions. The reason is that they usually require them to be broken into a set of biparty interactions and a protocol to coordinate them. Very often, reusing or reasoning about those protocols is very difficult, which argues for a solution in which complex interactions amongst multiple artefacts can be expressed more naturally.

Figure §1.1 sketches how the classical leader election problem can be represented in terms of a multiparty interaction and a set of biparty interactions with a protocol. Obviously, the solution on the left is more simple from a conceptual point of view, but as efficient as the solution on the right since there are well-known, automatic methods to transform multiparty interactions into biparty interaction protocols [38, 40, 109]. These methods, thus, relieve the programmer from the burden of coding them manually, and allow to reuse them more easily and safely. Furthermore, reasoning about a program in terms of

multiparty interactions was proved to be more simple than reasoning in terms of biparty interactions [58].

Currently, there are many research multiparty interaction models and languages, *e.g.*, POLYPHONIC C♯ [17], IP [58], CAL [38], LOTOS [21], EXTENDED LOTOS [24], or RADDLE [56]; furthermore, the number of commercial languages is increasing at a constant rate, *e.g.*, XLANG by Microsoft [137], WSFL by IBM [89], or the joint proposal called BPEL4WS [132].

1.1.4 Fairness

Fairness notions allow programmers to abstract away from the low-level details concerning how an artefact is scheduled or how the underlying operating system implements a particular primitive in a concurrent environment. There is not a single prevailing definition, but they all envelope the same idea: no artefact should be denied forever its rightful opportunity to proceed [102].

For instance, consider an artefact that executes the following Dijkstra's multi-choice command [49]:

```
[ true → skip [] true → x := x + 1; print(x) ]
```

It states that it does not matter whether x is increased and printed out or not, and an implementation that does not take the second choice into account is, in principle, as correct as another that selects choices according to the result a random number generator outputs. Nevertheless, if we incorporate this small fragment into the following loop, assuming that both choices shall eventually be executed is the only way to guarantee termination.

```
x := 0;
*[ x < 10 →
   [ true → skip
   [] true → x := x + 1; print(x)
   ]
 ]
```

From an intuitive point of view, we should be willing to accept that this is a program that prints out the sequence 1, 2, . . ., 10 and stops. However, assuming that **skip** may take a few milliseconds to finish, the time it takes to print out this sequence may vary from run to run. If we assume that nondeterminism is embodied in the implementation of this program, we would like to be sure that both choices have a rightful chance to be executed independently from how such commands are implemented. In other terms, we would like to be able to assume that the implementation is fair.

Fairness is a key concept in most multiparty interaction models since they usually allow an artefact to be willing to participate into several (groups of) interactions that are mutually exclusive. Thus, the only way to guarantee that each (group of) interactions has a chance to be executed is to assume that the implementation is fair.

1.2 Summary of contributions

Our dissertation focuses on studying fairness in the context of the multiparty interaction model supported by the IP_{CORE} language [58]. We have analysed classical fairness notions carefully and concluded that they are not usually well-suited to be applied to real-world programs because they suffer from two practical problems, namely: they rely on eventuality and infiniteness, *i.e.*, concepts that cannot be implemented, and many of them do not take conspiracies into account, *i.e.*, situations in which an unfortunate interleaving of independent actions prevents an interaction from being ready to be executed.

Our main result is a new fairness notion to which we refer to as k-*conspiracy-free fairness*. Its main advantages follows:

i. It solves the above-mentioned problems *simultaneously*, whereas the others notions consider them separately.

ii. The value of k can be set beforehand to tune its goodness to solve both problems.

iii. It can be applied to any programs, whereas some notions can be applied to a subset of normal-form programs only.

iv. We have devised a non-transformational, general algorithm to implement it, whereas other notions have been implemented by means of *ad hoc* schedulers only.

We have also devised a theoretical framework to define the multiparty interaction model with which we deal rigorously and independently from the concrete interaction language used to support it. This framework is general enough to define other fairness notions, as well as common safety and liveness properties. We have also devised an efficient $\text{J}^\sharp$-based implementation to validate it.

Many results have materialised in publications in mainstream conferences and journals. Below is a complete list of publications in which we point out the main cornerstones during the development of our results:

1998: We designed a compiler for the IP_{CORE} language in fulfilment of my Master's Thesis requisites. The results were presented at the Third Menhir Workshop [36].

1999: We refined our results on implementing IP_{CORE} and devised a centralised, general, strongly fair scheduler that was presented at the SEID'99 Conference [126] and the IEEE Euromicro'99 Conference [44]. We soon began working on a semi-distributed version that was presented at the Third IEEE CSCC'99 Conference [41]. An empirical evaluation of our schedulers was presented at the Fourth Menhir Workshop [42].

2000: We presented a detailed report on uses and applications of multiparty interactions in journal *Novática* [43]. This year, we identified infiniteness and conspiracies as two major practical problems of classical fairness notions. We reported on them at the Fifth Menhir Workshop [119], but kept on working on a fully distributed strongly fair scheduler. We presented it at the SEID'00 Conference.

2001: We devised an open-multiparty interaction model in which artefacts do not need to know each other before hand, and presented a CORBA/JAVA-based framework to implement it at the DAIS'00 Conference [109]. We worked on solving conspiracies and finiteness and presented preliminary ideas at the PROLE'01 and SI+T'01 Conferences [120, 121].

2002: We devised a notion to address conspiracies, which still can be applied to programs that terminate. We referred to this notion as strong k-fairness and published some results in The Colombian Journal of Computation [124] and the proceedings of the Euro-par'02 Conference [122]. Our framework to support open multiparty interactions was refined and the results were presented at the ACM SAC'02 Conference [110]; the details on the algorithm to achieve mutual exclusion amongst such interactions were presented at the COORDINATION'02 Conference [111]. We realised that strong k-fairness was not restrictive enough to eliminate conspiracies, and worked on a stronger version that was presented at the PROLE'02 Conference [123].

2003: We polished our ideas and devised k-conspiracy-free fairness. The notion, the algorithm to implement it, an empirical study and a comparison with strong k–fairness was submitted and accepted for publication in journal Concurrency and Computation: Practice & Experience [125].

1.3 Structure of this dissertation

This manuscript is organised as follows:

Part I: Preface. It comprises this introduction only.

Part II: State of the art. Here, our goal is to provide the reader with a deep understanding of the research context in which our results have been developed. In Chapter §2, we report on the multiparty interaction model with which we deal and illustrate it by means of several examples; since this is not the only existing model, we also update Joung's taxonomy with recent proposals. In Chapter §3, we focus on fairness and present an overall idea of the notions and frameworks currently available; we also report on some controversial aspects. Finally, in Chapter §4, some algorithms to implement fairness are described, as well as some interesting results about the impossibility of implementing strong fairness by means of wait-free algorithms.

Part III: Our contribution. It is the core of our dissertation, and it is organised into five chapters. In Chapter §5, we motivate our research and prove that classical fairness notions are not practical enough. In Chapter §6, we present our framework to define the multiparty interaction model with which we deal, and illustrate how powerful it is. In Chapter §7, we define our fairness notion and prove that it is sound. In Chapter §8, we present the algorithms we have devised to implement both our framework and our notion. Finally, in Chapter §9, we present the results of a detailed experimental study that helped us validate our proposal.

Part IV: Final remarks. It consists of one chapter in which we report on our main conclusions and future research directions.

Part V: Appendices. We report on our preliminary attempts to cope with fair finiteness and conspiracies in Appendix §A; we present a bird's eye description of IP and IP_{CORE} in Appendix §B; finally, we summarise the mathematical notation we use in Appendix §C.

Part II

Background information

Chapter 2

Multiparty interactions

[9] *Two are better than one, because they have a good return for their work;*
[12] *Though one may be overpowered, two can defend themselves.*
A cord of three strands is not quickly broken

Ecclesiastes, Chapter 4

Our goal in this chapter is to provide the reader with a good understanding of multiparty interactions. In Section §2.1, we motivate their need and present a detailed study in which we identify several fields in which they play an important role; in Section §2.2, we show a taxonomy of languages that support them; since there are many similar multiparty interaction models, we necessarily had to focus on only one whose semantics are described in Section §2.3; we illustrate the model by means of several examples in Section §2.4; finally, we summarise the chapter and our contributions to this field in Section §2.5.

2.1 Introduction

Interaction primitives such as remote procedure call or message passing are the *de facto* industrial standard for communicating software artefacts, and new materialisations such as SOAP [23, 52] or .NET REMOTING [115] are sprouting out at an increasing pace. Using these primitives to model complex interactions amongst an arbitrary number of artefacts very often requires breaking them into a set of biparty interactions and a protocol to coordinate them. In general, devising such protocols is not trivial, and the result is often poorly reusable and difficult to understand or reason about [58]. Therefore, the concerns of abstractness and reusability argue for primitives in which natural multiparty interactions can be expressed directly, but transformed later into efficient biparty protocols automatically, thus relieving the programmer from the burden of coding, testing, and maintaining them [38, 40, 109]. Furthermore, it has also been shown that such interactions may help achieve optimal concurrency in some scenarios [136].

The situations in which several artefacts need to interact jointly are common in practice [58]: transferring money from a bank to another by means of a point of sales terminal (three artefacts) [55, 117], paying taxes on-line (three artefacts in Spain: a taxpayer, the Exchequer, and Spain's Certification Authority), filtering in e-commerce [54] (a customer, a filter system, and several service providers), or reaching a virtual agreement in an auction sale (multiple artefacts).

The need for primitives to support multiparty interactions directly has been recognised in many contexts, namely:

Analysis and design methods: Most of them provide designers with tools to model multi-object collaborations. Different terms are used to refer to them: component diagrams [22], process models [29], message connections [34], data-flow diagrams [128], collaboration graphs [139], scenario diagrams [116] or collaborations [51, 129]. These proposals are accompanied by a rich set of examples that show the adequacy of such modelling tools in fields including finance, telecommunication, insurance, manufacturing, embedded systems, process control, flight simulation, travel and transportation, or systems management.

Web applications: Very recently, the increasing complexity of web applications and the rather elaborate technologies behind the scenes, have motivated most software companies to produce tools to deal with multiparty interactions on the Internet. Microsoft, for instance, proposed a

framework called .NET ORCHESTRATION [90] to help programmers coordinate an arbitrary number of networked artefacts, *e.g.*, web services, message queues, DCOM or .NET components. The key to this framework is the XLANG language [137], which is an extension to WSDL. IBM proposed a language called WSFL [89] as the basis to describe complex interactions amongst multiple business processes. It was designed to be the core of their WEB SERVICES TECHNOLOGY FRAMEWORK and relies and complements existing specifications such as XMLP, SOAP, WSDL or UDDI. Recently both proposals have been unified into the BPEL4WS framework [132]. It consists of a set of components that provide the infrastructure needed to support interactions amongst web services, and a formal specification language to define business processes and interactions amongst them.

JAVA concurrency frameworks: Multiparty interactions have attracted the attention of many researchers who have devised several JAVA frameworks to support them. In Reference [86], the authors propose a framework that supports barrier synchronisation, which may be viewed as a simple form of multiparty interaction; this framework has become the basis for the specification request #166 within the JAVA COMMUNITY PROCESS [87]. In Reference [55], the authors present a framework that supports 1-to-n interactions in which an object may request several objects to interact with it atomically. The proposal in Reference [77] leverages SR [7, 64] and extends JAVA to provide a rich concurrency model that includes ADA-like biparty interactions. Such interactions were generalised to the multiparty case in Reference [31]. In Reference [65], another JAVA framework that supports CSP-like biparty interaction was introduced. Unfortunately, except for the framework in Reference [38], none of the previous proposals can be viewed as a full implementation of multiparty interactions since either interactions are not symmetric [55, 77], or they cannot guard the execution of a choice in a multi-choice command [86, 55].

Agent societies: Recently, the need for multiparty interaction has also been recognised in the field of multiagent societies [6, 14, 15, 27, 100]. Such interactions are normally used in early stages during the design of multiagent societies, but there are some proposals that aim at using them to implement behavioural relationships amongst agents [105, 106, 107].

Research languages: Beyond JAVA, there are many research languages that support multiparty interactions. A taxonomy can be found in Reference [75], and an updated version is shown in the following section.

Primitive	Description	Languages
Channel	It defines an interaction between two known artefacts.	CSP [67], MULTIPARTY CSP [73], OCCAM [66], Pϵω [9]
Port	It defines an interaction between two roles that may be played by different artefacts.	ADA [101], CCS [98], GMS [63], MEIJE [45]
Gate	It defines an interaction amongst a fixed set of known artefacts.	CSPS [62], CIRCAL [97], RADDLE [56], LOTOS [21], EXTENDED LOTOS [24], ACTION SYSTEMS [11], IP_{CORE} [58], CAL [39], CLIP [35]
Team	It defines an interaction amongst a fixed set of roles which may be played by different artefacts.	SCRIPT [59], RADDLE, IP [58], JPS [105], BPEL4WS [132]

Table 2.1: *A taxonomy of languages for multiparty interactions.*

2.2 A taxonomy of multiparty interaction models

Although it is very usual to refer to *the* multiparty interaction model as if it was unique, it is not a single abstraction, but a collection of models with an overall constant theme: to allow several artefacts to interact coordinately using high-level primitives. Unfortunately, very few references make a distinction between the interaction model and the interaction or programming language that supports it, which may cause confusion in some cases. This is the reason why we refer to individual interaction models using the name of one of the languages that support it.

In Reference [70], Joung made an effort to devise a taxonomy of languages that support multiparty interactions. First, he classified the primitives they support into four groups, namely: channels, ports, gates, and teams (*cf.* Table §2.1). Later, he classified them depending on whether they are synchronous or asynchronous, and the level of intra-artefact parallelism they allow, namely:

Null: Although it is not very usual, languages such as SCRIPT or COMPACT do not allow any artefact to execute (or be willing to execute) more than an interaction at a time.

Conjunctive: In this case, the model allows an artefact to execute interactions in groups that are expressed by means of so called *compound commands*. Such interactions must be executed jointly for the compound command to be executed, or none of them is executed, *i.e.*, they behave as a transaction.

Disjunctive: This level allows interaction commands to be used as guards in multi-choice commands. Therefore, an artefact may express its willingness to participate in an arbitrary set of interactions, although only one of them can be executed at a time.

Disjunctive normal form: This level allows the guards of a multi-choice command to include compound commands. In this case, each choice allows an artefact to express its willingness to participate in a subset of interactions, although only a subset may be executed at a time.

Disjunctive-conjunctive: This is a restrictive level in which disjunctive parallelism is allowed, but compound statements may not be used as guards in multi-choice commands.

Figure §2.1 sketches an updated version of Joung's taxonomy. Note that languages that support biparty interactions only have been omitted, *e.g.*, ADA, CCS, CSP, OCCAM. Furthermore, new languages that did not exist as of the time of publishing Reference [70] have been added in *italics*.

2.3 Semantics

In this dissertation we focus on the IP_{CORE} language, which outstands since it is intended to have a dual role. On the one hand, it is equipped with sound semantics that turn it into a language amenable to formal reasoning; thus, it is very adequate to specify concurrent systems formally. On the other hand, it is intended to be assembler language by means of which other higher-level languages can be implemented [35].

In Appendix §B, we present a detailed introduction to IP_{CORE}. In this section we strictly focus on the semantics of the interaction model it supports since it is the key to understand why *fairness* is so important in this context. Anyhow, focusing on this language does not amount to loss of generality since the results we present in this dissertation can be easily applied to any interaction model or language as long as the set of interactions in a system is finite and known before hand, which is a realistic constraint.

In IP_{CORE}, artifacts are modelled as processes (also called participants) which own a local state and communicate with each other by means of multiparty interactions only. Processes are single-threaded and support disjunctive parallelism. This, however, may not be seen as a shortcoming since a real-world artefact with, say, three threads may be viewed as three related IP_{CORE} processes.

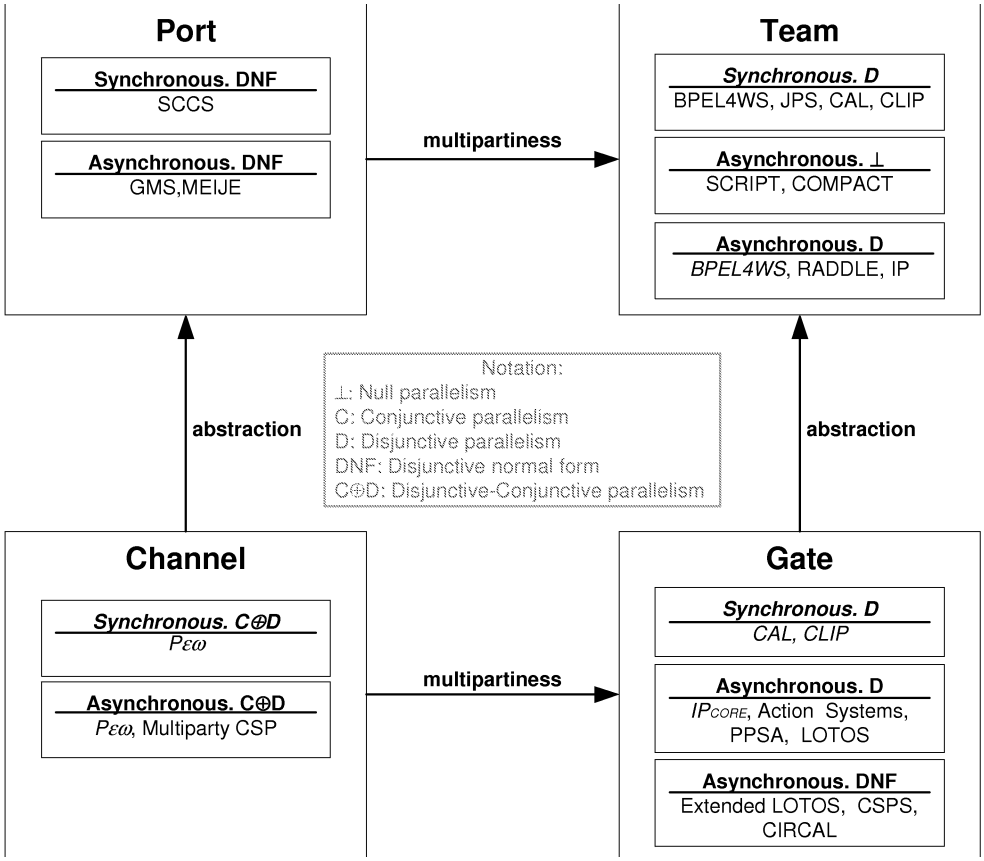


Figure 2.1: *A taxonomy of languages that support multiparty interactions.*

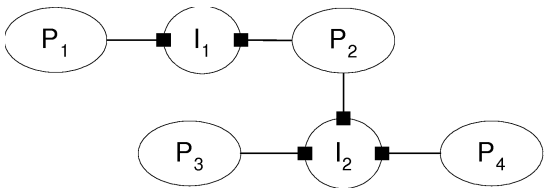


Figure 2.2: *A sample multiparty-based system.*

Interactions are identified by a unique name and have a fixed set of participants that need to be known beforehand. For an interaction to become enabled, all of its participants need to offer to participate in it by executing an interaction command of the form $A\langle x_1 := e_1; \ldots; x_n := e_n\rangle$, where A is the name of the interaction and $x_1 := e_1; \ldots; x_n := e_n$ is a communication statement. Once an interaction becomes enabled, it can be executed as long as its participants commit to it, *i.e.*, they do not withdraw their offers and commit to another interaction before. (Often, it is said that the interaction selected achieves mutual exclusion.) Once an interaction is selected for execution, a state that combines a copy of the local states of the participants of that interaction is formed. This shared state can be viewed as a blackboard that allows a process to have access to the information other processes own by means of communication statements. In the example above, if a process succeeds in executing $A\langle x_1 := e_1; \ldots; x_n := e_n\rangle$, *i.e.*, it agrees with the other participants to execute interaction A, it then can evaluate expressions e_1, e_2, ..., e_n on the local state of other participants and transfer the results to variables x_1, x_2, ..., x_n in its own local state. This simple communication mechanism was improved in Reference [44], for instance, but the details fall outside the scope of this dissertation.

It is worth noting that an interaction being enabled does not amount to its execution. Figure §2.2 sketches a simple system composed of four processes, a two-party interaction and a three-party interaction. Note that process P_1 can offer to participate in I_1, whereas P_3 and P_4 can offer to participate in I_2, and P_2 can offer to participate in either I_1 or I_2. This means that these interactions cannot be executed simultaneously since IP_{CORE} supports disjunctive parallelism only. Thus an election under them needs to be held to decide which one should be executed, and it is important to select *fairly* so as to avoid denying some of them a rightful opportunity to be executed.

2.4 Some examples

Our goal in this section is to illustrate IP_{CORE} by means of some simple, but not trivial examples. Please, consult Appendix §B if necessary for an explanation of the local computation commands we use.

2.4.1 The dining philosophers

We illustrate multiparty synchronisation by means of the well-known dining philosophers problem [48]. It consists of five philosophers sitting at a table who do nothing but think and eat. There is a single fork between each

```
DINNER :: [|||_{i=1}^{N} P_i || F_i], where
        P_i :: *[ Get_i⟨⟩ → eat; Rel_i⟨⟩; think ] ;
        F_i :: *[ Get_i⟨⟩ → Rel_i⟨⟩
              [] Get_{i+1}⟨⟩ → Rel_{i+1}⟨⟩ { We assume that subindex arithmetic is module N }
              ] .
```

Program 2.1: *An* IP_{CORE} *solution to the dining philosophers problem.*

two neighboring philosophers, and they need to get both forks before eating. In addition, each philosopher should be able to eat as much as the rest, *i.e.*, the executions of this system should be fair. This problem is the core of a large class of problems in which an active artefact needs to acquire a set of resources in mutual exclusion.

The obvious solution to this problem, using two-party interactions, consists of picking up forks in sequence. Nevertheless, a problem arises if each philosopher grabs the fork on his/her right, and then waits for the fork on his/her left to be released. In this case, a deadlock has occurred, and all philosophers shall starve. In Reference [92], the authors prove that assuming no means of communication amongst philosophers other than through information attached to their forks, any solution in which all philosophers are programmed identically must have a possibility of deadlock. Thus, correct solutions must rely on some distinctions to be made amongst the philosophers. These solutions are usually not scalable or reusable since the distinction a philosopher has to implement depends heavily on the topology of the problem under consideration.

If we used multiparty interactions, each philosopher would get his/her two forks at the same time so that no deadlock could arise. Program §2.1 shows a solution to this problem and Figure §2.3 and static view. The philosophers are represented by P_i and the forks by F_i ($i \in [1, N]$). Each P_i eternally tries to get its associated forks by participating in the three-party interaction called Get_i together with F_i and F_{i-1}. Thus, acquiring a resource is specified as synchronising with the corresponding processes in a multiparty interaction. After P_i has got its forks, it "eats", releases them and spends some time at "thinking". Note that interactions Get_i and Get_{i+1} are in conflict when they are both enabled, but only one of them can be executed at the same time. The only way to guarantee that each interaction that is enabled sufficiently often shall eventually be selected for execution consists of assuming that the underlying conflict resolution mechanism is *fair*.

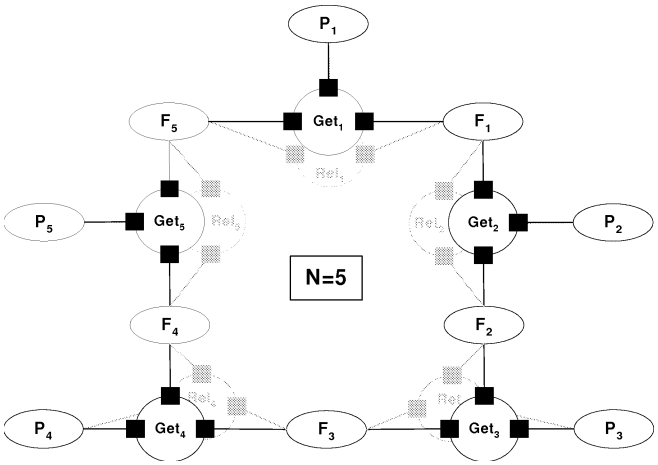


Figure 2.3: *A static view of the dining philosophers problem.*

2.4.2 The leader election

We illustrate how multiparty communication works by means of the leader election problem [85]. It has been paid much attention because it is the core of a large class of problems in which there are several artefacts able to execute an algorithm, but there is no *a priori* candidate to run it. Therefore, an election under them needs to be held.

This situation can be the case of an initialisation procedure that must be executed initially or a recovery procedure that must be executed after a crash of a distributed system. It is not possible to assign an artefact to the role of leader because it might crash and the system would not be able to restart. The criterion used to select a leader is quite simple: each of the artifacts has a different natural weight, *e.g.*, its net address, and the leader is the artefact with the maximum weight. Note that this is a clear example in which several artefacts need to collaborate simultaneously in a multiparty interaction because there is absolutely no way to select a leader if an artefact does not have information about the weights of the whole set of artefacts.

An IP_{CORE} solution to this problem is shown in Program §2.2. As we show in the static view in Figure §2.4, the multiparty interaction *Elect* synchronises all of the processes, allowing them to exchange information and decide which one has to be assigned to the role of leader. When several processes synchronise and interact, a temporary global combined state is formed so that each process can read information in the state of other participants of that interaction and transfer results to their own local state. This way, the communi-

LEADER :: $[|||_{i=1}^{N} M_i]$, **where**
M_i :: { w_i: *natural; leader*$_i$: *boolean* }
w_i := *a weight*;
Elect⟨leader$_i$:= $(w_i = \max_{1<j<n} \{w_j\})$⟩;
[leader$_i$ → *execute algorithm*].

Program 2.2: *An* IP$_{\text{CORE}}$ *solution to the leader election problem.*

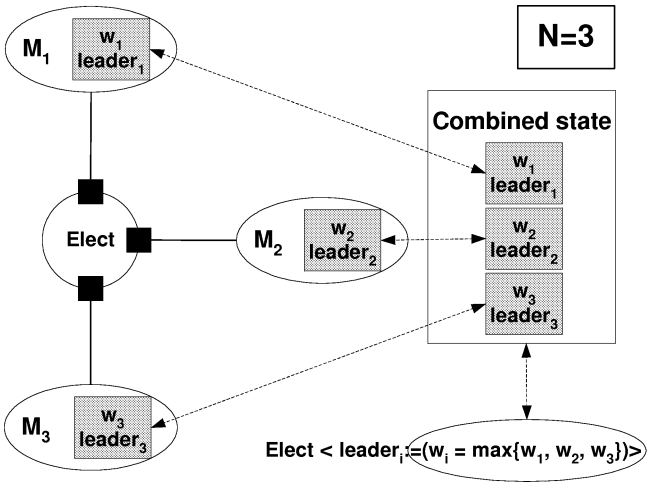


Figure 2.4: *A static view of the leader election problem.*

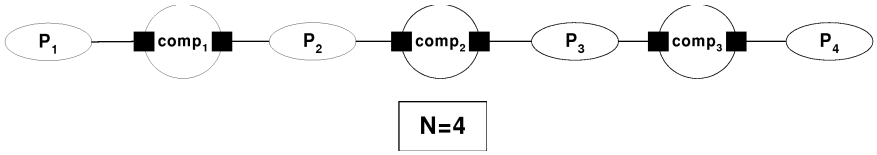


Figure 2.5: *A static view of the* OE SORT *algorithm.*

cation statement $leader_i := (w_i = \max_{1 \leq j \leq n}\{w_j\})$ allows each participant to compute the maximum weight in parallel and store the result in its local variable $leader_i$. After the interaction is executed, the one that finds itself having the maximum weight executes the appropriate algorithm.

Summing up, multiparty communication is a symmetric inter-process communication construct involving an arbitrary number of processes. This is a different view of communication, which is more traditionally broken into sending and receiving data.

2.4.3 The OE SORT algorithm

Assume that a system is composed of N processes, referred to as P_1, P_2, ..., P_n, which hold a set of values, referred to as a_1, a_2, ..., a_n, respectively. The goal is to coordinate them so that after a finite number of interactions to exchange their values, the final state satisfies the constraint $a_1 \leq a_2 \leq \ldots \leq a_n$. The OE SORT algorithm by Batcher [13] is very adequate in this scenario, and Tang and Muraoka used it to show that multiparty interactions can be used to describe sequential algorithms in a concurrent setting, and they allow to achieve optimal parallelism [135].

The OE SORT algorithm can solve the problem in $\frac{N}{2}$ dual phases, namely: in odd phases, odd participants compare and exchange data with their right, even neighbours; in even phases, even participants compare and exchange data with their right, odd neighbours. For instance, during an odd phase, P_1 and P_2, P_3 and P_4, P_5 and P_6, and so on would exchange their values if they are not ordered; during even phases, P_2 and P_3, P_4 and P_5, P_6 and P_7, and so on would do the same. It was proved that after k odd-even phases are executed, no process is further than $N - 2k$ steps from its final sorted state. Thus, when k equals $\lceil \frac{N}{2} \rceil$, all of the processes have reached their final states.

Program §2.3 shows an IP_{CORE} implementation inspired by Quinn's solution [114], and Figure §2.5 shows a static view when $N = 4$. Figure §2.6 illustrates an execution in which only two even-odd phases are needed to sort the values.

```
OESORT :: [||_{i=1}^{N} P_i], where
    P_1 :: { a_1: natural; result_1: natural; j: natural }
        a_1 := a data item;
        j := 1;
        *[ j ≤ ⌈N/2⌉ & comp_1⟨a_1 := min(a_1, a_2)⟩ → j := j + 1 ] ;
        result_1 := a_1;
    P_N :: { a_N: natural; result_N: natural; j: natural }
        a_N := a data item;
        j := 1;
        *[ j ≤ ⌈N/2⌉ & comp_{N-1}⟨a_N := max(a_{N-1}, a_N)⟩ → j := j + 1 ] ;
        result_N := a_N;
P_{i ∈ (1, N)} :: { a_i: natural; result_i: natural; j: natural }
        a_i := a data item;
        j := 1;
        *[ j ≤ ⌈N/2⌉ →
            [ odd(i) & comp_i⟨a_i := min(a_i, a_{i+1})⟩ → skip
            [] even(i) & comp_{i-1}⟨a_i := max(a_{i-1}, a_i)⟩ → skip
            ];
            [ even(i) & comp_i⟨a_i := min(a_i, a_{i+1})⟩ → skip
            [] odd(i) & comp_{i-1}⟨a_i := max(a_{i-1}, a_i)⟩ → skip
            ];
            j := j + 1;
        ] ;
        result_i := a_i.
```

Program 2.3: *An* IP_{CORE} *implementation of the* OE SORT *algorithm.*

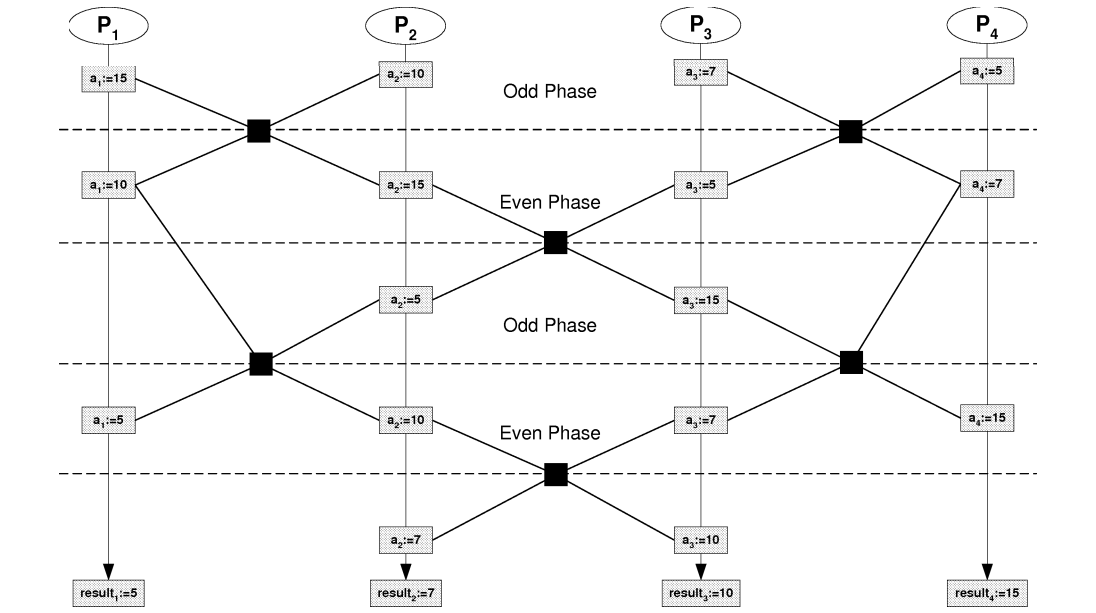


Figure 2.6: *An execution of the* OE SORT *algorithm.*

2.4.4 The ALOHA protocol

There exists many protocols to communicate workstations by means of a shared transmission medium that is not able to transmit more than one message at a time. The ALOHA protocol is a classical solution that deserves attentions since it put the foundations for the almost-universal Ethernet network [1, 134].

In the original ALOHA protocol, workstations communicate by means of a satellite. If only one workstation needs to transmit a message, then the satellite broadcasts the information to all of the workstations, including the sender; on the contrary, if several workstations need to transmit messages at the same time, the satellite is able to detect the collision and broadcasts a special message. When a workstation that was attempting to send a message receives this special message, it has to wait for a random period of time and try then to send its message again. Hopefully, this random delay detains the involved workstations so that they do not collide again.

This basic idea has many variants. One of the most interesting is the so-called SLOTTED ALOHA whose difference lies in that it does not allow the workstations to transmit their messages at arbitrary times, but at the beginning of fix-sized intervals called slots. This variation was proved to improve the efficiency of the ALOHA protocol around 35%.

```
SALOHA :: [Satellite || ||_{i=1}^{N} Station_i], where
  Satellite :: { broadcast: message; vector: array (1..N) of message }
               *[ true →
                   ascend⟨vector(1) := sent_1; ...; vector(N) := sent_N⟩;
                   [ collision(vector) → broadcast := COLLISION
                   [] ¬ collision(vector) → broadcast := extract(vector)
                   ];
                   descend⟨⟩
               ];
Workstation_i :: { sent_i, recv_i, save_i: message; waitfor_i: natural }
               sent_i := EMPTY; recv_i := EMPTY;
               *[ true →
                   [ sent_i = EMPTY → sent_i := prepare message ];
                   ascend⟨⟩; descend⟨recv_i := broadcast⟩;
                   [ recv_i = sent_i → sent_i := EMPTY
                   [] recipient(recv_i) = i → process message
                   [] recv_i = COLLISION ∧ sent_i ≠ EMPTY →
                       waitfor_i := a random number;
                       save_i := sent_i; sent_i := EMPTY;
                       *[ waitfor_i > 0 →
                            ascend⟨⟩; descend⟨recv_i := broadcast⟩;
                            [ recipient(recv_i) = i → process message ];
                            waitfor_i := waitfor_i - 1
                        ];
                       sent_i = save_i
                   ]
               ].
```

Program 2.4: *The* SLOTTED ALOHA *protocol.*

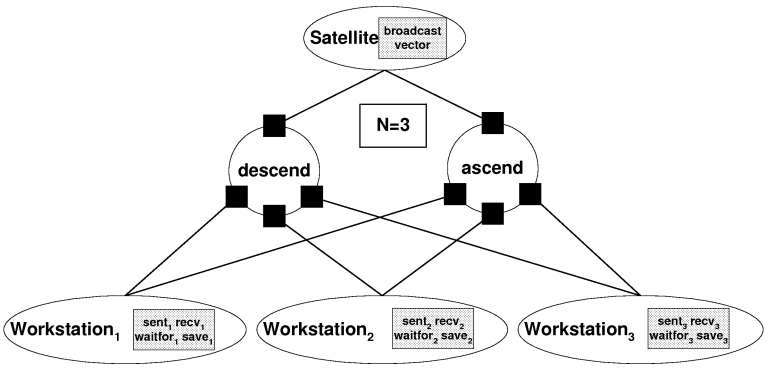


Figure 2.7: *A static view of the* SLOTTED ALOHA *protocol.*

Program §2.4 shows an IP_{CORE} implementation of this protocol, and Figure §2.7 shows its static view when $N = 3$. Note that we use two $(N + 1)$-party interactions, namely: one for the ascending link, which allows workstations to send messages to the satellite, and another for the descending link, which allows the satellite to broadcast messages to the workstations. For the sake of simplicity, we assume that there exists a data type called *message* with the obvious meaning. We also assume that there exist two special messages called *EMPTY* and *COLLISION*. The former denotes that a workstation does not need to transmit any messages, whereas the latter is used to denote that a collision was detected.

The satellite has a variable called *vector* in which it stores the messages it receives from the workstations. Its life cycle is simple since it first participates in interaction *ascend* and coordinates with the whole set of workstations so as to know what messages they are willing to transmit. Each workstation stores its messages in a variable called $sent_i$, so when the satellite executes the interaction command $\text{ascend}\langle \text{vector}(1) := \text{sent}_1; \ldots; \text{vector}(N) := \text{sent}_N \rangle$, it stores the message the corresponding workstations want to transmit in *vector*. A collision exists if two or more cells have a non-empty message, and the satellite uses function *collision* to detect these situations.

As for the workstations, their life cycle is a little more complex. First, they participate in interaction *ascend* to let the satellite know the messages they are willing to transmit. Immediately after, they participate in interaction *descend* and capture the message the satellite has broadcast. If the message they receive coincides with the message they sent, this means that no collision has occurred and the original message has been delivered to the recipient. Otherwise, if the message they receive is *COLLISION* and the message they sent was not empty, this implies that other workstations attempted to transmit their messages at the same time. In this case, the involved workstations wait for a random number of slots to elapse. Note that they have to clear $sent_i$ or otherwise the satellite would think that they are attempting to retransmit their messages immediately. In order not to lose those messages, an auxiliary variable called $save_i$ is needed.

2.5 Summary

In this chapter, we have motivated the need for multiparty interactions. There exists a variety of models and languages that support these primitives and differ on the degree of multipartiness and abstractness they support. We have presented a taxonomy that builds on Joung's original proposal and updates it with current, state-of-the-art languages. Finally, we have presented

several examples that may help realise the benefits of modelling a system using multiparty interactions.

We have published several results on this topic, namely: we presented our preliminary results on implementing IP_{CORE} by means of a centralised, general, strongly fair scheduler at the SEID'99 Conference [126] and the IEEE Euromicro'99 Conference [44]; in ACM Sigplan [39], we presented our preliminary ideas on the design of a language able to support multiparty interactions in an aspect-oriented manner; these ideas were further refined and published in Computer Networks [38]; in the ACM SAC'02 Conference [110], we presented a framework to support a multiparty interaction model in which the participants of an interaction need not be known beforehand and may change at run time; the details on how to achieve mutual exclusion in this model were presented at the COORDINATION'02 Conference [111] and the complete algorithm to schedule such interactions was published in Concurrency and Computation: Practice & Experience [40]. Our preliminary attempts to devise a practical fairness notion in this context were presented at the Euro-Par'02 Conference [122] and The Colombian Journal of Computation [124]. Our final results on fairness were published in Concurrency and Computation: Practice & Experience [125].

Chapter 3

Fairness

Fairness does not consist so much of everybody's doing the same thing, but of everybody's being willing to do something that others don't want to do.

Judith Martin, 1938–
Washington Post columnist

In this chapter, we focus on fairness and argue on why it is a desirable property in the context of multiparty-based systems. It is organised as follows: In Section §3.1, we introduce the concept; in Section §3.2, we define safety and liveness properties and illustrate them with some examples; several taxonomies of fairness are shown in Section §3.3; in Section §3.4, we present a couple of frameworks for defining fairness; we present a discussion on some controversial aspects of fairness in Section §3.5; finally, we summarise the chapter and our main contributions to this field in Section §3.6.

```
x := 0; goon := true;
*[ goon → x := x + 1
[] goon → goon := false
].
```
(a)

```
SYSTEM :: [Provider || Client_A || Client_B], where
Provider :: *[ A⟨⟩ → skip [] B⟨⟩ → skip ];
Client_A  :: *[ A⟨⟩ → skip ];
Client_B  :: B⟨⟩.
```
(b)

Program 3.1: *Fairness to ensure termination and eventual response to a request for service.*

3.1 Introduction

Usual fairness notions were introduced in the context of Dijkstra's language [49] and Petri Nets [32]. Since then, it is very usual to refer to fairness as if it was a single concept. However, there exists a rich lattice of notions that are not equivalent to each other, but share a constant theme: they attempt to restrict the set of potential executions of a system by disallowing those in which a process is forever denied its rightful opportunity to proceed [102]. This allows programmers to abstract away from low-level details since they can assume that their processes will progress independently from the relative speed of the processors, the latency of the network or the scheduling policy.

Consider, for instance, Program §3.1.(a). To determine whether it terminates or not, we need to know how the compiler implements multi-choice commands. Theoretically, if several choices can be executed, the election is non-deterministic, but how non-determinism is implemented depends on the compiler and the underlying run-time support. If we know that the implementation ensures that the probability of selecting each choice is 0.5, the program should terminate in finite, but unbounded, time; contrarily, if the criterion is to select the first enabled choice, the program shall never stop. The scenario is similar in Program §3.1.(b). In this case, there is a provider process that can serve either $Client_A$ or $Client_B$ by participating in interactions A or B, respectively. Unfortunately, unless we know how the program is implemented, we cannot prove that $Client_B$ shall ever have an chance to be served.

The things change dramatically as soon as we know that the compiler implements a fairness notion. In such case, Program §3.1.(a) must terminate and $Client_B$ must be served in finite, but unbounded time since no choice may be denied a rightful chance to be executed forever. In this case, fairness allows us to prove two liveness properties that we would not have been able to prove otherwise. Furthermore, fairness can also help us prove some safety properties, since it is a well-known result that fairness is essential in systems in which their participants need to get mutual exclusion to a resource, which is a

Property	Bad thing
Mutual exclusion	Two processes enter their critical sections at the same time.
Deadlock freedom	A set of processes wait for each other to have access to the same resource.
Partial correctness	The initial state satisfies a precondition, but the final state does not satisfy the corresponding postcondition.
First-come-first-served	Two requests for service are not granted in the same order as they arrived.

Table 3.1: *Examples of safety properties.*

Property	Good thing
Progress	Every process progresses and executes local computation eventually.
Termination	The final command of a process is executed eventually, *i.e.*, it arrives a fixed point in which it can neither execute local computation nor interact.
Starvation freedom	Every request for service is granted eventually.

Table 3.2: *Examples of liveness properties.*

safety property [79].

3.2 Safety and liveness properties

In Reference [82], Lamport introduced some properties that deserve special attention in the context of concurrent systems. He classified them into two groups, namely: *safety properties*, which assert that "*something bad*" does not happen, and *liveness properties*, which assert that "*something good*" must happen eventually. Tables §3.1, §3.2, and §3.3 present some usual safety and liveness properties.

The distinction between them is very important since proving that a program satisfies a particular safety property is equivalent to proving the fulfilment of an invariant; on the contrary, proving that a program satisfies a liveness property is usually equivalent to finding a well-founded order in which a ranking function decreases or keeps constant at every computational step. In summary, safety properties may be viewed as features of individual states, whereas liveness properties can only be viewed as properties of executions.

Safety	Liveness
Every sentence is executed in less than 10 milliseconds.	Every sentence is executed in finite time.
The program finishes in 1.7E24 centuries.	The program shall finish eventually.
No interaction is rejected more than 5 times in a row.	No interaction is rejected forever.

Table 3.3: *Sample safety properties and their corresponding liveness properties.*

Since they were introduced informally by Lamport, many authors have focused on characterising them formally, namely: Alford *et al.* characterised an execution of a concurrent system as an infinite sequence of states, and a property as a subset of such sequences [2]; Alpern and Schneider characterised them by means of structural properties of Büchi automata [3]; Sistla focused on structural properties of temporal formulae, which was later found to be equivalent to the previous proposal since there exist automatic procedures to translate linear or branching temporal formulae into their equivalent Büchi automata [33]; Lichtenstein *et al.* presented another characterisation based on temporal formulae that does not take past operators into account.

We think that Alpern and Schneider's proposal is the most suitable in the context of concurrent systems because it was proved that there exist properties that can be described by means of Büchi automata, but cannot be specified with temporal logic [140]. Their definition follows [3]:

> *Let S be the set of program states, S^∞ the set of infinite sequences of programs states, and S^* the set of finite sequences of programs states. The elements of S^∞ are called* executions*, and the elements of S^* are called* partial executions*. A property is a subset of S^∞. By $\lambda \models P$, we denote that execution λ satisfies P, i.e., it belongs to the subset of executions defined by this property. Property P holds for a program if the whole set of executions defined by the program is contained within the property. Finally, let λ_i denote the partial execution consisting of the first i states in λ. P is a safety property iff*
>
> $$\forall \lambda \in S^\infty \cdot (\lambda \not\models P \Rightarrow \exists i > 0 \cdot \forall \beta \in S^\infty \cdot \lambda_i \beta \not\models P).$$
>
> *On the contrary, it is a liveness property iff*
>
> $$\forall \alpha \in S^* \cdot \exists \beta \in S^\infty \cdot \alpha\beta \models P.$$

Note that the definition of safety property unconditionally prevents "bad things" from happening because, otherwise, there must exists an identifiable point at which such things can be detected. On the contrary, the definition of liveness property indicates that no partial execution is irremediable since it is always possible for them to be extended to infinite executions that preserve the liveness property under consideration. Is is also worth noting that according to this definition, a liveness property stipulates that a "good thing" must happen eventually, but not the time at which it happens. This is a well-known phenomenon which is usually referred to as *unbounded non-determinism* [102].

Although Alpern and Schneider's characterisation of liveness is widely accepted, there are other complementary definitions which are more restrictive, *e.g.*, *uniform liveness* and *absolute liveness* [4]. A property is a uniform liveness property iff there is a single suffix that can be used to extend any partial execution so that the result preserves the property under consideration; on the contrary, it is an absolute liveness property iff it is non-empty and any execution which satisfies it can be appended to any partial execution so that the result still satisfies the property. Unfortunately, absolute fairness does not include properties such as eventual response to a request for service, and uniform liveness does not capture the intuition of liveness.

3.3 Taxonomies of fairness notions

As we mentioned in the introduction, the constant theme behind fairness notions is that no artefact should be denied forever a rightful opportunity to proceed [102]. In other words, every *element* of a system that may cause an artefact to proceed and is *sufficiently often* enabled for execution must be executed *sufficiently often*. Different notions of fairness are possible by giving different meanings to the terms "element" and "sufficiently often".

In the context of multiparty interaction models, the most common "elements" are the interactions themselves, so it is very usual to study *interaction fairness*. There are some authors who also take *process fairness* into account, *i.e.*, they consider the artefacts of the system themselves. In this dissertation we focus on interaction fairness since every execution that is interaction-fair, is also process-fair. The reason is that we assume that processes synchronise and exchange information by means of multiparty interactions only, which implies that every time an interaction is selected fairly, the processes that participate in it proceed fairly, too. Obviously, the definition of interaction fairness assumes that the *minimal progress property* is implemented so that every process that can execute a local action on its local state shall eventually execute it.

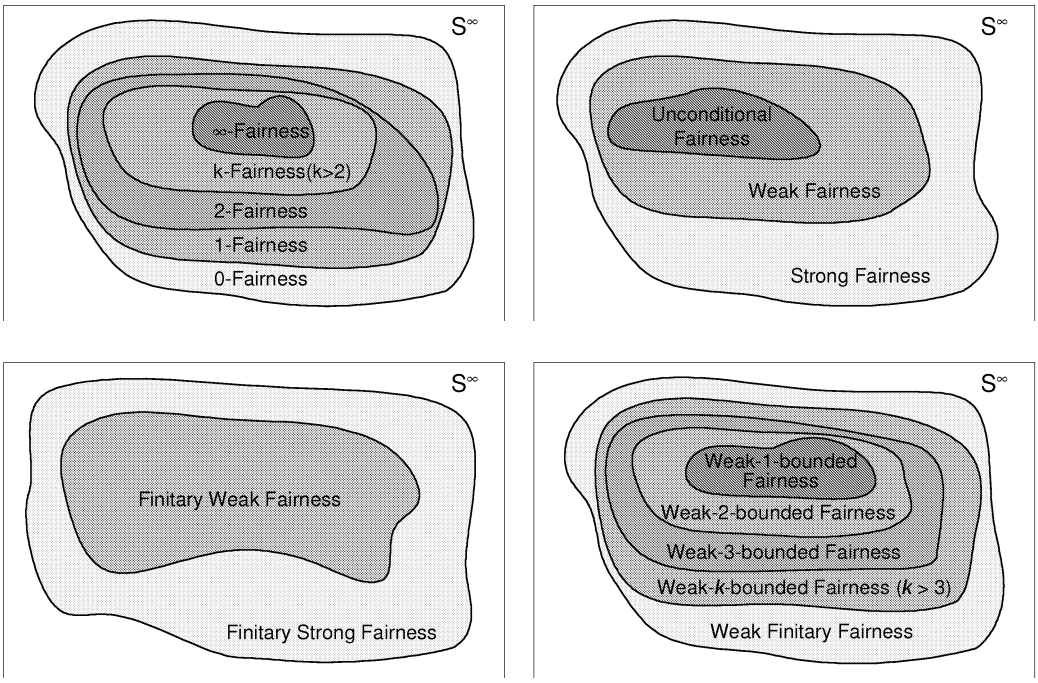


Figure 3.1: *Relationships amongst different fairness levels.*

Furthermore, each of these notions have several *strictness levels* depending on the definition of "sufficiently often". In the following subsections, we present several taxonomies to give the reader an overall idea of the rich lattice of existing fairness levels in widespread use. The known relationships between some of these levels are sketched in Figure §3.1. Unfortunately, there are some levels whose relationships to the rest are not known, yet, but the details fall beyond the scope of this dissertation.

3.3.1 Best's taxonomy

Best considered the possible occurrence of events as the basic property to distinguish between different levels [19, 18, 20], and designed a taxonomy that builds on Chrząstowski-Wachtel's previous work on Petri nets.

To understand his taxonomy, which is illustrated in Figure §3.2, it is necessary to introduce the concept of k-*activation*: it is said that a partial execution k-activates an interaction iff it can be extended so that it is possible for that interaction to become enabled before $k + 1$ other interactions are executed; in general, it is said that a partial execution ∞-activates an interaction iff there

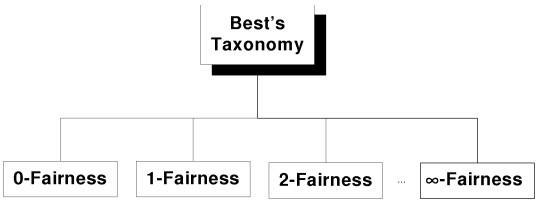


Figure 3.2: *Best's taxonomy of fairness.*

exists a $k \in \mathbb{N}$ so that it k-activates this interaction. This concept allowed him to define k-*fairness* for any $k \in \mathbb{N} \cup \{\infty\}$ as follows: an execution is k-fair iff every interaction that is k-activated infinitely often is executed infinitely often.

Note that 0-activation of an interaction implies that it is detected to be enabled immediately, whereas k-activation for $k \geq 1$ implies that it can possibly become enabled before $k+1$ other interactions are executed. This implies that k-fairness does not depend on an execution itself, but on the possible extensions of its finite prefixes. In the case of ∞-fairness, this observation has motivated some authors to state the notion as follows [10, 78, 83]: an execution is ∞-fair iff every interaction that can become enabled infinitely often, becomes enabled infinitely often and, thus, is executed infinitely often. Note the subtle intricacy in this sentence since the notion does not takes into account the execution under consideration, but the possibility for an interaction to have become enabled if the execution had taken another path. This is the reason why some authors tend to refer to these levels as *oracle-based levels*; furthermore, there does not exist a reasonable general scheduler to implement k-fairness if $k \geq 1$, since it should be able to forecast if an interaction can become enabled in the future. Note, however, that some implementations are possible in some restrictive cases that are explained in Section §4.4 and References [10, 78, 83].

It is also worth mentioning that 0-fairness is also known as the *compassion property* [88] or *strong fairness* [59], and that it is one of the most practical levels. The subset of ∞-fairness that has been implemented in the context of multiparty interactions is usually referred to as *hyperfairness* [10, 83].

3.3.2 Francez's taxonomy

Francez's taxonomy is organised into two main structural categories according to the subset of interactions that is subject to fairness, namely [59]: if fairness is applied to the whole set of interactions, then the notion is said to be *all-levels fair*; on the contrary, if only normal-form programs are considered

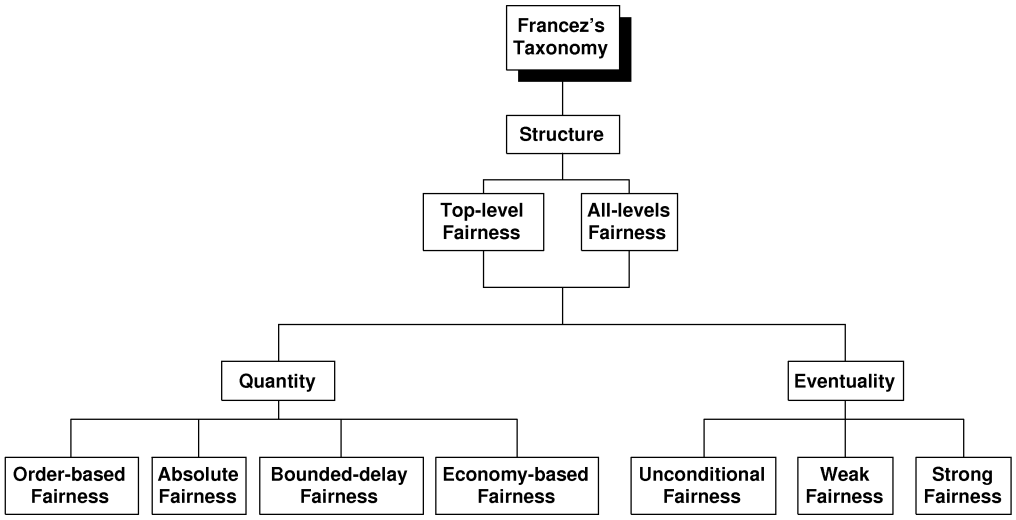


Figure 3.3: *Francez's taxonomy of fairness.*

and fairness is applied to top-level interactions only, then the notion is said to be *top-level fair* (*cf.* Section §B.4).

Independently from the structure, Francez further distinguishes between the quantitative-based and the eventuality-based levels that are sketched in Figure §3.3. The idea behind the former is to relax pure eventuality and take the relative ordering in which interactions are executed into account; on the contrary, the idea behind the latter is to disallow executions in which an interaction that is enabled infinitely many times is executed finitely many times.

Although *quantitative-based levels* seem very adequate, they are not common in practice since they have not attracted the attention of many researchers. They include the following: *order-based fairness*, which ensures that an interaction is executed as soon as it becomes enabled and has mutual exclusion over the set of interactions with which it shares a participant [113]; *absolute fairness*, which ensures that the relative order in which interactions are executed permutes infinitely often; *bounded-delay fairness*, which aims at setting a (hopefully small) bound on the length of the intervals between successive executions of the same interaction; and *economy-based fairness*, whose goal is to avoid the number of executions of every interaction from diverging too much from an ideal number that is calculated according to an economic criterion.

In contrast, *eventuality-based levels* have attracted the attention of many researchers. They include the following: *unconditional fairness*, which ensures that an interaction that is enabled *permanently* is executed infinitely often;

weak fairness, which is more relaxed and does not require an interaction to be executed infinitely many times unless it is continuously enabled from a point in the execution onwards; *strong fairness*, which is the most restrictive since it does not require an interaction to be executed infinitely often unless it is enabled infinitely many times, not necessarily in a row. In spite of the eventuality-based levels being so common in practice, their names are not universal. For instance, Lehmann *et al.* refer to unconditional and weak fairness as *the justice* and *the minimal progress properties*, respectively [88], and Best refers to strong fairness as 0*-fairness*.

Note that the application of these levels to multiparty interactions depends completely on the semantics under considerations. Since there can be several equivalent semantics for the same model, this implies that there exists a tight relationship between them and the applicable notions. For instance, in our framework (*cf.* Chapter §6), the unconditional and the weak levels are not applicable since no interaction may be enabled continuously. The reason is that once an interaction is executed, it becomes disabled and the participants need to offer to participate in it again; this implies that some time elapses since the moment an interaction is executed until the moment it becomes enabled again, and this time is not ignored in our framework. However, if we consider IP original semantics [58], an interaction may be enabled continuously since a process being ready to participate in an interaction is not considered an independent event. Thus, although some time may elapse between consecutive enablements of the same interaction, it is not taken into account.

3.3.3 Alur and Henzinger's taxonomy

The main problem with Francez's eventuality-based levels is that they do not put a bound on the number of times an interaction may be neglected. The relationships between such notions and *unbounded non-determinism* are well known in the literature [102], and they motivated Alur and Henzinger to devise a new hierarchy of levels [5] that is illustrated in Figure §3.4.

Their taxonomy distinguishes between the *finitary* and the k*-bounded levels*, which are abstract in the sense they need to be applied to other levels. Although the authors provided a general characterisation by means of which any notion can be restricted to be finitary of k-bounded, they focused on the weak and strong levels only. They defined an execution to be weakly-k-bounded iff no interaction can be neglected more than k times without being executed; on the contrary, it is strongly-k-bounded iff any subsequence in which an interaction has been enabled k times contains an execution of that interaction. Building on these concepts, an execution is said to be finitarily weak-fair iff there

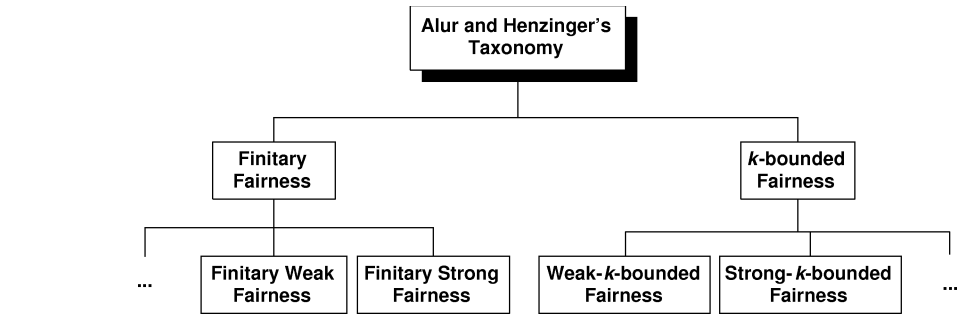


Figure 3.4: *Alur and Henzinger's taxonomy of fairness.*

exists a $k \in \mathbb{N}$ such that no interaction is postponed more than k consecutive times; similarly, an execution is said to be finitarily strong-fair iff there exists a $k \in \mathbb{N}$ such that any subsequence in which an interaction has been enabled k times contains an execution of that interaction. Note that the only difference lies in the fact that k must be known beforehand in the former, whereas it is unknown in the latter.

The bounded level has long been considered part of the folklore until this characterisation appeared in the literature [46, 69]. Recently, Dershowitz *et al.* have presented further results on this topic in Reference [46]. They build on a previous proposal that was presented in Reference [69]. Independently, Beauquier *et al.* [16] introduced a notion called daemon-k-fairness that coincides with Alur and Henzinger's weak-k-bounded fairness.

3.3.4 Examples of fair and unfair executions

Our goal in this section is to illustrate some of the previous levels by means of the examples in the program box number §3.2. For the sake of simplicity, we have used uniparty interactions only; furthermore, we focus solely on the interactions that are executed, *i.e.*, the interaction traces, since it is not necessary to dig into low-level details concerning the states of the processes or the events they produce.

Example 3.3.1 *An execution that produces the interaction trace* $[a_1]^\infty$ *is not unconditionally fair if we take* IP *original semantics into consideration. The reason is that interaction* a_2 *is always enabled but it is never selected. Consequently, this execution is neither weak, nor* k*-fair for any* $k \in \mathbb{N}$*, nor* ∞*-fair. More precisely, there are infinitely many prefixes that* k*- or* ∞*-activate* a_2*, but it is not executed infinitely many*

EXAMPLE §3.3.1 ::
```
*[ true & a1⟨⟩ → x := x + 1
[] true & a2⟨⟩ → x := x - 1
].
```

EXAMPLE §3.3.2 ::
```
goon := true; A := true; B := true;
*[ goon & a1⟨⟩ → A := ¬A
[] goon & a2⟨⟩ → B := ¬B
[] ¬A ∧ ¬B & a3⟨⟩ → goon := false
].
```

EXAMPLE §3.3.3 ::
```
goon := true; x := 1;
*[ goon & a1⟨⟩ → x := x + 2
[] goon & a2⟨⟩ → x := x - 1
[] x ≤ 0 & a3⟨⟩ → goon := false
].
```

EXAMPLE §3.3.4 ::
```
goon := true; x := 0;
*[ goon & a1⟨⟩ → x := 1
[] x = 0 & a2⟨⟩ → goon := false
].
```

Program 3.2: *Sample programs to show fair and unfair executions.*

times. If we take the semantics of our framework into account, the unconditional or the weak levels are not applicable.

On the contrary, an execution that produces the interaction trace $[a_1a_2]^\infty$ fulfils the requirements of all oracle- or eventuality-based notions because both a_1 and a_2 are infinitely often 0-, k- and ∞-activated, and they both are executed infinitely many times. Obviously, it is also finitarily fair at any level since the bound on the number of times each interaction is rejected does not exceed one.

Example 3.3.2 *Any execution whose interaction trace is $[a_1a_1a_2a_2]^\infty$ is 0-fair and 1-fair but not k-fair for any $k \geq 2$, and, consequently, not ∞-fair. Note that, interaction a_3 is never enabled, and this is the reason why it is never executed. However, a_3 can become enabled after the selections of choices a_1 and a_2, since there exists infinitely many prefixes of this execution that can be extended by two interaction executions yielding a sequence in which a_3 can be executed.*

This is a clear example of the theoretical interest of these notions since it is impossible to implement a scheduler able to forecast that the guard of interaction a_3 can become enabled if interactions a_1 and a_2 are executed in this or the converse order.

Example 3.3.3 *Any execution whose interaction trace is $[a_1a_2]^\infty$ is k-fair for every $k \geq 0$, but not ∞-fair. Note that interaction a_3 is never executed, but every finite*

prefix of this execution can be extended so as to activate a_3 by executing interaction a_2 enough times; however, this interaction cannot be executed infinitely many times.

Example 3.3.4 *Any execution whose interaction trace is $[a_1]^\infty$ is k-fair for every $k \geq 0$ as well as ∞-fair. Now, the guard of interaction a_2 cannot be enabled after interaction a_1 is executed.*

3.4 Frameworks to define fairness

Although several theoretical frameworks to deal with fairness have been presented in the literature, we think that the original proposals by Olderog and Apt [103] and Lamport [83] are probably the most mature. In the following subsections, we present a brief introduction to them.

3.4.1 A transitional framework

To the best of our knowledge, Olderog and Apt were the first to introduce a framework to deal with fairness in the context of parallel programming languages with shared variables [103]. In order to simplify the study, they defined fairness in terms of enabled and selected elements, *i.e.*, interactions in our context. Thus, an execution is a sequence of pairs of the form (E, i), where E is a non-empty set of indexes that denote the interactions that are enabled, and $i \in E$ denotes the index of the interaction selected for execution. With this simple model, they defined two predicates that hold with *weakly* or *strongly fair* executions, namely:

$$WF(\lambda) \iff \forall i \in \{1, \ldots, n\} \cdot (\forall^\infty j \in \mathbb{N} \cdot i \in E_j) \Rightarrow (\exists^\infty j \in \mathbb{N} \cdot i = i_j)$$
$$SF(\lambda) \iff \forall i \in \{1, \ldots, n\} \cdot (\exists^\infty j \in \mathbb{N} \cdot i \in E_j) \Rightarrow (\exists^\infty j \in \mathbb{N} \cdot i = i_j)$$

where λ denotes the generic execution $[(E_0, i_0)(E_1, i_1) \ldots (E_j, i_j) \ldots]$.

As for the implementation, they presented a transformational method that builds on a previous proposal by Dijkstra [47]. It allows to transform any program into its equivalent weakly or strongly fair form, which may be problematical insofar as most current systems are deployed in the form of binary components whose source code is not generally available [133]. The transformation assigns a natural counter to each interaction, and initialises them to a positive random value. The criterion is to select the enabled interaction whose

counter has the minimum value, reset its counter to a new positive random value, and decrease the counter of the remaining interactions by one.

Note that the implementation of this framework requires the status of the whole set of interactions in a system to be known before reaching a decision, which may have a negative impact on the efficiency.

3.4.2 A framework based on temporal logic of actions

In Reference [83], Lamport got inspiration from Francez and Gabbay *et al.* to use *temporal logic of actions* to define fairness notions [60, 61]. This approach is more suitable and more powerful than Olderog and Apt's, but it is far from an efficient implementation [57].

There exists two versions depending on whether *linear* or *branching temporal logic* is used. The former has special connectives that hide the explicit quantification over time and make more intuitive and compact many program properties; on the contrary, the latter builds on trees instead of sequences, and its advantages are not clear enough in the research community.

Therefore, the linear version is the most common. Its time connectives are denoted as $\Box$ and $\Diamond$. If F is a logic formula, $\Box F$ means that F holds always, and $\Diamond F$ means that it must hold eventually. An execution is thus said to be weakly fair iff

$$\Diamond\Box enabled(x) \Rightarrow \Box\Diamond selected(x)$$

holds for every interaction x. Similarly, an execution is said to be strongly fair iff

$$\Box\Diamond enabled(x) \Rightarrow \Box\Diamond selected(x)$$

holds for every interaction x.

To the best of our knowledge, there is not a general implementation of this framework since the temporal connectives on which it builds are not implementable. They rely on eventuality and infiniteness, which are very adequate to express complex liveness and safety properties, but impossible to implement on finite machines.

3.5 Controversy on fairness

Although fairness seems to be very desirable from a practical point of view, there are some intricacies that have led many researchers to criticise it. In some

Notion	Sample executions allowed
Strong	$(a_1 \mid a_2)^\infty$
Finitary strong	$\bigcup_{i \in \mathbb{N}} (a_1^{\{0,i\}} a_2 \mid a_1 a_2^{\{0,i\}})^\infty$
k-bounded strong	$(a_1^{\{0,3\}} a_2 \mid a_1 a_2^{\{0,3\}})^\infty$

Table 3.4: *Differences amongst classical, finitary and bounded notions.*

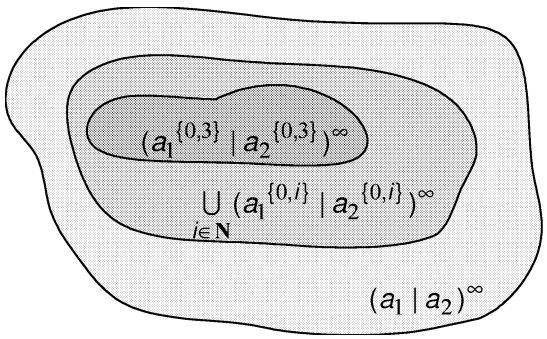


Figure 3.5: *Relationships amongst some sample strong, finitary, and bounded executions.*

cases, arguments to discard it altogether have been presented. In the following subsections, we address the most common criticisms.

3.5.1 Is fairness a safety or a liveness property?

The goal of unconditional, weak, strong and k-fairness is to avoid artefacts that are ready to progress sufficiently often from blocking forever. Note, however, that this does not entail that there is an upper bound to the time an enabled artefact may be delayed before it can proceed, and this is the reason why the finitary and k-bounded levels were introduced.

There is, however, an important difference amongst k-bounded fairness and the remaining notions since the former is a safety property whereas the latter are liveness properties. The reason is that k-bounded fairness is not a feature of the whole execution, but of individual states since it is possible to count the number of times an interaction has been selected to the detriment of another. Therefore the fulfilment of this property can be checked at every computational step. If there is a bound on the time the artefacts of a system spend

b := true; *[b → print(0) [] b → print(1); b := false].	(a) Original program.
print(1).	(b) Implementation 1.
*[true → print(0)].	(c) Implementation 2.

Program 3.3: *Dijkstra's example to show that fairness is a void property.*

at performing local computation, k-bounded fairness can thus be viewed as a materialisation of the *bounded-delay level* in Francez's taxonomy, which is a safety property, too.

Table §3.4 illustrates the differences amongst some classical, finitary, and bounded notions, and Figure §3.5 shows their relationships graphically. Note that the first and the second samples are liveness properties since they ensure that the occurrence of a_1 or a_2 must happen eventually; on the contrary, the third sample is a safety property since something bad happens when an a_1 or a_2 are rejected more than three times.

In summary, since fairness is not a unique concept, but a collection of abstractions, it can be viewed as a safety or a liveness property depending on the level under consideration.

3.5.2 Can fairness be ignored?

Since the violation of a safety property may be detected in finite time [3], this question makes sense in the context of fairness notions that are liveness properties only. In general, since they rely on the future and they describe features of a whole execution instead of individual states, they cannot be checked in finite time. (The only exception is the so-called class of *finitary liveness properties* [81], but none of the usual fairness notions lie in this category.)

On account of the previous problem, Dijkstra argued that fairness can be ignored with impunity because implementing it is a *void obligation* [50]. For instance, if we assume that the multi-choice command in Program §3.3.(a) is implemented fairly at any eventuality-based level, then we should conclude that it prints a random number of zeroes and stop after printing a single one. However, both Program §3.3.(b) and §3.3.(c) are *right implementations* and they do not behave this way since the former never prints a zero, and the latter never stops. By right implementation, Dijkstra meant that we can perform finite experiments to demonstrate its correctness or, otherwise, we cannot perform any finite experiments to demonstrate its weakness.

x := 1.	(a) Original program.
*[true → **skip**].	(b) Implementation 1.

Program 3.4: *Schneider and Lamport's counterexample.*

We translated the previous program into several languages that support multiple-choice commands, and the results were always the same: it seemed to loop forever and it did not print a number other than zero. We obviously were disappointed that our program never stopped, but what we might consider a bug, would not cause the programmer who implemented the compiler a single sleepless night since non-determinism is a void obligation, *i.e.*, an obligation nobody can prove it has not been fulfilled by means of finite, black-box experiments.

Therefore, according to Dijkstra, there is no need to implement fairness since nobody is able to prove that it is not implemented, and the distinction between fair and unfair executions is then meaningless. This argument was refuted by Schneider and Lamport using the example in Program §3.4 [131]. According to Dijkstra's reasoning, an assignment sentence can be rightly implemented with an unbounded waiting sentence, because no black-box, finite experiment can demonstrate that the assignment is not going to be performed. In general, proving the termination of a deterministic program is undecidable [102], but very few would argue that the distinction between terminating and non-terminating programs is meaningless.

Broy and Nelson [25] looked deeper into the problem of fairness and unbounded non-determinism, and concluded that it becomes realistic and meaningful as long as one thinks about fair programs not as defining an algorithm, but as defining an algorithm for every scheduler that is correct regarding the fairness assumption under consideration.

If is also worth mentioning that in spite of fairness being a void obligation, there is usually a clear behavioural distinction between programs that are controlled by a fair or an unfair scheduler. To illustrate this point, we use a benchmark that consists of running the dining philosophers system presented in Program §2.1 and studying how lunches are distributed amongst the participating philosophers. Note that if they spend the same time at thinking and eating, they all should have the same opportunities to have lunch, which implies that the standard deviation should be very small or null. In Figures §3.6 and §3.7, we present the results of running this benchmark with twelve philosophers for the duration needed to schedule 500 lunches. The details about the schedulers may be found in Chapter §4 and a detailed exper-

imental study in Chapter §9.

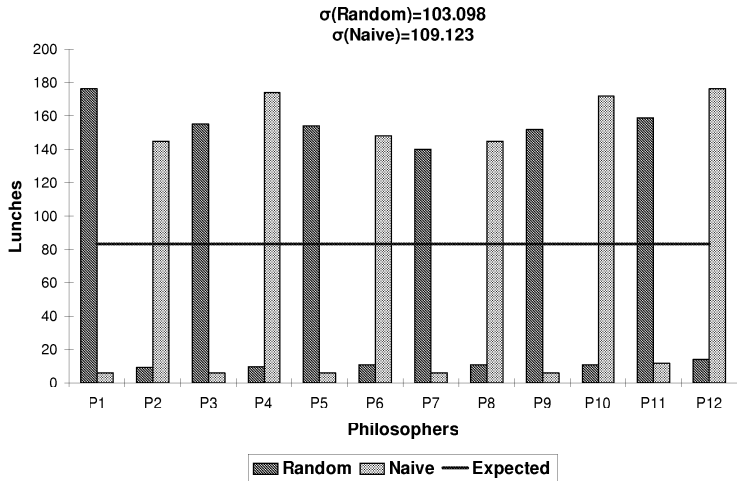
In Figure §3.6.(a), we present the results with a couple of unfair schedulers called Random and Naive. Both select interactions as soon as they become enabled, but the former solves conflicts by means of a random number generator, whereas the latter selects the first enabled interaction it finds and does not attempt to solve conflicts at all. As the figures show, the standard deviation they produce is big and there are some philosophers that do not have almost any lunches in the case of the Naive scheduler. In Figures §3.6.(b), §3.7.(c), and §3.7.(d) we present the results of the same experiment using the strong interaction-fair schedulers by Francez and Forman [58], Corchuelo [35] and Joung [72]. The difference is obvious, since as soon as the scheduler implements strong fairness the standard deviation is smaller, which is quite desirable in this scenario.

Note, however, that all of the executions on which we have reported are strong interaction-fair since they all are finite. It is also important to take into account that this behaviour depends completely on the random number generators used. For instance, if we use a random number generator that tends to produce numbers in the range $[10000, 50000]$ more likely than numbers in the range $[1, 10]$, both Francez and Forman's or Corchuelo's scheduler would produce executions very similar to the executions obtained with Random or Naive (*cf.* Figure §3.8). The difference is that fair schedulers can guarantee that no interaction is neglected forever, whereas unfair schedulers cannot. Since it would be desirable for the philosophers to have the same opportunities to have lunch in our benchmark, it is obvious that a careful tuning of the random number generators used in the schedulers may help achieve this goal. That is not possible in the case of unfair schedulers.

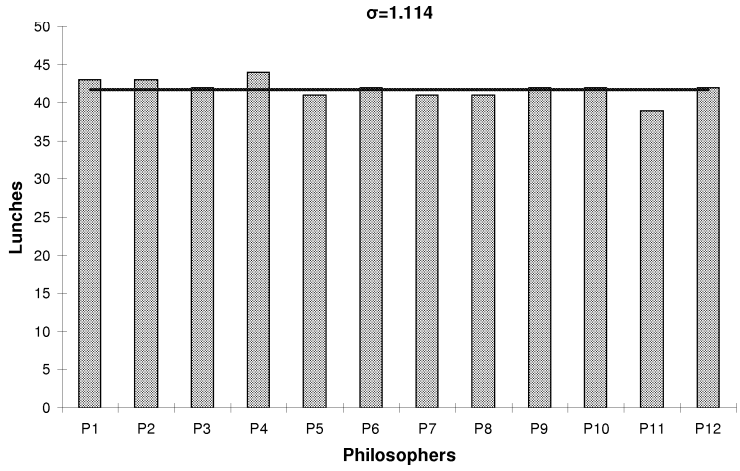
3.5.3 Are fair implementations complete implementations?

According to the semantics of the multi-choice command presented in Reference [58], it is non-deterministic. Note that, strictly speaking, non-determinism does not implies any liveness or safety property. There is only a promise to select amongst enabled choices arbitrary, so we should be willing to accept any outcome. In other words, the possible outcomes of a Program §3.1, for instance, should be "x is assigned a random natural value or the program loops", and they both are correct. Note that implementing the program fairly implies that the implementation prevents the second outcome from happening, which, in turn, implies that the implementation is not complete.

Some authors have argued that fairness should not be part of the implementation or the language, since it is the programmer's responsibility to prove

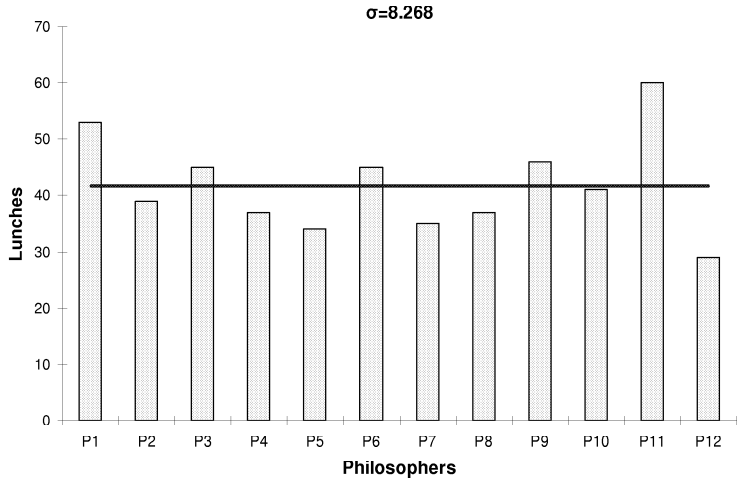


(a) Unfair schedulers.

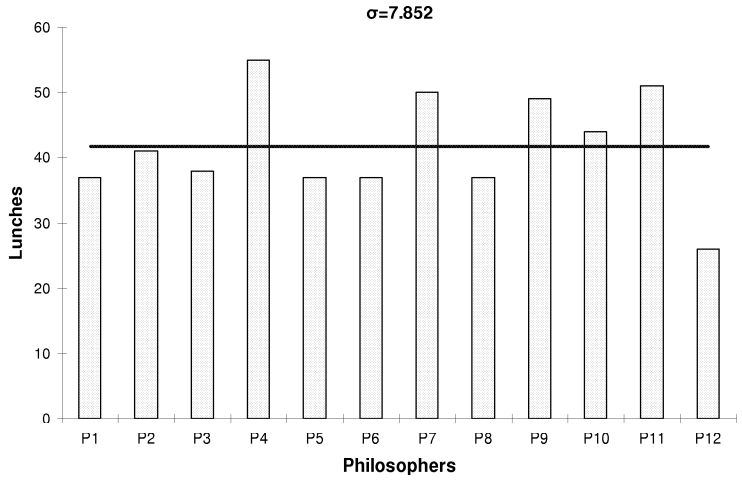


(b) Francez and Forman's strong interaction-fair scheduler.

Figure 3.6: *Distribution of lunches with both fair and unfair schedulers (I).*



(c) Corchuelo's strong fair scheduler.



(d) Joung's strong interaction-fair scheduler.

Figure 3.7: *Distribution of lunches with both fair and unfair schedulers (II).*

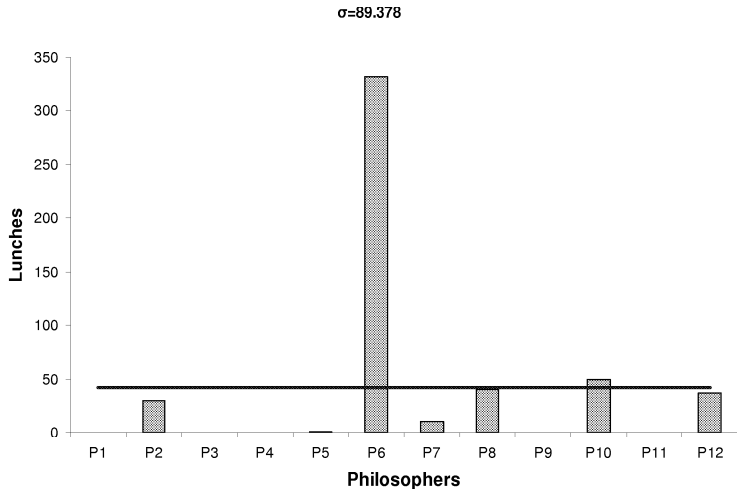


Figure 3.8: *Distribution of lunches with Francez and Forman's scheduler using a bad random number generator.*

that his or her programs are correct without relying on any properties of the implementation, but only on the semantics of the language [102]. However, this is impractical since proving that a program terminates, for instance, implies proving that it satisfies a liveness property, at least, which requires in-depth knowledge of how the program is implemented unless we assume that the scheduler is fair so that we can abstract away from those details.

Broy and Nelson [25] settled the discussion and concluded that the best idea is to consider both non-deterministic and fair multi-choice commands separately. The implementation of non-deterministic commands must guarantee that all possible outcomes may be obtained, whereas the implementation of fair commands must guarantee that no choice is neglected forever if it is enabled sufficiently often.

They designed a fair multi-choice operator called *dovetail operator*, which is denoted as ∇ and can be easily incorporated into IP_{CORE}. Using this operator, the semantics of the following program are "x is assigned a random natural number" and no implementation may loop forever or, otherwise, it would be incorrect:

```
x := 0; goon := true;
*[ goon → x := x + 1
∇ goon → goon := false
 ].
```

The main problem with considering operators such as ∇ is that the semantics of fairness are built in the language itself. Since there is not a single prevailing definition and they are not equivalent to each other, this implies that using such operators amounts to freezing the program forever, and no alternate fairness notions can be applied to it. This is the reason why we think that the idea of incorporating fairness into the implementation instead of the language is more attractive, since it is easy to change the scheduler and test several alternatives without changing the program itself. Every fair execution must be correct, although not every execution needs to be fair. This does not seem to be a shortcoming, but a desirable property.

3.6 Summary

In this chapter, we have presented fairness as a desirable property to abstract away from the low-level details that may have an effect on the behaviour of a system without ignoring them altogether. We have shown that fairness is not a single concept but a collection of abstractions that depend on the elements to which they are applied and the level of strictness under consideration. We have presented several taxonomies and a couple of frameworks to deal with these notions. Finally, we have argued on some controversial aspects.

We have published several results on this topic, namely: we presented our preliminary results on implementing IP_{CORE} by means of a centralised, general, strongly fair scheduler at the SEID'99 Conference [126] and the IEEE Euromicro'99 Conference [44]. Our preliminary attempts to devise a practical interaction fairness notion were presented at the Euro-Par'02 Conference [122] and The Colombian Journal of Computation [124]. Our final results on fairness were published in Concurrency and Computation: Practice & Experience [125].

Chapter 4

Implementing fair multipartiness

It is important to build a kingdom
through ruling in fairness.

King Solomon, 970–928 BC
Ancient israelian sovereign

The goal of this chapter is to present an overall idea of the most up-to-date algorithms to schedule multiparty interactions fairly. (Most of them are compared from an empirical point of view in Chapter §9.) We begin with a proposal to implement the economy-based level that has been adapted from the carpool scheduling setting; in Section §4.3, we present a description of three proposals to implement the strong level and a discussion on some well-known impossibility results; we then report on the only available implementation of ∞-fairness in Section §4.4; finally, we report on the implementation of k-bounded and finitary fairness, which is still an open research topic.

4.1 Introduction

In the literature, there are a variety of centralised and distributed techniques for implementing the multiparty interactions IP_{CORE} provides. The problem is usually decomposed into three main tasks, namely: implementing synchronisation, achieving mutual exclusion, and communicating several parties.

The first proposals were produced in the context of CSP [67], but they were restricted to two-party interactions. Later, the problem became of great interest, and Chandy and Misra [30] developed two algorithms that are the basis of Bagrodia's MEM algorithm [12], which is currently one of the most popular because it can be configured so as to achieve optimal performance in a particular system. Since the first solutions were developed, many techniques have been proposed, namely: to implement multiparty synchronisation, it is usual to use polling [35], message-counts [12], or auxiliary resources like tokens [30]; the exclusion problem may be solved with priorities [80], time stamps [74], auxiliary resources [12, 80], probabilistic techniques [76, 72], or queues [91]; very little work, however, has been reported on how to implement multiparty communication [141].

Most of the aforementioned techniques share a common drawback since they can guarantee weak fairness only; very often, this is a result of the underlying message passing or shared memory primitives used to implement them. This is the reason why current research in this field focus on implementing stronger notions. To the best of our knowledge, there are some good implementations of *economy-based fairness*, *strong fairness*, and ∞*-fairness*; furthermore, there are some results on the implementation of *weak finitary* and *weak-*k*-bounded fairness*, but they are not practical enough, yet.

4.2 Implementing economy-based fairness

To the best of our knowledge, no implementation of *economy-based interaction fairness* has appeared in the literature. However, the results by Fagin and Williams in the context of carpool scheduling algorithms may be easily adapted if we consider that there is a bound on the time a process may be performing local computation [53].

Their algorithm first defines U as the least common multiple of $1, 2, \ldots, N$, where N is the maximum number of interactions that can be enabled at a time. It needs an array of counters in which each column represents an interaction and each row the selection of one of them for execution. Initially, the array has

a row initialised to zeroes only. For an interaction to be selected, the system needs to be stable, *i.e.*, no process can be able perform local computation since they all must be waiting for an interaction to be executed. Every time this happens, the interaction selected is the one whose counter in the last row of the array is minimum. A new row is then added and the counters are updated as follows: the counter associated with the interaction that was selected is increased by $\frac{U(k-1)}{k}$, where k denotes the number of interactions that were enabled; the other counters are decreased by $\frac{U}{k}$. The authors proved that the algorithm is correct if the counters range in the interval $[-(N-1)f(N)U, f(N)U]$, where $f(N)$ ranges in the interval $\left[\frac{(N-1)}{3}, a(N)\right]$ with

$$a(x) = \begin{cases} 0 & \text{if } x = 1 \\ (x-1)a(x-1)+1 & \text{otherwise.} \end{cases}$$

(The authors attributed to Don Coppersmith from the T.J. Watson Research Centre lowering the exponential upper bound of $f(N)$ to $\frac{(N-1)}{2}$, but this result remains unpublished as of the time of writing this dissertation.)

Although the algorithm is quite intricate, the idea behind it is to guarantee that the difference between the ideal number of times an interaction should have been executed and the actual number of times it is executed does not exceed a bound. The ideal number of times is

$$\frac{1}{2}b_2 + \frac{1}{3}b_3 + \ldots + \frac{1}{N}b_N = \sum_{k=2}^{N} \frac{1}{k}b_k,$$

where b_k is the number of times when exactly k interactions are enabled, and the authors proved that their algorithm guarantees that the absolute value of the difference is not greater than $\frac{(N-1)a(N)}{2}$. (If Coppersmith's upper bound is true, it is easy to demonstrate that this exponential upper bound may be reduced to $\frac{(N-1)^2}{2}$.)

Recall that if we assume that there is a bound on the time a process may perform local computation without requiring to interact with others, then an implementation of the economy-based level may be viewed as an implementation of the bounded-delay level in some cases, but the converse is not true in general.

4.3 Implementing strong fairness

Strong fairness, which is also called *0-fairness* or *compassion*, is probably the most common practical fairness notion, and this is the reason why it has attracted the attention of most researchers in this field. In this section, we first

```
SYSTEM :: [P1 || P2 || P3], where
    P1 :: *[ A⟨⟩ → skip [] C⟨⟩ → skip ] ;
    P2 :: *[ A⟨⟩ → skip [] B⟨⟩ → skip ] ;
    P3 :: *[ B⟨⟩ → skip [] C⟨⟩ → skip ] .
```

Program 4.1: *Tsay and Bagrodia's counterexample.*

report on some results that prove that it is not implementable with deterministic wait-free algorithms, and then present three well-known proposals by Francez and Forman, Corchuelo *et al.*, and Joung.

4.3.1 Impossibility results

Tsay and Bagrodia were the first to prove that strong interaction fairness cannot be implemented by means deterministic wait-free schedulers [138]. They proved the impossibility result for biparty interactions and generalised it to the multiparty case. The assumptions they took into account were the following:

i. No participant in an interaction can perform local computations indefinitely, *i.e.*, participants must offer to participate in an interaction from time to time. This implies that they cannot finish.

ii. The participants in an interaction are autonomous and independent from each other, *i.e.*, the subset of interactions in which a process may participate is not dependent on the other processes.

iii. The state changes of a participant are not noticeable by the other participants immediately, *i.e.*, the time needed to communicate a change is not negligible.

They demonstrated their result by means of the counterexample shown in Program §4.1, in which the following execution is possible: initially all of the processes are performing local computation, and after a while both P_1 and P_2 offer to participate in either interaction A or B, which enables A for execution. If we attempt to schedule interactions deterministically without introducing any delay, A should be executed as soon as it is detected to be enabled. However, some time must elapse between these events. Since processes are autonomous, it is possible for P_3 to offer to participate in either B or C in the interim. This implies that all of the interactions are enabled immediately after

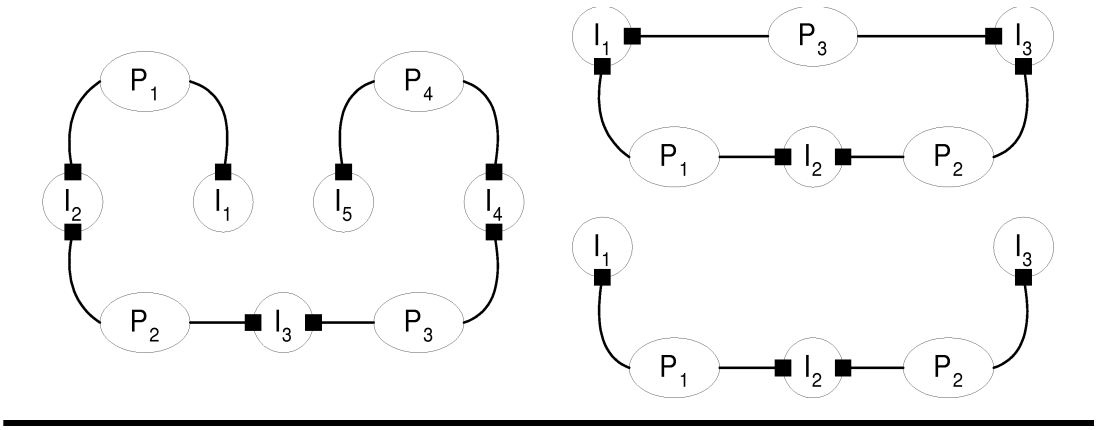


Figure 4.1: *Joung and Smolka's impossibility results.*

the decision to execute A has been made, but just before it has been executed. Obviously, if this scenario is repeated infinitely many times, neither interaction B nor C can be selected for execution, but they become enabled infinitely many times. That is, this execution is not strong interaction-fair.

Independently, Joung and Smolka [71] reached similar conclusions, but they characterised the exact topology of the subsystems in which strong interaction fairness cannot be implemented with deterministic wait-free algorithms. These subsystems, which are very common practice, are depicted in Figure §4.1.

In summary, the only solution to implement strong interaction fairness is to use deterministic algorithms that can delay the execution of some interactions for arbitrarily long periods of time, or to use probabilistic techniques. Obviously, this amounts to inefficiency, but no other solutions are possible.

4.3.2 A straightforward scheduler

Francez and Forman devised a scheduler that builds on Olderog and Apt's results in the context of parallel programs with shared variables [103]. This solution associates a counter initialised to a random natural number to every interaction. Thus, an enabled interaction is selected iff its counter is minimum over the set of conflicting interactions; in such cases, its counter is reset to a random natural value, and the counters associated with the remaining interactions is decreased by one, thus increasing their priority if conflicts arise.

It is worth mentioning that this algorithm requires to know the status of the whole set of interactions before a decision can be reached, *i.e.*, it delays the

execution of the interactions until the whole system is stable. Furthermore, it requires infinite counters to be implemented or, on the contrary, the executions it produces may be both incorrect and incomplete. This motivated Corchuelo to devise a slightly modified version in which counters can be finite and the scheduler remains correct, although it is not complete in such cases [35].

4.3.3 An incremental scheduler

The main motivation behind Corchuelo's scheduler is to produce an incremental version of Francez and Forman's proposal in which the status of the whole set of interactions needs to be known in the worst case only, thus improving efficiency [35]. It arranges interactions in a queue according to their counters so that the nearer to the front an interaction is, the smaller its counter is. It then examines the interactions of this queue in sequence and stops when it finds the first enabled interaction. Its counter is then reset to a random natural value, whereas the counters of the remaining interactions are decreased by one in order to increase their priority.

This scheduler has proved to be faster than Francez and Forman's original proposal in the average case. Furthermore, it can be implemented with finite counters and this does not amount to incorrectness, as was the case before.

4.3.4 A randomised scheduler

Recently, Joung [72] presented two randomised schedulers that are based on the *Theory of Big Numbers* and allow to implement strong interaction fairness with probability one. They both are equivalent, and the difference lies in that one of them uses message-passing primitives, whereas the other uses shared-memory primitives. They improve on a previous result in that the time an entity spends at doing local computation or the transmission delays need not be bounded by any predetermined constant, although they cannot deal with systems in which an entity increases the time it spends at doing local computation monotonically [76].

Both algorithms are based on the idea of "attempt, wait, and check": when a process is ready to participate in a subset of interactions, it then selects one of them randomly, and waits for some time which is referred to as the monitoring window; if after that it can detect that the other participants are also interested in that interaction, it then can be executed; otherwise, a new random attempt is made and the procedure is repeated again.

The cost of a random attempt-wait-check cycle may be considerably high. The time that a participant spends at trying to establish an interaction was proven not to be greater than

$$\frac{|\mathcal{P}|\eta}{\prod_{p\in\mathcal{P}}\psi_p} + (|\mathcal{P}| - 1)\eta + \epsilon,$$

where $\mathcal{P}$ is the set of processes that may eventually participate in that interaction, ψ_p is the probability that participant p chooses that interaction in a random draw, and $\eta - \epsilon$, $\eta > \epsilon > 0$ is the length of the monitoring window. The time complexity increases sharply as $|\mathcal{P}|$ increases and makes it impractical for applications with four- or five-party interactions or applications in which participants offer more than three interactions at a time.

4.4 Implementing ∞-fairness and hyperfairness

To the best of our knowledge, there is only one implementation of ∞-*fairness*. However, it has many limitations and implements only a subset of the notion that is referred to as *hyperfairness*. This proposal was presented by Attie *et al.* in Reference [10], and it is a transformational technique in which the source code of the program under consideration needs to be changed so as to embed an *ad hoc* hyperfair scheduler.

The transformation consists of adding a new process to the system. It is referred to as the scheduler since it controls the enablement and execution of interactions independently from the scheduler provided by the run-time system or the underlying hardware. It has a counter for each interaction, and a variable that denotes the interaction whose counter is minimum. The counters are initialised to random natural values. Every process can read these variables, but the only that can change them is the scheduler. Furthermore, the processes need to be in interaction-driven normal form (*cf.* Appendix §B), and the guarding predicates in their main loops need to be changed as follows:

$$B_x^P \wedge (\forall \text{ interaction } y \cdot \text{participates}(P, y) \wedge y = \text{MIN} \Rightarrow (x = y \vee \neg B_y^P)),$$

B_x^P being the original guarding predicate, x its corresponding interaction, P the process to which the guard belongs, and MIN the interaction with the minimum counter. When an interaction is executed its counter is reset to a random natural value, the counter of the remaining interactions is decreased by one, and MIN is updated accordingly.

We illustrate the transformation by means of Program §4.3, which is the hyperfair equivalent to Program §4.2. Note that this transformation can and should be simplified, but this is not essential for the algorithm to be correct.

```
S :: [P || Q], where
P :: *[ true & x⟨⟩ → skip
     [] true & y⟨⟩ → skip
     ];
Q :: s := 0;
     *[ s ∈ {0, 1} & y⟨⟩ → s := 1
     [] s = 1 & x⟨⟩ → skip
     ].
```

Program 4.2: *Hyperfair transformation: original program.*

In summary, the idea behind this implementation is to delay the enablement of interactions other than the most priority on the hope that it becomes enabled eventually and thus can be selected for execution. Note that this can produce a deadlock if that interaction cannot become enabled if the remaining are delayed. This may happen, for instance, in Program §4.2 if the initial random assignment to the counters makes x the most priority interaction. In this case the scheduler would delay the execution of interaction y until the other becomes enabled; however, this may not happen since x cannot be executed unless y is executed before.

Attie *et al.* characterised the subset of programs to which the algorithm can be applied. They concluded that it can be used as long as the processes are in interaction-driven normal form and every interaction is *conspiracy resistant*. Roughly speaking, an interaction is conspiracy resistant iff delaying the execution of the rest does not prevent it from becoming enabled in finite time. However, they did not devise an algorithm to calculate the set of conspiracy-resistant interactions in a program, which usually turns into a very difficult problem.

4.5 Implementing k-bounded and finitary fairness

Alur and Henzinger presented a transformational method to implement weak-k-bounded and finitary weak fairness in the context of standard transition systems [5, 95]. The work on implementing multiparty interactions using this concurrency model is still in progress [106, 107], but it is easy to realise that the execution of an interaction in a multiparty-based system can be interpreted as the execution of an appropriate transition in its equivalent standard transition system, whereas an interaction being enabled amounts to its corresponding transition being enabled.

```
S :: [P || Q || H], where
P :: *[ true ∧
        (x = MIN ⇒ (x = x ∨ ¬true)) ∧
        (y = MIN ⇒ (x = y ∨ ¬true)) & x⟨⟩ → skip
     [] true ∧
        (x = MIN ⇒ (y = x ∨ ¬true)) ∧
        (y = MIN ⇒ (y = y ∨ ¬true)) & y⟨⟩ → skip
     ];
Q :: s := 1;
     *[ s ∈ {0, 1} ∧
        (x = MIN ⇒ (y = x ∨ s ≠ 1)) ∧
        (y = MIN ⇒ (y = y ∨ s ∉ {0, 1})) & y⟨⟩ → s := 1
     [] s = 1 ∧
        (x = MIN ⇒ (x = x ∨ s ≠ 1)) ∧
        (y = MIN ⇒ (x = y ∨ s ∉ {0, 1})) & x⟨⟩ → skip
     ];
H :: { This is the scheduler }
     z(x) := Ξ; z(y) := Ξ; MIN := (z(x) ≤ z(y) ? x : y);
     *[ Enabled(x) → z(x) := Ξ; z(y) := z(y) - 1; MIN := (z(x) ≤ z(y) ? x : y)
     [] Enabled(y) → z(x) := z(x) - 1; z(y) := Ξ; MIN := (z(x) ≤ z(y) ? x : y)
     ].
```

Program 4.3: *Hyperfair transformation: hyperfair program.*

A standard transition system may be viewed as a three-tuple of the form (Q, Q_0, T), where $Q = \{q_0, q_1, \ldots\}$ is a set of states, $Q_0 \subseteq Q$ is a set of initial states, and $T = \{\tau_0, \tau_1, \ldots, \tau_m\}$ is a finite set of transitions. Each state $q \in Q$ is an assignment of values to every variable, and each transition is interpreted as a binary relation on the states. For a transition $\tau \in T$ and a state $q \in Q$, $\tau(q)$ denotes the set $\{q' \mid (q, q') \in \tau\}$. Alur and Henzinger's transformation consists of adding a new counter associated with each transition and a threshold variable that helps prevent a transition that is enabled k times in a row from not executing. That is, the state space of the transformation is $Q \times \mathbb{N}^m \times \mathbb{N}^+$, the set of initial states is $Q_0 \times \{0\}^m \times \mathbb{N}^+$, and the set of transitions contains $((q, c_1, c_2, \ldots, c_m, b), (q', c'_1, c'_2, \ldots, c'_m, b))$ iff $q' \in \tau(q)$, $b' = b$, and $\forall i \in [1, m], \tau_i(q) = \emptyset \vee q' \in \tau_i(q)$ implies that $c'_i = 0$, or else $c_i < b \wedge c'_i = c_i + 1$.

Roughly speaking, the goal of the transformation is to count the number of times a transition has been neglected in a row so that is never greater than the threshold, which is left unspecified. For instance, Figure §4.2.(a) shows a simple state chart that models a system with a variable that ranges in the set $\{0, 1, 2, 3\}$ and is assigned a random value initially; there are also two transitions that may increase or decrease it as long as it is not null. Figure §4.2.(b) represents its corresponding standard transition system, and Figure §4.3 its finitarily weak transformation for any $b \geq 4$.

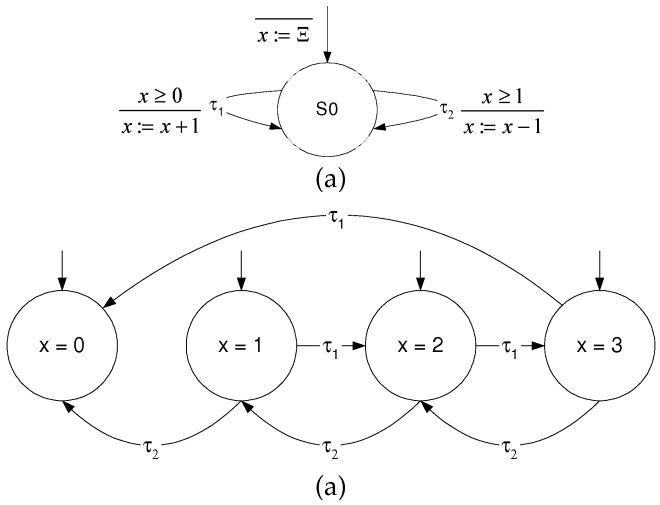


Figure 4.2: *Finitary transformation: original state chart and transition system.*

Note that, in general, the number of states in a standard transition system is enormous for simple systems. In our example, if x were a typical 32-bit natural variable, the standard transition system would consist of 2^{32} different states and the finitary transformation would exceed this number by many orders of magnitude. In practice, setting k to a constant value beforehand is necessary, and the transformation is then weakly-k-bounded, but the number of states may still exceed the number of original states by several orders of magnitude. To the best of our knowledge, no results on how to transform a state chart have been published, which would be more suitable for practical purposes.

4.6 Summary

In this chapter, we have presented an overview of current algorithms to implement fair interaction schedulers at different levels. Most researchers have focused on strong interaction fairness, which was proven not to be implementable with deterministic wait-free algorithms. Very little results regarding the remaining levels have been reported in the literature. However, some algorithms from other fields can be easily applied to our research context, *e.g.*, the carpool scheduling algorithm by Fagin and Williams, whereas others still need research before they can be adapted, *e.g.*, the transformation by Alur and Henzinger.

Our main results in this field are the following: we presented our prelimi-

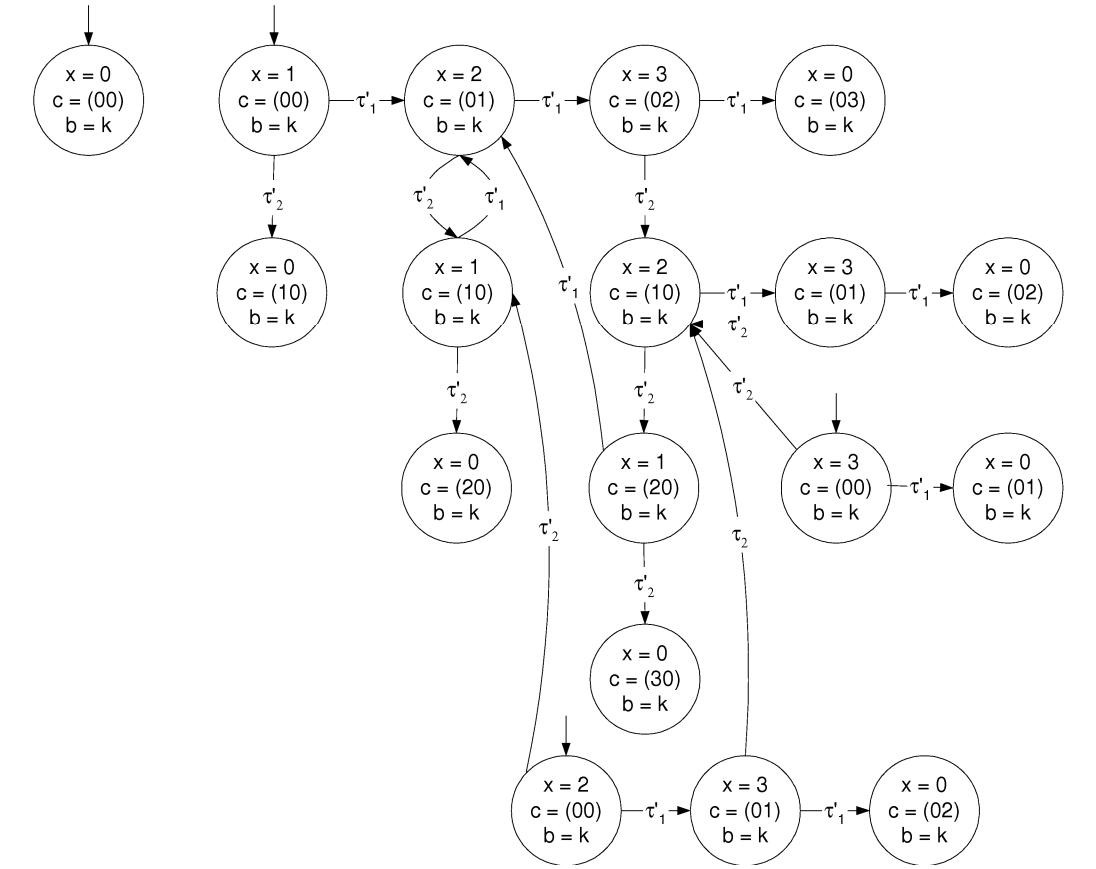


Figure 4.3: *Finitary transformation: resulting transition system.*

nary ideas on implementing IP_{CORE} by means of a centralised, general, strongly fair scheduler at the SEID'99 Conference [126] and the IEEE Euromicro'99 Conference [44]. Our preliminary attempts to devise a practical interaction fairness notion and to implement it were presented at the Euro-Par'02 Conference [122] and The Colombian Journal of Computation [124]. Our final results on the implementation of interaction fairness were published in Concurrency and Computation: Practice & Experience [125].

Part III

Our contribution

Chapter 5

Motivation

Motivation will almost always beat mere talent.

Norman R. Augustine, 1935–
Chairman of Lockheed Martin Corporation

Recently, fairness has been one of the most interesting research areas in the field of multiparty interactions. However, current notions are not practical enough since they are not able to address fair finiteness and conspiracies. Our goal in this chapter is to present these problems and motivate the need for stronger, but yet implementable notions. In Sections §5.1, §5.2 and §5.3, we present these problems in detail and identify their cause; in Section §5.4, we prove that none of the current fairness notions solve these problems at a time; in Section §5.6, we discuss our results and argue on why other authors have not attempted to provide a satisfactory solution; finally, we summarise this chapter and our main publications in Section §5.5

5.1 Introduction

Fairness was designed to help verify some complex properties of non-deterministic programs from a theoretical point of view. The goal was to devise an abstraction by means of which it could be possible to abstract away from many low-level details concerning the scheduling of conflicting choices without ignoring them altogether. In other words, the goal was to identify a common property able to generalise all desirable behaviours of a program as a start point for reasoning and verifying other properties [59].

Eventuality was such a start point, but, unfortunately, it can be viewed as a major practical problem since no real-world finite-state scheduler can implement eventuality in its full extent. Later, the idea of *bounded eventuality* was presented as a means to put a bound on the eventuality a scheduler has to implement. Recently, it has attracted the attention of several researchers since it helps prove some properties more easily than pure eventuality [5, 46].

Some authors have also proposed fairness notions based on eventual occurrence of some events as a means to keep the execution of a system under control. Unfortunately, no current notion seems to be appropriate enough in the context of multiparty interactions. The reason is that either they are not (reasonably) implementable or they cannot deal with a couple of problems known as fair finiteness and conspiracies. This argues for a new fairness notion, and it is the main motivation for this thesis.

5.2 Fair finiteness

Since both oracle- and eventuality-based fairness notions rely on the eventual execution of every interaction that may become or is enabled infinitely often, this implies that no real-world finite-state scheduler can produce the whole set of executions that satisfy such notions [5, 59]. The reason is that there is always a physical bound on the eventuality a computer may implement that depends on the number of states, the range of the counters, the available memory or whichever resource a scheduler may need, which must be finite by nature. For instance, the counter-based scheduler by Corchuelo on which we reported in Section §4.3.3 cannot generate any fair execution in which an interaction is executed after $N + 1$ other interactions have been executed if the counters can take N different values only.

However, as we proved in Section §3.5.2, using incomplete fair schedulers seems desirable in practice since they may produce executions in which the

$[P_2.\{Get_2\}\ P_4.\{Get_4\}\ P_5.\{Get_5\}\ F_4.\{Get_4, Get_5\}] \frown$
$[P_1.\{Get_1\}\ P_3.\{Get_3\}\ F_1.\{Get_1, Get_2\}\ F_2.\{Get_2, Get_3\}\ F_3.\{Get_3, Get_4\}\ F_5.\{Get_5, Get_1\}$
$\mathbf{Get_1}\ \mathbf{Get_3}\ P_1.\{Rel_1\}\ F_1.\{Rel_1\}\ F_5.\{Rel_1\}\ P_3.\{Rel_3\}\ F_2.\{Rel_3\}\ F_3.\{Rel_3\}\ \mathbf{Rel_1}\ \mathbf{Rel_3}]^*$

Figure 5.1: *A sample execution to illustrate fair finiteness.*

scheduling of interactions does not diverge too much from the ideal theoretical distribution. Since we also showed that this behaviour depends on the random number generators used, this implies that the goodness of the result does not depend on the notion itself, but on the implementation of the scheduler. This is not desirable in general since although all of the implementations guarantee that interactions are scheduled in a strongly fair manner (as long as the counters are large enough), the difference between them matters from a practical point of view.

Furthermore, no program can run forever since they must necessarily stop or be stopped a day. This implies that every real-world execution is finite by nature, which in turn implies that every real-world execution fulfils trivially the requirements of any oracle- or eventuality-based fairness notion. The reason is that no real-world interaction can be enabled infinitely many times in a finite execution, neither possibly nor continuously nor sporadically; thus, no oracle- or eventuality-based notion can force any interaction to be executed from time to time. Thus, one might argue that it does not matter whether the scheduler is fair or not since the observed behaviour of any program shall always be k interaction-fair for all $k \in \mathbb{N}$, ∞ interaction-fair, strongly interaction-fair, weakly interaction-fair and unconditionally interaction-fair [25, 50, 102, 131].

For instance, Figure §5.1 illustrates a partial execution of the well-known dining philosophers system presented in Program §2.4.1. This system does not terminate, but this execution is possible if we press a break key on the keyboard, or send a kill signal to the corresponding operating system process, or the system is rebooted for maintenance, or whichever other reasonable cause. We have left out the details concerning the states of the processes, and have focused on the moments at which a process offers to participate in a subset of interactions and the moments at which an interaction is executed. The former events are denoted as $P.\{I_1, I_2, \ldots, I_n\}$, whereas the latter are denoted as the name of the interaction itself. Note than only interactions Get_1, Get_3 and their corresponding Rel_1 and Rel_3 are executed. However, Get_2, Get_4, and Get_5 are always enabled when both Get_1 and Get_3 become enabled, and their execution would allow Rel_2, Rel_4, and Rel_5 to have a chance to be executed.

$[P_1.\{Get_1\}\ P_2.\ \{Get_2\}\ P_3.\{Get_3\}\ P_4.\{Get_4\}\ P_5.\{Get_5\}] \frown$
$[F_1.\{Get_1, Get_2\}\ F_3.\{Get_3, Get_4\}\ F_5.\{Get_5, Get_1\}\ \mathbf{Get}_1\ P_1.\{Rel_1\}\ F_1.\{Rel_1\}\ F_5.\{Rel_1\}\ \mathbf{Rel}_1$
$F_2.\{Get_2, Get_3\}\ \mathbf{Get}_3\ P_3.\{Rel_3\}\ F_2.\{Rel_3\}\ F_3.\{Rel_3\}\ \mathbf{Rel}_3\ P_1.\{Get_1\}\ P_3.\{Get_3\}$
$F_3.\{Get_3, Get_4\}\ F_4.\{Get_4, Get_4\}\ \mathbf{Get}_4\ P_4.\{Rel_4\}\ F_3.\{Rel_4\}\ F_4.\{Rel_4\}\ \mathbf{Rel}_4\ P_4.\{Get_4\}]^{\infty}$

Figure 5.2: *A sample execution to illustrate conspiracies.*

In spite of this observation, this execution fulfils the requirements of any oracle- or eventuality-based fairness notion. Even if Get_1 and Get_3 are executed one *googolplex*[†1] times to the detriment of the rest, the execution still fulfils the requirements of any oracle- or eventuality-based fairness notion. We refer to this problem as *fair finiteness*, and it clearly demonstrates that eventuality is too weak for practical purposes and argues for a stronger notion. Bounded fairness might be the answer, but there is still a problem that it is unable to address: conspiracies.

5.3 Conspiracies

Conspiracies are very intricate and one of the major obstacles for a fairness notion to be fully adequate according to the criteria by Apt *et al.* [8, 10, 18, 19]. Roughly speaking, a conspiracy occurs if an interaction cannot become enabled because its participants are ready to execute it infinitely many times, but never at the same time, *i.e.*, if the interleaving was different, there would be a chance for they all being ready to execute that interaction at the same time.

For instance, Figure §5.2 illustrates an infinite execution of the dining philosophers system that fulfils the requirements of all eventuality- and bounded-based notions. There are only some oracle-based notions, in particular, ∞-fairness, which are not satisfied. Note, however, that neither interaction Get_2 nor interaction Get_5 become enabled in this execution. In the case of Get_2, the reason is that fork F_1 commits to interaction Get_1 before fork F_2 can offer to participate in Get_2. A simple reordering of independent actions in which fork F_2 offers to participate in either Get_2 or Get_3 just before Get_1 is executed would

[†1]The American mathematician Edward Kasner once asked his nine-year-old nephew to invent a name for a very large number: ten to the power of one hundred. The boy called it a googol. He thought that this would overflow people's minds, being bigger than anything that can ever be put into words. Another mathematician then shot back with googolplex, and defined it to be ten to the power of googol, proving Kasner wrong in the first instance. The name become popular with Carl Sagan's Cosmos TV show.

```
SYSTEM :: [P || Q], where
        P :: *[ A⟨⟩ → B⟨⟩ [] C⟨⟩ → skip ] ;
        Q :: *[ A⟨⟩ → [ B⟨⟩ → skip ‖C⟨⟩ → skip ] ] .
```

Program 5.1: *A program with inherent conspiracies.*

allow Get_2 to become enabled and, therefore, it would have infinitely many opportunities to be executed. The reasoning is similar in the case of Get_5.

It is worth mentioning that not all conspiracies may be avoided. For instance, in Program §5.1, it is impossible for interaction C to become enabled since this program is altogether deterministic because interaction A must be executed before interaction B, and there is absolutely no chance for P and Q to be ready to participate in interaction C at the same time. We say that this conspiracy is *inherent*, whereas the previous case was *non-inherent*. The distinction between them is important since we can attempt to spoil non-inherent conspiracies by delaying the execution of some interactions, *i.e.*, by rearranging the independent atomic actions executed in a system. However, there does not exist a rearrangement able to spoil the inherent conspiracy in Program §5.1.

On account of this observation, it is clear that it would be desirable for a practical fairness notion in the context of multiparty interactions to be able to spoil conspiracies, thus giving every interaction that could become enabled an opportunity to be executed.

5.4 Analysis of current solutions

Our goal in this section is to prove that none of the implementable fairness notions presented in Chapter §4 addresses the aforementioned problems at a time, but independently.

5.4.1 Economy-based fairness

The economy-based level in Francez's taxonomy requires the difference between the number of times an interaction is executed and the ideal number to be bounded by a predefined constant that depends on the number of interactions only.

This disallows an execution such as the one presented in Figure §5.1. In this case the difference would increase monotonically as the length of the execution increased in the case of interactions Get_2, Get_4, Get_5, Rel_2, Rel_4, and Rel_5 since they are never executed. The problem with this notion is that it allows conspiratorial executions such as the one presented in Figure §5.2 since the interactions that do not get enabled are not taken into account.

Recall that if we assume that there is a bound on the time a process may perform local computation without requiring to interact with others, then an implementation of the economy-based level may be viewed as an implementation of the bounded-delay level in some scenarios. In this case, this level would disallow some finite executions, but it still could not deal with conspiracies.

5.4.2 Strong, weak and unconditional fairness

Technically, the strong level prevents interactions that are enabled infinitely often from not executing. Thus, it is subsumed in both the weak and unconditional levels. Unfortunately, it cannot deal with neither fair finiteness nor conspiracies.

For instance, the partial execution in Figure §5.1 is strong, weak and unconditionally interaction-fair since no interaction is enabled infinitely many times, which implies they need not be executed at all. Similarly, the execution in Figure §5.2 also fulfils the requirements of these fairness levels because the conspiracy against Get_2 and Get_5 prevents them from becoming enabled.

5.4.3 ∞-fairness and hyperfairness

To the best of our knowledge, the only notion that can deal with conspiracies is ∞-fairness since it does not depend on the execution under consideration, but on the possible continuations of its finite prefixes. Thus, if an interaction can become enabled infinitely often along an execution, the notion would force it to become enabled infinitely often, which, in turn, would imply that it must be executed infinitely often.

The only materialisation of which we are aware in the context of multiparty interactions is hyperfairness [10]. Unfortunately, this notion has several practical drawbacks since every finite execution is hyperfair by definition; furthermore, if there is one non-conspiracy-resistant interaction in a program, then every execution that program may produce is hyperfair by definition; moreover, no general algorithm to determine whether an interaction

is conspiracy-resistant has been presented in the literature; additionally, the notion can be applied to interaction-driven programs only, but this is not an important drawback since it is not to difficult to transform a program into its equivalent in interaction-driven normal form by introducing some state variables.

These drawbacks imply that both the executions in Figure §5.1 and §5.2 are hyperfair since the former is finite, and the latter was produced by a program with non-conspiracy-resistant interactions. Recall that an interaction is said to be conspiracy-resistant if delaying the execution of the rest of the interactions cannot prevent it from becoming enabled in finite time. Note that if the execution of, say, Get_1 is delayed, then Rel_1 cannot become enabled in finite time since Rel_1 requires Get_1 to be executed before it can proceed, *i.e.*, a fork cannot be released unless it was taken before.

5.4.4 k-bounded and finitary fairness

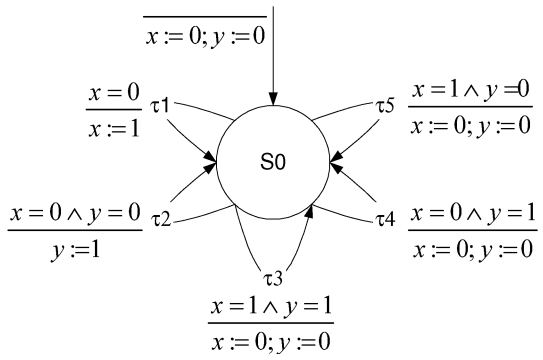
Alur and Henzinger's k-bounded and finitary levels were designed in the context of standard transition systems [95]. The work on implementing multiparty interactions using this concurrency model is still in progress [106, 107], but we can prove that the results shall not be satisfactory since neither k-bounded nor finitary notions can deal with conspiracies.

As for the fair finiteness problem, both weak-k-bounded and strong-k-bounded fairness can disallow the finite execution in Figure §5.1, for instance. It does not fulfil the requirements of these notions since interaction Get_2 is enabled more than k times for any non-null natural k smaller than the length of the execution, but it is never executed. Nonetheless, the generalisation of k-bounded fairness to finitary fairness does not solve the fair finiteness problem since every finite execution is finitarily fair by definition. The reason is that there is always an upper bound to the number of times an interaction can be executed in a row if the execution is finite.

Anyhow, none of these levels can spoil conspiracies. To prove this point, we use the program in Figure §5.3 since the standard transition system associated with the dining philosophers system would be unnecessarily complex and difficult to understand. Note that an execution in which the interactions are selected according to the pattern $[\tau 2 \tau 4 \tau 1 \tau 5]^{\infty}$ is possible and prevents interaction $\tau 3$ from becoming enabled and executing. The reason is that transitions $\tau 1$ and $\tau 2$ enable the guarding predicate of transition $\tau 3$ partially and intermittently, but the execution of transition $\tau 4$ in-between prevents transition $\tau 3$ from becoming enabled infinitely many times [18, 19]. Figure §5.3

S :: [P || Q], **where**
P :: { *x: natural* }
*[x = 0 & τ1⟨⟩ → x := 1
[] x = 0 & τ2⟨⟩ → skip
[] x = 1 & τ3⟨⟩ → x := 0
[] x = 0 & τ4⟨⟩ → x := 0
[] x = 1 & τ5⟨⟩ → x := 0
]
Q :: { *y: natural* }
*[y = 0 & τ2⟨⟩ → y := 1
[] y = 1 & τ3⟨⟩ → y := 0
[] y = 1 & τ4⟨⟩ → y := 0
[] y = 0 & τ5⟨⟩ → y := 0
]

(a)



(b)

Figure 5.3: *A program with conspiracies and its corresponding state chart.*

shows the standard transition system associated with this program and its corresponding finitarily weak transformation. Note that the aforementioned conspiracy may occur since this finitary transition system does not force transition τ1 to occur in state $x = 0, y = 1$.

It is also worth mentioning that setting k to some values may spoil this conspiracy, but other against τ5 arises then. For instance, if $k = 2$, then transition τ4 cannot occur in state $x = 0, y = 1$ since it would neglect transition τ1 two consecutive times. In this case, if τ2 is executed initially, then the only subsequent choices are to execute τ1 and τ3 in sequence. Thus an execution in which the interactions are selected according to the pattern $[\tau 2\tau 1\tau 3]^{\infty}$ is possible, and it would prevent τ5 from becoming enabled and τ4 from executing.

Although we are not aware of any finitarily strong transformation, the result would not be able to spoil conspiracies either. The reason is that transition τ3 depends on transition τ1 to be executed when the system is in state $x = 0, y = 1$, but it need not be executed in this state. Note that this does not contradict the strong requirement since transition τ1 is enabled infinitely many times and it is executed infinitely many times when the system is in state $x = 0, y = 0$.

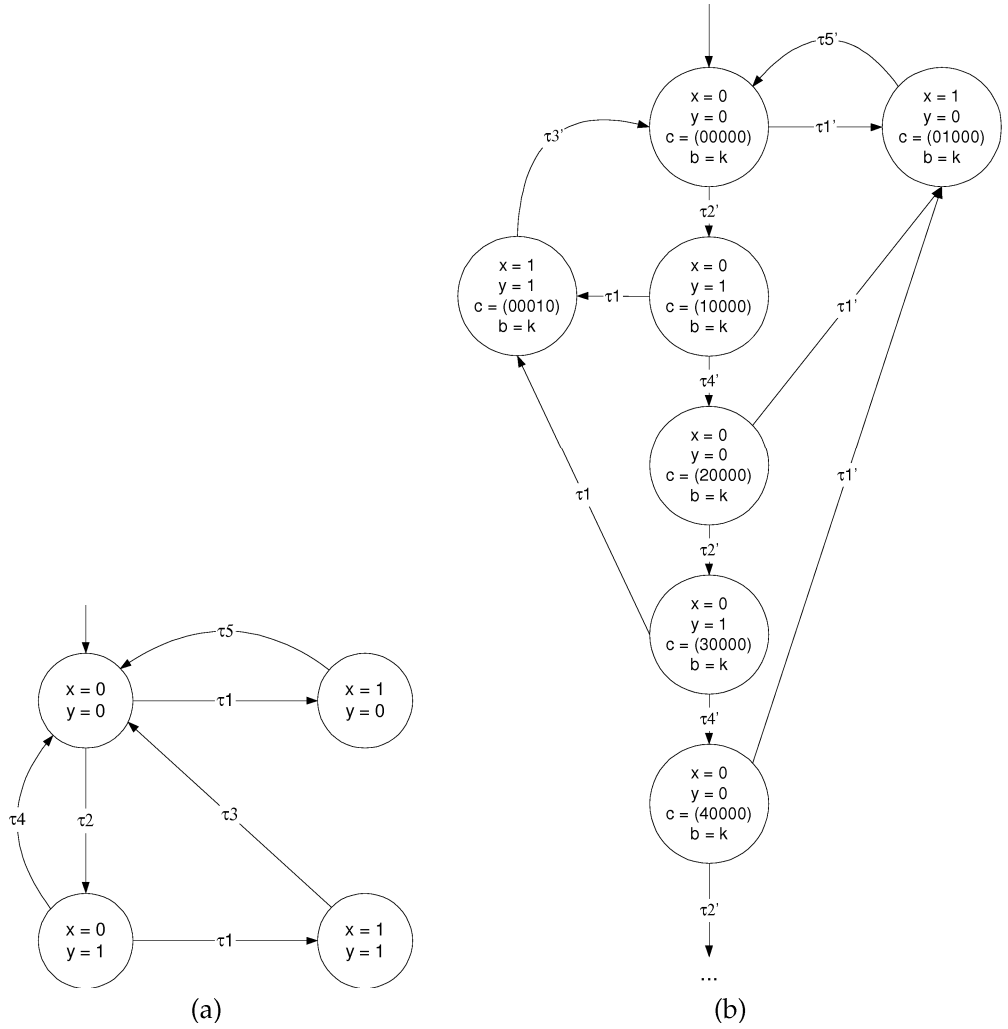


Figure 5.4: *Standard transition system for the program in Figure §5.3, and its weakly finitary-fair transformation.*

5.5 Discussion

From the previous sections, it follows that a notion able to address both fair finiteness and conspiracies is very desirable from a practical point of view; unfortunately, current fairness proposals focus on one of these problems at most (with some limitations in the case of conspiracies). Table §5.1 summarises the proposals we have taken into account and their ability to solve fair finiteness and conspiracies. The table is not complete, but the notions that have been left out are special cases. For instance, weak fairness is a special case of strong fairness. The last row shows the properties of k-conspiracy-free fairness, which is the notion we introduce in this doctoral dissertation. As we prove, it is the only that solves both fair finiteness and conspiracies at a time. Strong k-fairness was a previous attempt, but it could alleviate some non-inherent conspiracies only.

We think that current proposals have not addressed both problems simultaneously since their authors focused on different settings. For instance, the work by Alur *et al.* on finitary fairness focuses on concurrent systems that are not based on multiparty interactions, but on standard transition systems and the concept of conspiracy is not common in this field. To the best of our knowledge, it has appeared in the literature regarding fair selection in multi-choice commands, Petri nets and multiparty interactions only [10, 18, 19].

Attie *et al.* focused on multiparty interactions, and they laid the foundations of hyperfairness; unfortunately, their ideas were not developed to their full extent since their proposal has many practical limitations. However, they deserve attention since they were the first to identify the problem and devise a solution. Very recently, in References [83, 84], Lamport considers hyperfairness and ∞-fairness a corner-stone of action-based concurrent systems and recognises the need for further research on this topic. Hyperfairness does not address fair finiteness since the authors focused on designing a notion to preserve a property called equivalence robustness, which is mandatory for a notion to be fully adequate according to the criteria in References [8, 57]. Conspiracies were proved to constitute a major obstacle to preserving this property, and avoiding them was Attie *et al.*'s main motivation. Note that according to the definition, no finite execution may be conspiratorial.

5.6 Summary

Our goal in this chapter has been to motivate the reason why we embarked on the development of this thesis. We have analysed two major practical prob-

Notion	Description	Fair fin.	Conspir.
Economy-based fairness	No interaction is executed less than it should according to an economic criterion.	Yes	No
Strong fairness	No interaction that is enabled sufficiently often is neglected forever.	No	No
∞-fairness	Every interaction that can possibly become enabled becomes enabled infinitely many times and is executed infinitely many times.	No	Yes
Hyperfairness	Conspiracy-resistant interactions become enabled infinitely many times and are executed infinitely many times.	No	Yes
Strong-k-bounded fairness	No subsequence of an execution in which an interaction has become enabled k times contains no occurrence of that interaction. k must be known beforehand.	Yes	No
Finitary strong fairness	No subsequence of an execution in which an interaction has become enabled k times contains no occurrence of that interaction. k is not set *a priori*, but it must exist.	No	No
Strong k-fairness	No interaction is selected more than k times without analysing the state of the interactions with which it is in conflict.	Yes	Alleviates
k-conspiracy-free fairness	No interaction is selected more than k times without analysing the state of the interactions that may be in conflict with it.	Yes	Yes

Table 5.1: *Comparison with related notions.*

lems of eventuality, and have proved that none of the notions that have been applied to multiparty interactions succeed in solving them at a time. This proves that our contribution is original and advances the state of the art a step forward.

Our main results on this topic were published at the Euro-Par'02 Conference [122] and The Colombian Journal of Computation [124], where we presented our preliminary ideas and a notion that solved fair finiteness but could not solve conspiracies completely. Our main result was published in Concurrency and Computation: Practice & Experience [125].

Chapter 6

Our framework to support multiparty interactions

We dissect nature along lines laid down by our native language, which is not simply a reporting device for experience, but a defining framework for it.

Benjamin L. Whorf, 1897–1941
Business specialist

Due to the inherent complexity of multiparty-based models, it is important to define a conceptual framework in which every term can be defined precisely. Our goal in this chapter is to present the framework we have devised to deal with the interaction model IP_{CORE} *provides. In Section §6.1, we present an informal introduction by means of a simple mental map; in Section §6.2, we present its formal definition; we report on several applications to define common safety, liveness or fairness notions in Section §6.3; finally, we summarise our contributions and publications in Section §6.4.*

6.1 Introduction

Although the idea of multiparty interactions is conceptually simple, there are many details that must be taken into account. This argues for a conceptual framework in which they can be described precisely so that no misunderstanding can occur. The framework we have devised is simple, but not trivial, and this is the reason why it is important to have a good intuitive understanding before presenting it. The mental map in Figure §6.1 may help understand it more easily. It revolves around four concepts that are explained below, namely: participants, interactions, systems, and executions.

A *participant* is a software artefact that can perform local computation on its own state independently from the rest of artifacts, but needs to cooperate with them from time to time. A participant can be in three different states, namely: *working*, *waiting for an interaction* to be executed or *finished*. (Sometimes, we say that a participant that is in the waiting state is *blocked*.) They can produce *offering events* of the form $p.\chi$ to let other participants know that they cannot proceed unless they cooperate with others by means of an interaction in set χ, or consume *synchronisation events* of the form x that allow them to cooperate with others by means of interaction x. Note that the offering events are controlled by the semantics of the language under consideration, whereas synchronisation events are controlled by the semantics of the interaction model. It is worth noting that an event of the form $p.\emptyset$ disallows participant p from executing any action, which we identify with its termination since it reaches a fixed point in which it can perform neither local computation nor interact with others.

A *multiparty interaction* is an abstraction that allows a fixed set of known *participants* to execute a set of data exchange actions atomically and coordinately. An interaction is said to be *stable* if none of its participants is working; otherwise it is said to be *unstable*. A stable interaction is *enabled* iff all of its participants have offered to participate in it; it is *disabled* iff it is unstable or it is stable but one of its participants is waiting for other interactions to be executed. The set of interactions in which a process may participate is said to be *statically linked*, whereas the set of interactions in which a process offers to participate at run time is said to be *dynamically linked*. Note that every interaction is linked to itself, both statically and dynamically.

A *multiparty-based system* is composed of a fixed set of participants and interactions. When it executes, it produces an *execution*, which may be viewed as a sequence of transitions between *configurations* that are produced by the occurrence of offering or synchronisation events. A configuration is an object that captures the data and the execution states of every participant in a system.

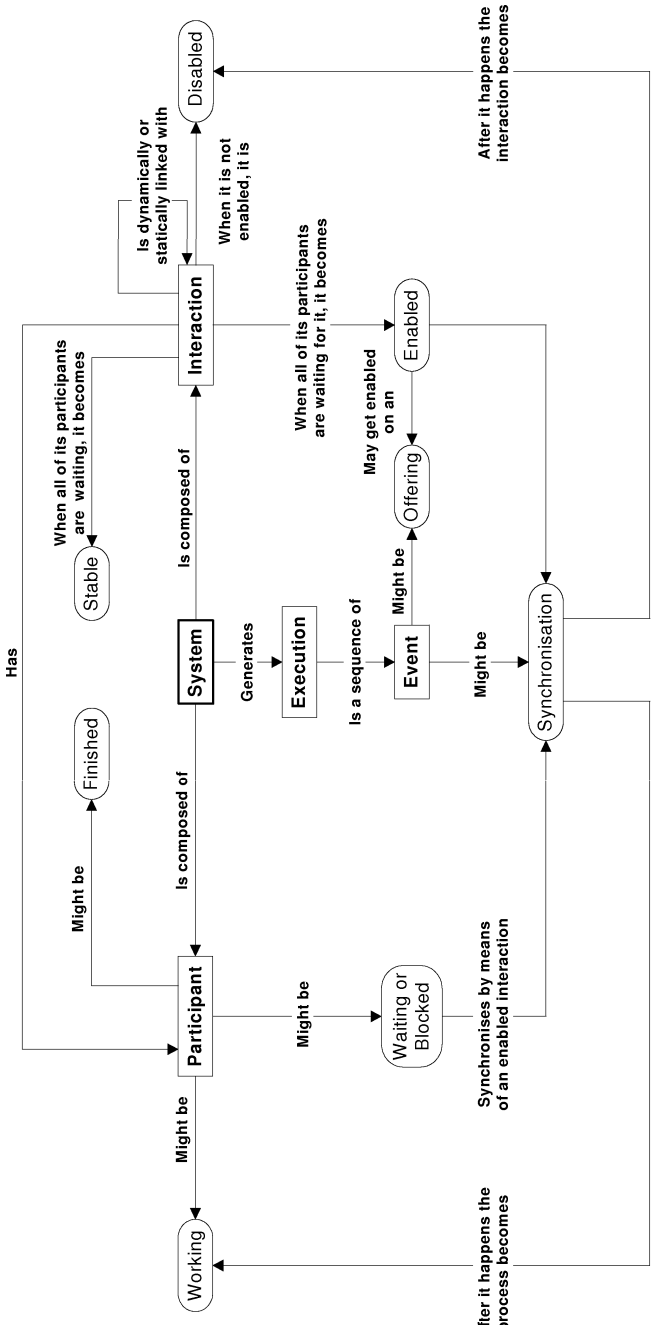


Figure 6.1: *A mental map of multiparty-based systems.*

An execution may be finite or infinite in theory, but they must be finite in practice (*cf.* Section §5.2). For an execution to be correct, it must satisfy the following safety properties:

i. When a participant offers to participate in a subset of interactions, it must remain blocked until one of them is executed, or, otherwise, it remains blocked forever.

ii. If a participant finishes, then it cannot offer to participate in any interaction again. Furthermore, the interactions in which it can participate cannot be executed again.

iii. A synchronisation by means of an interaction can happen iff it is enabled. After that, the interactions that are linked to it must become unstable and disabled, and the processes participating in it must begin to work.

6.2 A formal characterisation

In this section, we present a complete formalisation of the previous framework. We first describe the assumptions on which it builds, and then present a set of definitions that constitute its core; some useful definitions are then presented, and they are classified into two groups depending on whether they can be implemented or not; next, we formalise the safety properties any multiparty-based execution must satisfy.

6.2.1 Assumptions

Common assumptions in the context of concurrent systems are usually divided into two groups, namely: *physical assumptions* and *logical assumptions*. The first group allows to abstract from the many details concerning possible failures of the underlying hardware, whereas the latter focus on difficult-to-formalise, higher-level properties of the artefacts that compose a system.

To define our framework, we need to make the following common physical assumptions:

i. The computer does not fail and every participant that is ready to perform local computation proceeds in finite time. This is also known as the *minimal progress property*, and it is worth noting that it does not entail that there is a known bound on the time local computation may take to be executed.

ii. The communication links do not fail and every event can be detected in finite time by the participants of a system. Again, the time taken to transmit information must be finite, but the bound may be unknown *a priori*.

Furthermore, we need to make the following logical assumptions:

i. The behaviour of each participant is autonomous, and the code they run is independent from each other.

ii. Multiparty interactions are the sole means for a participant to synchronise and exchange information with others.

iii. There is an upper bound to the time each participant spends at working. Note that the bound must exist, but it need not be known beforehand.

None of these assumptions is crazy since our work does not focus on fault tolerance, and it is realistic to think that current technologies allow local computations to proceed in a reasonable time. Furthermore, they are common in the literature.

6.2.2 Core definitions

The core of our framework is a set of definitions by means of which we can define the concepts presented in Figure §6.1 rigorously. We use the dining philosophers system in Program §2.1 to illustrate some of them.

Definition 6.2.1 (Multiparty-based systems) *A system Σ is a two-tuple of the form (P_Σ, I_Σ) in which $P_\Sigma \neq \emptyset$ is a finite set of participants and $I_\Sigma \neq \emptyset$ is a finite set of interactions. We denote the set of participants that may offer to participate in interaction x as $\mathbb{P}(x)$, and the set of interactions in which participant p can offer to participate as $\mathbb{I}(p)$.*

Our example consists of a system with N philosopher participants called P_i and N fork participants called F_i ($i \in [1, N]$). They are synchronised by means of N Get_i interactions to take the forks and N Rel_i interactions to release them. The set of participants of interaction Get_i is $\mathbb{P}(Get_i)$=$\{F_{i-1}, P_i, F_i\}$, and the set of interactions in which F_i may participate is $\mathbb{I}(F_i)$=$\{Get_i, Rel_i, Get_{i+1}, Rel_{i+1}\}$.

Definition 6.2.2 (Events) *An event is a happening that induces a system to transit from a configuration to another. A configuration is an object that may be viewed as a snapshot of a system at run time. In our model, we distinguish between offering events and synchronisation events. The former are denoted as* $p.\chi$ *and indicate that participant* p *is ready to participate in any of the interactions in set* χ*; the latter are denoted as the name of an interaction and indicate that it has been selected for execution. An event of the form* $p.\emptyset$ *indicates that participant* p *is finished.*

For instance, when philosopher P_i offers to participate in interaction Get_i, an event of the form $P_i.\{Get_i\}$ occurs; similarly, when P_i takes its forks, an event of the form Get_i occurs and synchronises the execution of P_i, F_i, and F_{i-1} ($i \in [1, N]$).

Definition 6.2.3 (Executions) *An execution is a three-tuple of the form* (C_0, α, β)*, where* C_0 *is an initial configuration,* $\alpha = [C_1, C_2, C_3, \ldots]$ *is a maximal configuration trace, and* $\beta = [e_1, e_2, e_3, \ldots]$ *is a maximal event trace. Obviously, both* α *and* β *must have the same length. Given an execution* λ*, we usually denote its configuration trace as* λ_α *and its event trace as* λ_β*.*

Consider, for instance, the following execution:

$$\begin{aligned}
&\lambda = (C_0, \alpha, \beta)\\
&\alpha = [C_1 C_2 C_3 C_4 \ldots]\\
&\beta = [P_1.\{Get_1\}\, F_5.\{Get_5, Get_1\}\, F_1.\{Get_1, Get_2\}\, Get_1 \ \ldots]
\end{aligned}$$

Here, philosopher P_1 offers to participate in interaction Get_1, then fork F_5 offers to participate in either Get_5 or Get_1, and fork F_1 offers to participate in either Get_1 or Get_2. Interaction Get_1 becomes enabled at configuration C_3 and, in this case, it is selected and the execution continues.

Definition 6.2.4 (Semantics) *We refer to the rule that captures the semantics that controls the transition between consecutive configurations as* $\longrightarrow_L$*. For instance,* $C_i \xrightarrow{e_{i+1}}_L C_{i+1}$ *indicates that the system may transit from configuration* C_i *to configuration* C_{i+1} *on occurrence of event* e_{i+1}*. Thus, given an execution of the form* $(C_0, [C_1, C_2, C_3, \ldots], [e_1, e_2, e_3, \ldots])$*, we usually write it as:*

$$C_0 \xrightarrow{e_1}_L C_1 \xrightarrow{e_2}_L C_2 \xrightarrow{e_3}_L \cdots$$

Our example is implemented in IP_{CORE}, thus $\longrightarrow_L$ amounts for $\longrightarrow_{\text{IP}_{\text{CORE}}}$. Consult Reference [58] for a complete description of the semantics of the IP_{CORE} language or Appendix §B for a brief introduction.

Definition 6.2.5 (Participant states) *Participant* p *is waiting for a non-empty set of interactions* $\Upsilon \subseteq \mathbb{I}(p)$ *at the* i*-th configuration in execution* λ *iff it has offered to participate in a subset of interactions* $\chi \supseteq \Upsilon$ *and no interaction in* χ *has been selected since that moment. Participant* p *is finished at the* i*-th configuration in execution* λ *iff it has offered to participate in an empty set of interactions before. It is working iff it is neither finished nor waiting for an interaction to be executed.*

$$\begin{aligned}
\mathit{Waiting}(\lambda, p, \Upsilon, i) &\iff \exists \chi \supseteq \Upsilon, k \in [1, i] \cdot \\
&\qquad (\lambda_\beta(k) = p.\chi \wedge \nexists j \in (k, i] \cdot (\lambda_\beta(j) = x \wedge x \in \Upsilon)) \\
\mathit{Finished}(\lambda, p, i) &\iff \exists k \in [1, i] \cdot \lambda_\beta(k) = p.\emptyset \\
\mathit{Working}(\lambda, p, i) &\iff \neg\, \mathit{Finished}(\lambda, p, i) \wedge (\nexists \Upsilon \subseteq I_\Sigma \cdot \mathit{Waiting}(\lambda, p, \Upsilon, i))
\end{aligned}$$

In our example, philosopher P_j is waiting for the set of interactions $\{Get_j\}$ at configuration C_i if an event of the form $P_j.\{Get_j\}$ happened before C_i and interaction Get_j was not selected since that moment ($i \geq 1, j \in [1, N]$). They can never finish, but begin working immediately after they execute an interaction.

Definition 6.2.6 (Interaction states) *Interaction* x *is stable at the* i*-th configuration in execution* λ *iff its participants are finished or waiting for an interaction to be executed. It is enabled iff all of its participants are waiting for it to be executed; otherwise it is disabled.*

$$\begin{aligned}
\mathit{Stable}(\lambda, x, i) &\iff \forall p \in \mathbb{P}(x) \cdot (\mathit{Finished}(\lambda, p, i) \vee \exists \Upsilon \subseteq I_\Sigma \cdot \mathit{Waiting}(\lambda, p, \Upsilon, i)) \\
\mathit{Enabled}(\lambda, x, i) &\iff \forall p \in \mathbb{P}(x) \cdot \mathit{Waiting}(\lambda, p, \{x\}, i) \\
\mathit{Disabled}(\lambda, x, i) &\iff \neg\, \mathit{Enabled}(\lambda, x, i)
\end{aligned}$$

Interaction Get_j is enabled at the i-th configuration iff all of its participants (P_j, F_j, and F_{j-1}) are offering to participate in it at that configuration. Furthermore, Rel_j is stable at that configuration because its participants are waiting for Get_j to be executed. Note that enablement implies stableness, but disablement does not.

Definition 6.2.7 (Interaction linkage) *Two interactions are said to be statically linked iff they share a common participant; they are dynamically linked at the* i*-th configuration in execution* λ *iff there is a participant that is waiting for any of them to be executed.*

$$\begin{aligned}
\mathit{StaLinked}(x) &= \{y \in I_\Sigma \mid \mathbb{P}(x) \cap \mathbb{P}(y) \neq \emptyset\} \\
\mathit{DynLinked}(\lambda, x, i) &= \{y \in \mathit{StaLinked}(x) \mid \exists p \in P_\Sigma \cdot \mathit{Waiting}(\lambda, p, \{x, y\}, i)\}
\end{aligned}$$

6.2.3 Miscellaneous implementable sets

For the sake of readability, it is worth defining several sets of indices that may help define other concepts more easily. For every execution λ and interaction x, we define the following sets at the i-th configuration:

i. The set of indices up to i that identify the configurations at which interaction x is enabled (note that no interaction may be enabled at the initial configuration):

$$EnaSet(\lambda, x, i) = \{k \in [1, i] \mid Enabled(\lambda, x, k)\}$$

ii. The set of indices up to i that identify the configurations at which interaction x is stable:

$$StaSet(\lambda, x, i) = \{k \in [1, i] \mid Stable(\lambda, x, k)\}$$

iii. The set of indices up to i that identify the configurations at which interaction x is executed:

$$ExeSet(\lambda, x, i) = \{k \in [1, i] \mid \lambda_\beta(k) = x\}$$

iv. The set of indices up to i that identify the configurations at which p offers to participate in a subset of interactions $\Upsilon \subseteq \mathbb{I}(p)$:

$$OffSet(\lambda, \Upsilon, p, i) = \{k \in [1, i] \mid \lambda_\beta(k) = p.\chi \wedge \Upsilon \subseteq \chi\}$$

v. The set of indices up to i that identify the configurations at which p is waiting for x to be executed:

$$WaitSet(\lambda, x, p, i) = \{k \in [1, i] \mid Waiting(\lambda, p, \{x\}, i)\}$$

6.2.4 Miscellaneous non-implementable predicates

In the previous section, we defined some useful sets. Sometimes, it is also necessary to know if they are finite or not since this matters from a theoretical point of view. The following predicates can be useful to define some properties, but they are clearly not implementable, so they should be used with care.

$$\begin{aligned}
InfEna(\lambda, x) &\iff \nexists k \in \mathbb{N} \cdot |EnaSet(\lambda, x, |\lambda|)| < k \\
InfSta(\lambda, x) &\iff \nexists k \in \mathbb{N} \cdot |StaSet(\lambda, x, |\lambda|)| < k \\
InfExe(\lambda, x) &\iff \nexists k \in \mathbb{N} \cdot |ExeSet(\lambda, x, |\lambda|)| < k \\
InfOff(\lambda, x, p) &\iff \nexists k \in \mathbb{N} \cdot |OffSet(\lambda, \{x\}, p, |\lambda|)| < k \\
InfWait(\lambda, x, p) &\iff \nexists k \in \mathbb{N} \cdot |WaitSet(\lambda, x, p, |\lambda|)| < k
\end{aligned}$$

6.2.5 Safety properties

Obviously, not every combination of events and configurations is correct. The safety properties that every multiparty-based execution must obey are presented in the following definitions, where λ represents any of the executions generated by a system of the form $\Sigma = (I_\Sigma, P_\Sigma)$.

Definition 6.2.8 (Safe offers) *If participant p offers to participate in any of the interactions in set χ at the i-th configuration, then that participant cannot perform any action until one of the interactions in χ is executed; if none of them is executed again, then p must remain blocked forever.*

$$\begin{aligned}&\mathtt{SafeOffers}(\lambda) \iff \forall i > 0, p \in P_\Sigma, \chi \subseteq I_\Sigma \cdot \lambda_\beta(i) = p.\chi \Rightarrow \\ &((\nexists j > i, x \in \chi \cdot \lambda_\beta(j) = x \wedge \nexists k > i, \Upsilon \subseteq I_\Sigma \cdot \lambda_\beta(k) = p.\Upsilon) \vee \\ &(\exists j > i, x \in \chi \cdot \lambda_\beta(j) = x \wedge \nexists k \in (i, j), \Upsilon \subseteq I_\Sigma \cdot \lambda_\beta(k) = p.\Upsilon))\end{aligned}$$

Definition 6.2.9 (Safe terminations) *If participant p offers to participate in an empty set of interactions at the i-th configuration, then that participant can never again perform any action, and the interactions in which it may participate cannot be executed again.*

$$\begin{aligned}&\mathtt{SafeTerms}(\lambda) \iff \forall i > 0, p \in P_\Sigma \cdot \lambda_\beta(i) = p.\emptyset \Rightarrow \\ &(\nexists j > i, \chi \subseteq I_\Sigma \cdot \lambda_\beta(j) = p.\chi \wedge \nexists j > i, x \in \mathbb{I}(p) \cdot \lambda_\beta(j) = x)\end{aligned}$$

Definition 6.2.10 (Safe synchronisations) *If interaction x is executed at the i-th configuration, then it must be enabled at that configuration, the interactions that are statically linked with it must be disabled at the following configuration (note that this set includes x itself), and all of its participants must be working at the following configuration.*

$$\begin{aligned}&\mathtt{SafeSyncs}(\lambda) \iff \forall i > 0, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow \\ &(\mathtt{Enabled}(\lambda, x, i-1) \wedge \\ &\forall y \in \mathtt{StaLinked}(x) \cdot (\mathtt{Disabled}(\lambda, y, i) \wedge \neg\, \mathtt{Stable}(\lambda, y, i)) \wedge \\ &\forall p \in \mathbb{P}(x) \cdot \mathtt{Working}(\lambda, p, i))\end{aligned}$$

Note that the previous definition is not minimal since there are redundancies, *e.g.*, $\neg\, \mathtt{Stable}(\lambda, y, i)$ implies $\mathtt{Disabled}(\lambda, y, i)$, but we prefer not to minimise it since we think this formulation is more intuitive.

As an example, consider any of the executions whose event trace is of the form $[F_5.\{Get_5\}\, F_5.\{Get_1\}\, Get_1\, F_2.\emptyset\, F_2.\{Get_2\}\ldots]$. It does not fulfil any of the previous safety properties since F_5 offers to participate in two sets of interactions at different times without executing any of them in-between, Get_1 is executed before it becomes enabled, and F_2 offers to participate in interaction Get_2 immediately after it finishes.

6.3 Realising the benefits

In this section, we show how to use our framework to define some usual *safety* and *liveness* properties, fairness notions and executions metrics, which proves that it is general enough to be used in contexts other than this dissertation. However, it is not universal since, for instance, it cannot deal with oracle-based notions easily. Anyhow, it was not our goal to produce such a universal framework.

In the following examples, we assume that λ is an execution of system $\Sigma = (P_\Sigma, I_\Sigma)$. Unless otherwise stated, λ is assumed to be infinite.

Example 6.3.1 (Deadlock) *If λ is a finite execution, then it deadlocks iff all of the interactions in I_Σ are stable at the last configuration, but none of them is enabled.*

$$\mathtt{Deadlock}(\lambda) \iff \forall x \in I_\Sigma \cdot (\mathtt{Stable}(\lambda, x, |\lambda|) \wedge \neg\, \mathtt{Enabled}(\lambda, x, |\lambda|))$$

Example 6.3.2 (Conspiracy) *It is said that execution λ is conspiratorial iff there exist an interaction in I_Σ whose participants are ready to execute it infinitely many times but it does not get enabled infinitely many times. Note that this characterisation does not distinguish between inherent and non-inherent conspiracies since this would require to analyse the source code of the system that generated λ.*

$$\mathtt{Conspiratorial}(\lambda) \iff \exists x \in I_\Sigma \cdot (\forall p \in \mathbb{P}(x) \cdot \mathtt{InfWait}(\lambda, x, p) \wedge \neg\, \mathtt{InfEna}(\lambda, x))$$

Example 6.3.3 (Unconditional fairness) *It is said that λ is unconditionally fair iff every interaction that is continuously enabled is executed infinitely often.*

$$\mathtt{UF}(\lambda) \iff \forall x \in I_\Sigma \cdot (\forall i \geq 1 \cdot \mathtt{Enabled}(\lambda, x, i)) \Rightarrow \mathtt{InfExe}(\lambda, x)$$

Example 6.3.4 (Weak fairness) *It is said that λ is weakly fair iff every interaction that is enabled from a point in the execution onwards is executed infinitely often.*

$$WF(\lambda) \iff \forall x \in I_\Sigma \cdot (\exists i \geq 1 \cdot \forall j \geq i \cdot Enabled(\lambda, x, j)) \Rightarrow InfExe(\lambda, x)$$

Note that the requirements of UF and WF can never be satisfied according to the semantics of our framework. The reason lies in that a correct execution must satisfy the SafeSyncs safety property, which implies that after an interaction is executed it must become disabled necessarily. *Cf.* Section §3.3.2 for further details.

Example 6.3.5 (Strong fairness) *It is said that λ is strongly fair iff every interaction that is infinitely often enabled is executed infinitely often.*

$$SF(\lambda) \iff \forall x \in I_\Sigma \cdot (InfEna(\lambda, x) \Rightarrow InfExe(\lambda, x))$$

Example 6.3.6 (Rejection ratio) *If λ is a finite execution, the rejection ratio of interaction x is defined as one minus the quotient between the number of times x has been executed and the number of times it has been enabled.*

$$RejectionRatio(\lambda, x) = 1 - \frac{|\, ExeSet(\lambda, x, |\lambda|)\,|}{|\, EnaSet(\lambda, x, |\lambda|)\,|}$$

Note that if the rejection ratio tends to 0, this means that interaction x has been executed most times it was enabled; on the contrary, if it tends to 1, this implies that it has been enabled many times, but it has not been executed so often as was desired.

Example 6.3.7 (Enablement degree) *If λ is a finite execution, the enablement degree of interaction x is defined as the quotient between the times it is executed and the average number of times its participants offer to participate in it.*

$$EnablementDegree(\lambda, x) = \frac{|\, ExeSet(\lambda, x, |\lambda|)\,|}{\left|\bigcup_{p \in \mathbb{P}(x)} OffSet(\lambda, \{x\}, p, |\lambda|)\right| / |\,\mathbb{P}(x)|}$$

In this expression, the numerator counts the number of times that interaction x has been executed, and the denominator counts the average number of times that the participants have offered to participate in x. Thus, it has to range between 0 and 1, namely: if it tends to 1, this means that participants do not have to insist on offering to participate in x to execute it; if it tends to 0, they then have to offer to participate in x many times before executing it.

Example 6.3.8 (Static linkage degree) *The static linkage degree of participant* p *is defined as the quotient between the number of interactions that are statically linked to it, except for itself, and the maximum number of interactions in which it can offer to participate.*

$$\mathtt{StaLinkageDegree}(p) = \frac{\left| \bigcup_{q \in P_\Sigma \setminus \{p\}} \mathbb{I}(p) \cap \mathbb{I}(q) \right|}{|\mathbb{I}(p)|}$$

If this expression tends to 1, that means that participant p shares most of its interactions with others participants; on the contrary, if it tends to 0, it shares very few interactions, including none at all.

6.4 Summary

In this chapter, we have presented a rigorous framework in which we have defined the multiparty interaction model with which we deal. Furthermore, we have shown that it can be used to define common safety, liveness and fairness notions, although it was not our goal to devise a universal framework.

Earlier versions were published at the PROLE'01 Conference [120], the PROLE'02 Conference [123], the Colombian Journal of Computation [124], and the Euro-Par'02 Conference [122]. The final results were published in Concurrency and Computation: Practice & Experience [125].

Chapter 7

A new fairness notion

Equal opportunity means that everyone
will have a fair chance at being incompetent.

Laurence J. Peter, 1919–1988
Educator and writter

Fair finiteness and conspiracies are the motto of this dissertation. In this chapter, we present the notion we have devised to solve these problems. In Section §7.1, we present an overall idea that may help the reader understand the definition in Section §7.2 more easily; in Section §7.3, we prove that the notion solves the aforementioned problems and preserves strong fairness for infinite executions; finally, we summarise our contributions and publications in Section §7.4.

7.1 Introduction

k-*conspiracy-free fairness* is a new fairness notion whose requirements are not trivially fulfilled by all finite execution and can prevent conspiracies from happening. The idea behind it is to identify potentially conspiratorial situations and put a bound on the number of times they can occur. Note that conspiracies may happen in infinite executions only, but it is easy to detect the configurations at which a conspiracy may possibly be being hatched. Roughly speaking, this may happen if an interaction is executed but the interactions with which it is linked are still unstable. Thus, conspiracies can be avoided by delaying the execution of the interactions that become enabled too frequently in the presence of unstable linked interactions.

Our first attempt to devise a notion able to solve both fair finiteness and conspiracies led us to *strong* k-*fairness* [122]. In this notion, we took the set of interactions linked at run time into account, which allowed to alleviate conspiracies but not to solve them completely (*cf.* Appendix §A). On the contrary, k-conspiracy-free fairness takes the statically linked interactions into account and allows to eliminate conspiracies altogether.

The constraints that this new notion imposes on the systems to which it can be applied are very light, namely: the physical and logical assumptions presented in Section §6.2.1 must be satisfied by the hardware and the compiler used, and the system must be fully connected. Intuitively, a system is fully-connected when every couple of participants can communicate directly or indirectly by means of one or more interactions, *i.e.*, the graph that connects participants and interactions is bipartite and connected. Figure §7.1 shows several fully-connected systems. Note that they are disjoint since they do not share any interaction, thus it is impossible for a participant in a fully-connected system to communicate with a participant in another fully-connected system, which implies that the executions of fully-connected systems are independent from each other.

7.2 k-conspiracy-free fairness

According to Attie *et al.*'s definition [10], a conspiracy against an interaction happens iff its participants are ready to participate in it infinitely many times, but it does not get enabled infinitely many times because some of them commit to another interaction too rashly.

Therefore, we can prevent conspiracies by insuring that every time an interaction is executed, all of the interactions that are statically linked with it are

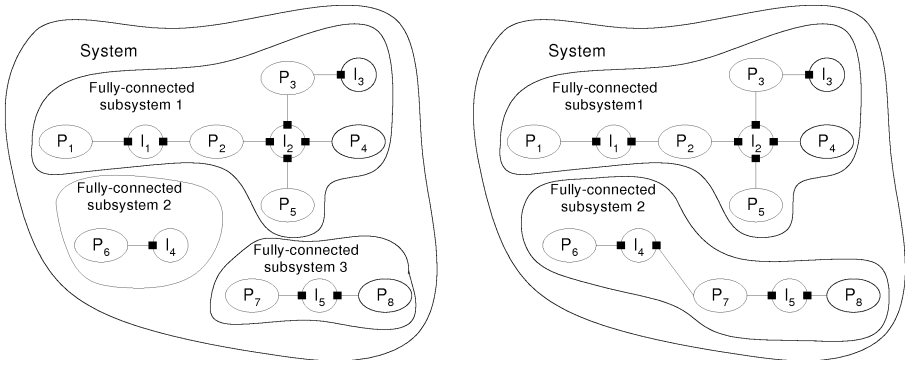


Figure 7.1: *A couple of fully-connected systems.*

stable, thus giving them a chance to become enabled. Formally, if the following formula holds, then execution λ does not have any conspiracies:

$$\forall i > 0, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow \neg \, \mathrm{PotConsp}(\lambda, x, i) \quad (7.1)$$

where $\mathrm{PotConsp}(\lambda, x, i)$ holds as long as there is an unstable interaction that is statically linked to x at the i-th configuration of execution λ. Formally,

$$\mathrm{PotConsp}(\lambda, x, i) \iff \exists y \in \mathrm{StaLinked}(x) \cdot \neg \, \mathrm{Stable}(\lambda, y, i)$$

Note that Formula §7.1 prevents conspiracies from happening, but it does not force interactions that are enabled infinitely often from been neglected forever. One might think that strengthening it as follows might suffice:

$$\mathrm{SF}(\lambda) \wedge \forall i > 0, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow \neg \, \mathrm{PotConsp}(\lambda, x, i) \quad (7.2)$$

where $\mathrm{SF}(\lambda)$ was defined in Example §6.3.5 and holds as long as λ is strongly fair. However, this would not solve the fair finiteness problem since SF holds trivially for every finite execution. A better idea to prevent an interaction that is enabled infinitely often from not executing is to take its age into account, so that an interaction can be selected as long as it is the oldest amongst the interactions that are enabled. The age of an interaction can be interpreted as the time that has elapsed since it was executed for the last time, but given that we do not take real-time constraints into account, it suffices to define it as the number of configurations that have elapsed since it was executed for the last time.

Therefore, the criterion in Formula §7.1 may be refined as follows:

$$\forall i > 0, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow (\neg\, \mathrm{PotConsp}(\lambda, x, i) \wedge \mathrm{OldestEna}(\lambda, x, i)) \quad (7.3)$$

where $\mathrm{OldestEna}$ is defined as follows:

$$\mathrm{OldestEna}(\lambda, x, i) \iff \forall y \in I_\Sigma \setminus \{x\} \cdot \mathrm{Enabled}(\lambda, y, i) \Rightarrow \mathrm{Age}(\lambda, y, i) < \mathrm{Age}(\lambda, x, i)$$

$$\mathrm{Age}(\lambda, x, i) = \begin{cases} i - \max \mathrm{ExeSet}(\lambda, x, i-1) & \text{if } \mathrm{ExeSet}(\lambda, x, i-1) \neq \emptyset \\ i + \Phi(x) & \text{otherwise} \end{cases}$$

The initial age of an interaction poses an important problem. Intuitively, it should be ∞, but this would not allow us to compare them in order to determine which one is the most priority initially. To solve this problem, we assume that there exists an arbitrary total order $\prec$ on the interactions that helps us define map Φ as follows:

$$\Phi(x) = |\{y \in I_\Sigma \mid y \prec x\}| + 1$$

Note that $\Phi(x) \in [1, |I_\Sigma|]$ for all $x \in I_\Sigma$, and this guarantees that no two different interactions can have the same age at any configurations. As we prove later, this random initial age does not affect the correctness of the notion.

The criterion in Formula §7.3 prevents conspiracies from happening and ensures that every interaction that is enabled infinitely often is executed infinitely often, but there is a subtle point: it requires *the whole set* of interactions that are statically linked with the interaction that is executed to be stable; if this happens naturally, the criterion is fine, but, otherwise, it forces the execution of some interactions to be delayed. This can amount to inefficiency and should be avoided if possible by introducing a conspiracy threshold that puts a bound on the number of times an interaction can be executed without waiting for the interactions that are linked to it to become stable. This threshold may help tune the criterion to the features of the system under consideration, namely: if it is null, then the pace at which a group of linked interactions progresses is determined by the slowest participant since they all need to wait for the interaction in which it participates to become stable; on the contrary, if it is sufficiently large, interactions are selected as soon as possible once they are detected to be enabled.

These observations led us to formulate our fairness notion as follows:

Definition 7.2.1 (k-conspiracy-free fairness) *Let λ be an execution of a fully-connected system of the form $\Sigma = (P_\Sigma, I_\Sigma)$. λ is said to be k-conspiracy-free fair iff interactions are executed as long as they are not in a potentially conspiratorial situation and they are the oldest, or they are in a potentially conspiratorial situation, but they have not been selected more than k times in such situations.*

$$\mathcal{CFF}_k(\lambda) \iff \forall i > 0, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow$$
$$(\neg \mathrm{PotConsp}(\lambda, x, i) \wedge \mathrm{OldestEna}(\lambda, x, i) \vee \sum_{j=0}^{i} \mathrm{PotConsp}(\lambda, x, j) < k)$$

It is worth noting that implementing k-conspiracy-free fairness does not amount to a *void obligation* since not every finite execution satisfies its requirements trivially. Any execution in which an interaction is executed more than k times in the presence of unstable, linked interactions can be noticed in finite time. Thus, it is possible for a scheduler to determine if the execution of an interaction can violate the notion or not.

7.3 Correctness

Our goal in this section is to prove that k-conspiracy-free fairness solves both the fair finiteness problem and conspiracies, but still preserves strong interaction fairness.

Lemma 7.3.1 *Let λ be an infinite k-conspiracy-free fair execution for some $k \in \mathbb{N}$, and z an interaction that is executed finitely many times along λ. There exists a configuration $C_{r_1^*}$ in λ such that every time an interaction that is statically linked with z executes at $C_{r_1^*}$ or the succeeding configurations, z is stable.*

Proof Note that the set of interactions can be partitioned into two disjoint groups I and F so that I is the subset of interactions that are executed infinitely many times, whereas F is the subset of interactions that are executed finitely many times. Clearly, $I \neq \emptyset$ since, otherwise, λ would be finite, and $F \neq \emptyset$ since z executes finitely many times according to our hypothesis. Formally,

$$I_\Sigma = I \cup F, I \cap F = \emptyset$$
$$\forall x \in I \cdot \mathrm{InfExe}(\lambda, x)$$
$$\forall x \in F \cdot \neg \mathrm{InfExe}(\lambda, x)$$

Furthermore, there must exists an index p such that no interaction in set F can be executed from configuration $\lambda_\alpha(p)$ onwards. This implies that z cannot become enabled after that configuration or our hypothesis would be violated.

Set I can be further partitioned into sets L and N so that the former contains the interactions that are linked with z, and the latter the interactions that are not linked with it. Note that $L \neq \emptyset$ since, otherwise, the system under consideration would not be fully-connected and k-conspiracy-free fairness could not be applied. Formally,

$$I = L \cup N, L \cap N = \emptyset$$
$$\forall x \in L \cdot x \in StaLinked(z)$$
$$\forall x \in N \cdot x \notin StaLinked(z)$$

L can be further partitioned into two sets that hold the interactions that can be selected infinitely many times without exceeding the potential conspiracy threshold, and the rest. Formally,

$$L = L_N \cup L_E, L_N \cap L_E = \emptyset$$
$$\forall x \in L_N, q \geq p \cdot \sum_{j=0}^{q} PotConsp(\lambda, x, j) < k$$
$$\forall x \in L_E \cdot \exists q \geq p \cdot \sum_{j=0}^{q} PotConsp(\lambda, x, j) = k$$

Now, we can differentiate two cases:

i. For all $x \in L_N$, there must exist an index $p_x \geq p$ such that from configuration $\lambda_\alpha(p_x)$ onwards, x is always executed in a situation in which no potential conspiracy is detected, and it is the oldest amongst the enabled interactions. Let p_1 be $\max_{x \in L_N}\{p_x\}$.

ii. For all $x \in L_E$, there must exist an index p_2 such that for all $q \geq p_2$, $\sum_{j=0}^{q} PotConsp(\lambda, x, j) = k$. This implies that every time x is executed, it must execute in a non-potentially conspiratorial situation and it must be the oldest, which implies that z must be stable.

Therefore, any index $r_1^* \geq \max\{p_1, p_2\}$ satisfies the lemma. □

Lemma 7.3.2 *Let λ be an infinite k-conspiracy-free fair execution for some $k \in \mathbb{N}$, and z an interaction that is executed finitely many times along λ. There exists a configuration $C_{r_2^*}$ in λ such that every time an interaction that is not statically linked with z executes at $C_{r_2^*}$ or the succeeding configurations, it is not in a potentially conspiratorial situation and it is the oldest amongst the set of enabled interactions.*

Proof From the proof of Lemma §7.3.1, it follows that there exists a subset of interactions N such that these interactions are executed infinitely many times and they are not linked with z. N can be further partitioned into N_N and N_E so that

$$N = N_N \cup N_E, N_N \cap N_E = \emptyset$$
$$\forall x \in N_N, q \geq p \cdot \sum_{j=0}^{q} \mathrm{PotConsp}(\lambda, x, j) < k$$
$$\forall x \in N_E \cdot \exists q \geq p \cdot \sum_{j=0}^{q} \mathrm{PotConsp}(\lambda, x, j) = k$$

Now, we can differentiate two cases again:

i. For all $x \in N_N$, there must exist an index $p_x \geq p$ such that from configuration $\lambda_\alpha(p_x)$ onwards, the counter of potentially conspiratorial situations does not change, or otherwise x would belong to set N_E instead. Let p_1 be $\max_{x \in N_N}\{p_x\}$.

ii. For all $x \in N_E$, there must exist an index p_2 such that for all $q \geq p_2$, $\sum_{j=0}^{q} \mathrm{PotConsp}(\lambda, x, j) = k$.

Therefore, any index $r_2^* \geq \max\{p_1, p_2\}$ satisfies the lemma since no interaction in N is executed in a potentially conspiratorial situation from this configuration onwards. □

Theorem 7.3.1 (Fair finiteness) *Given a $k \in \mathbb{N}$, not every finite execution fulfils the requirements of k-conspiracy-free fairness.*

Proof Proving this theorem is straightforward since it amounts to finding an execution that is not k-conspiracy-free fair for some $k \in \mathbb{N}$. Consider, for instance, the following program, which denotes a fully-connected system:

```
S :: [P || Q || R], where
P :: *[ A⟨⟩ → skip ];
Q :: *[ A⟨⟩ → skip [] B⟨⟩ → skip ];
R :: *[ B⟨⟩ → skip ].
```

Let λ be any of the possible executions whose event trace is of the form $[P.\{A\}\,Q.\{A,B\}\,\boldsymbol{A}]^{n} \frown [R.\{B\}\,Q.\{A,B\}\,\boldsymbol{B}\,P.\{A\}\,Q.\{A,B\}\,\boldsymbol{A}]^{m}$. It is not k-conspiracy-free fair for any $k < n$ because interaction A is executed more than k times in situations in which interaction B, which is statically linked with it, is not stable. □

Theorem 7.3.2 (Conspiracies) *No execution that satisfies the requirements of* k*-conspiracy-free fairness may suffer from conspiracies other than inherent conspiracies for any* $k \in \mathbb{N}$.

Proof Roughly speaking, we have to prove that if there exist an execution λ such that there is an interaction z whose participants wait for it to become enabled infinitely many times and it does not become enabled infinitely many times, but it satisfies the requirements of k-conspiracy-free fairness for any $k \in \mathbb{N}$, then the conspiracy against z must be inherent.

From the proof of Lemma §7.3.1, it follows that every time an interaction in L is executed at the r_1^*-th or the subsequent configurations, z must be stable. Therefore, the only possibility for this interaction not to become enabled is that there exists an inherent conspiracy against it that prevents its participants from being ready to execute it at the same time infinitely many times. Note that every time an interaction in set N is executed at the r_1^*-th or the subsequent configurations, z need not be stable, but these interactions cannot conspire against it since they are not linked with it. □

Theorem 7.3.3 (Strong fairness) *Let* λ *be an execution. If it satisfies the requirements of* k*-conspiracy-free fairness, it then satisfies the requirements of strong fairness.*

Proof We prove this theorem by *reductio ad absurdum*, since a contradiction may be reached if we assume that there is an interaction z that is enabled infinitely many times but executes finitely many times only.

From the proofs of Lemmata §7.3.1 and §7.3.2, it follows that for all index $r \geq \max\{r_1^*, r_2^*\}$, the interactions that are statically linked with z can execute as long as z is stable, and the rest of interactions execute in non-potentially conspiratorial situations. Consider an index $r^* \geq r$ such that every interaction that executes from configuration $\lambda_\alpha(r^*)$ onwards has already been executed at least once. Such index must exists or the execution would be finite, otherwise. In other words, r^* must satisfy that

$$\forall i \geq r^*, x \in I_\Sigma \cdot \lambda_\beta(i) = x \Rightarrow (x \in I \wedge \exists j < i \cdot \lambda_\beta(j) = x)$$

Let $p \geq r^*$ be any of the infinitely many indexes that denote a configuration at which z is enabled. We can distinguish the following cases:

i. If z has never been executed before configuration $\lambda_\alpha(p)$, then it follows that

$$-\Phi(z) < \max \mathit{ExeSet}(\lambda, x, p) \text{ for all } x \in I,$$

since $\Phi(x) \in [1, |I_\Sigma|]$ for all $x \in I_\Sigma$, and $\mathit{ExeSet}(\lambda, x, p) \geq 1$ because every interaction that executes from configuration $\lambda_\alpha(p^*)$ onwards has executed at least once before. Therefore, it follows that

$$\begin{aligned} &p - -\Phi(z) > p - \max \mathit{ExeSet}(\lambda, x, p), \text{ then} \\ &p + \Phi(z) > p - \max \mathit{ExeSet}(\lambda, x, p), \text{ then} \\ &\mathit{Age}(\lambda, z, p) > \mathit{Age}(\lambda, x, p) \end{aligned}$$

Let x be the interaction that was executed at the p-th configuration. If it was linked with z, then this is a contradiction since z was stable according to Lemma §7.3.1 and it was the oldest; otherwise, Lemma §7.3.2 guarantees that x must have been executed in a situation in which there was no potential conspiracy and it was the oldest, which contradicts the hypothesis again.

ii. If z was executed before it becomes enabled at configuration $\lambda_\alpha(p)$, then it follows that

$$\forall x \in X_I \cdot \mathit{Age}(\lambda, x, p) < \mathit{Age}(\lambda, z, p)$$

since every interaction in I must have been executed at least once after z was executed for the last time, *i.e.*, z is older than the rest of interactions, which contradicts the decision to execute an interaction other than it according to Lemmata §7.3.1 and §7.3.2.

These contradictions conclude the proof. □

7.4 Summary

In this chapter, we have presented our main contribution, which consists of a new fairness notion that deals with both fair finiteness and conspiracies, and can be applied to any fully-connected system without requiring it to be

in normal form or the interactions to be conspiracy resistant. We have also presented a formal proof to support these claims.

The notion was published in Concurrency and Computation: Practice & Experience [125], and previous results on strong k-fairness were published at the Euro-Par'02 Conference [122], the PROLE'02 Conference [123], and the Colombian Journal of Computation [124].

Chapter 8

Implementing our framework and our notion

If it's your idea, you get to implement it.

Leland E. Modesitt, Jr., 1943–
Science fiction and fantasy writer

Although our theoretical framework is quite expressive, it would not be practical enough unless we devised a finite-state implementation. In this chapter, we report on our results and apply them to the implementation of a scheduler for k*-conspiracy-free fairness. In Section §8.1, we present a brief introduction to the chapter; in Section §8.2 we describe the implementation of our framework and prove that it is correct; in Section §8.3, we describe how to implement* k*-conspiracy-free fairness on top of the implementation presented previously; finally, in Section §8.4, we summarise the chapter and our main publications.*

8.1 Introduction

Our theoretical framework is quite expressive, and it allows to describe properties that depend on the partial execution of a program, as well as properties that depend on the whole execution. Obviously, the latter are not implementable in general since they require knowledge about the future, but the former can be implemented efficiently by maintaining a set of incremental data structures to record information about past events.

We describe the implementation by means of Plotkin's transition rules since this technique has proved to be both simple and powerful [112]. Note that our goal is to demonstrate that the framework can be implemented using finite-state machines, but not to report on the many optimisations that are possible. We just point out some tips that may help optimise the implementation, but we do not present many details since they fall outside the scope of this dissertation. In Reference [118], we report on a $\jmath^\sharp$ implementation that includes many fine-grained optimisations.

8.2 Implementing the theoretical framework

In the following sections, we first describe the configurations that allow to implement our theoretical framework and how they are updated on occurrence of offer, termination or synchronisation events. We use `typewriter letters` to distinguish between a function or a predicate in the theoretical framework and its corresponding implementation. Next, we prove that the implementation is correct and preserves the safety properties that characterise a correct multiparty execution. Finally, we demonstrate that it is not necessary to rely on any fairness assumption to implement it.

8.2.1 Configurations

The configurations we need are three-tuples of the form (C, φ, ϑ), namely: C is an IP_{CORE} configuration, and its structure depends on the semantics of this language [58]; φ is an *offer map* that associates every interaction with the subset of participants that are currently waiting for it to be executed, *i.e.*, $\varphi\colon I_\Sigma \to 2^{P_\Sigma}$, $\operatorname{dom}\varphi = I_\Sigma$; ϑ is a *finished set* that records the participants that have finished their execution, *i.e.*, $\vartheta \subseteq 2^{P_\Sigma}$.

It is worth noting that our configurations do not record any information about the interactions that are executed or the configurations at which they

become enabled. Thus, we cannot implement the miscellaneous sets in Section §6.2.3 since they would require to maintain a few structures whose size would increase monotonically as the length of the execution increases. As we mentioned there, they were introduced for the sake of readability, but they are not strictly necessary.

We use the following functions to update configurations:

$$\begin{aligned}
\texttt{AddOffer}(\varphi, p, \chi) &= \{x \mapsto \varphi(x) \mid x \in I_\Sigma \setminus \chi\} \cup \{x \mapsto \varphi(x) \cup \{p\} \mid x \in \chi\} \\
\texttt{RemoveOffer}(\varphi, x) &= \{x \mapsto \varphi(x) \setminus \mathbb{P}(x) \mid x \in I_\Sigma\} \\
\texttt{AddFinished}(\vartheta, p) &= \vartheta \cup \{p\}
\end{aligned}$$

$\texttt{AddOffer}(\varphi, p, \chi)$ is used every time participant p offers to participate in the non-empty set of interactions χ, and it returns a map in which p is added to the set of participants that are waiting for any interaction in χ to be executed; contrarily, $\texttt{RemoveOffer}(\varphi, x)$ is used every time an interaction is selected, and it removes its participants from φ; $\texttt{AddFinished}(\vartheta, p)$ is used every time a participant finishes to add it the finished set.

8.2.2 Predicates and functions

The predicates and functions of our framework, except for the miscellaneous sets, are easily mapped onto their implementation using the information contained in a configuration. Let (C, φ, ϑ) be a framework configuration, then:

$$\begin{aligned}
\texttt{Waiting}(\varphi, p, \Upsilon) &\iff \forall x \in \Upsilon \cdot p \in \varphi(x) \\
\texttt{Finished}(\vartheta, p) &\iff p \in \vartheta \\
\texttt{Working}(\varphi, \vartheta, p) &\iff \neg\,\texttt{Finished}(\vartheta, p) \wedge \nexists x \in I_\Sigma \cdot \texttt{Waiting}(\varphi, p, \{x\}) \\
\texttt{Stable}(\varphi, \vartheta, x) &\iff \forall p \in \mathbb{P}(x) \cdot (\texttt{Finished}(\vartheta, p) \vee \exists y \in I_\Sigma \cdot \texttt{Waiting}(\varphi, p, \{y\}) \\
\texttt{Enabled}(\varphi, x) &\iff \varphi(x) = \mathbb{P}(x) \\
\texttt{Disabled}(\varphi, x) &\iff \neg\,\texttt{Enabled}(\varphi, x) \\
\texttt{StaLinked}(x) &= \{y \in I_\Sigma \mid \mathbb{P}(y) \cap \mathbb{P}(x) \neq \emptyset\} \\
\texttt{DynLinked}(\varphi, x) &= \{y \in \texttt{StaLinked}(x) \mid \varphi(y) \cap \varphi(x) \neq \emptyset\}
\end{aligned}$$

Note that `Working` and `Stable` may require to analyse the whole set of interactions to determine whether a process is waiting for one of them to be executed or not. For the sake of efficiency, an actual implementation should avoid examining this set in sequence and should maintain a map to associate

each participant with the set of interactions in which it has offered to participate. For instance, if this map is of the form $\omega\colon P_\Sigma \to 2^{I_\Sigma}$, then `Working` can be re-formulated as follows:

$$\texttt{Working}(\omega, \vartheta, p) \iff \neg\,\texttt{Finished}(\vartheta, p) \wedge \omega(p) = \emptyset$$

Similarly, the re-evaluation of `StaLinked` should be avoided by caching the sets of statically linked interactions when they are computed for the first time. Clearly, these design decisions depend completely on both the data structures used to implement the maps and the typical size of I_Σ. This is the reason why we do not go into deeper details here, but they may be found in Reference [118].

8.2.3 Semantics

The semantics of our framework allow to monitor the execution of a program to record the information we need. The initial configuration is of the form (φ_0, ϑ_0), where $\varphi_0(x) = \emptyset$ for all $x \in I_\Sigma$ and $\vartheta_0 = \emptyset$. From this configuration onwards, the system is controlled by the Offer, the Termination, and the Synchronisation Rules presented below.

i. The *Offer Rule* describes how our framework reacts on occurrence of an event the form $p.\chi$ in which $\chi \neq \emptyset$. In such cases, we need to update the offer map corresponding to the current configuration to record that there is a new set of offers that might enable one or more interactions. Formally,

$$\frac{C_i \xrightarrow{p.\chi}_{L} C_{i+1}}{(C_i, \varphi_i, \vartheta_i) \xrightarrow{p.\chi}_{FW} (C_{i+1}, \varphi_{i+1}, \vartheta_{i+1})} \text{ if } \chi \neq \emptyset$$

where $\varphi_{i+1} \triangleq \texttt{AddOffer}(\varphi_i, p, \chi)$
$\vartheta_{i+1} \triangleq \vartheta_i$

ii. The *Termination Rule* describes how our framework reacts on occurrence of an event of the form $p.\emptyset$. In such cases, we only need to add p to the finished set by means of function `AddFinished`. Formally,

$$\frac{C_i \xrightarrow{p.\emptyset}_{I.} C_{i+1}}{(C_i, \varphi_i, \vartheta_i) \xrightarrow{p.\emptyset}_{FW} (C_{i+1}, \varphi_{i+1}, \vartheta_{i+1})}$$

where $\varphi_{i+1} \triangleq \varphi_i$
$\vartheta_{i+1} \triangleq \texttt{AddFinished}(\vartheta_i, p)$

iii. The *Synchronisation Rule* describes how our framework reacts when the execution reaches a configuration in which an enabled interaction can be executed. In this case, we need to remove the participants of the interaction selected from the image of the offer map, which might disable some interactions. Formally,

$$\frac{C_i \xrightarrow{x}_L C_{i+1}}{(C_i, \varphi_i, \vartheta_i) \xrightarrow{x}_{FW} (C_{i+1}, \varphi_{i+1}, \vartheta_{i+1})} \text{ if } \texttt{Enabled}(\varphi_i, x)$$

where $\varphi_{i+1} \triangleq \texttt{RemoveOffer}(\varphi_i, x)$

$\vartheta_{i+1} \triangleq \vartheta_i$

Note that the implementation of our theoretical framework does not aim at controlling the execution at all, but at recording information about it that is necessary to implement our notion or other properties. For instance, we cannot decide when a process may produce an offer or a termination event since this depends on the semantics of the language under consideration, which is represented by means of the generic transition rule $\longrightarrow_L$. In this case, $\longrightarrow_L$ amounts for $\longrightarrow_{IP_{CORE}}$, but the framework can easily accommodate other languages as long as the interaction model they support is the same as IP_{CORE}. This makes the implementation more abstract and general, and very attractive since it can easily deal with, say, $J^\sharp$ and an appropriate library to implement multiparty interactions.

8.2.4 Correctness of the implementation

In this section, we prove that the predicates and functions we have presented in Section §8.2.2 can be viewed as correct implementations at the current configuration of the predicates and functions we presented in our theoretical framework in Section §6.2.2.

To understand what we mean by "correct implementation at the current configuration", consider any of the executions our framework can generate. Generically, it must be of the form $\lambda = (D_0, [D_1\, D_2\, D_3\, \ldots], [e_1\, e_2\, e_3\, \ldots])$ where each D_i is a three-tuple of the form $(C_i, \varphi_i, \vartheta_i)$ for all $i \geq 0$. We define its projection on IP_{CORE} as $\lambda|_{IP_{CORE}} = (C_0, [C_1\, C_2\, C_3\, \ldots], [e_1\, e_2\, e_3\, \ldots])$. Given this definition, we say that predicate $\texttt{pred}(x_1, x_2, \ldots, x_n)$ in our theoretical framework is implemented by predicate $\texttt{pred}(y_1, y_2, \ldots, y_m)$ at configuration D_i iff the following conditions are satisfied for all appropriate parameters x_i, y_j ($i \in [1, n], j \in [1, m]$):

if $\mathtt{pred}(x_1, x_2, \ldots, x_n)$ holds at C_i, then $\mathtt{pred}(y_1, y_2, \ldots, y_m)$ holds at D_i
if $\neg\mathtt{pred}(x_1, x_2, \ldots, x_n)$ holds at C_i, then $\neg\mathtt{pred}(y_1, y_2, \ldots, y_m)$ holds at D_i

Similarly, function $\mathtt{func}(x_1, x_2, \ldots, x_n)$ in our theoretical framework is implemented by function $\mathtt{func}(y_1, y_2, \ldots, y_m)$ iff the following condition is satisfied for all appropriate parameters x_i, y_j ($i \in [1, n], j \in [1, m]$) and result X:

$$\text{if } \mathtt{func}(x_1, x_2, \ldots, x_n) = X, \text{ then } \mathtt{func}(y_1, y_2, \ldots, y_m) = X$$

Theorem 8.2.1 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current configuration, then predicate* $Waiting(\lambda, p, \Upsilon, i)$ *is implemented by predicate* $\mathtt{Waiting}(\varphi_i, p, \Upsilon)$.

Proof We prove this theorem by *reductio ad absurdum*. First, assume that predicate $Waiting(\lambda, p, \Upsilon, i)$ holds whereas $\mathtt{Waiting}(\varphi_i, p, \Upsilon)$ does not hold. In other words, there exists an interaction $y \in \Upsilon$ such that $p \notin \varphi_i(y)$. According to Definition §6.2.5, if $Waiting(\lambda, p, \Upsilon, i)$ holds, then p offered to participate in a set of interactions $\chi \supseteq \Upsilon$ at a previous configuration C_j ($j < i$), but no interaction in set χ has been executed since that moment. The Offer Rule forced map φ_j to be updated by means of `AddOffer` so that $p \in \varphi_{j+1}(x)$ for every $x \in \chi$, and $\Upsilon \subseteq \chi$. The only way for p to be removed from the offer map is by means of function `RemoveOffer`, and this can happen iff the Synchronisation Rule is applied. This, obviously, cannot happen because, otherwise, it would contradict the initial hypothesis since $Waiting(\lambda, p, \Upsilon, i)$ would not hold.

Now, assume that predicate $Waiting(\lambda, p, \Upsilon, i)$ does not hold but predicate $\mathtt{Waiting}(\varphi_i, p, \Upsilon)$ holds. This implies that $p \in \varphi_i(x)$ holds for every $x \in \Upsilon$, which, in turn, implies that there exists a configuration D_j ($j < i$) on which the Offer Rule was applied on occurrence of an event of the form $p.\chi$, $\chi \supseteq \Upsilon$. Furthermore, no interaction in set χ has been executed since then because, otherwise, the Synchronisation Rule would have been applied and it would have removed p from the offer map of every interaction in set χ. This contradicts the initial hypothesis since it implies that $Waiting(\lambda, p, \Upsilon, k)$ should hold for any k in the interval $[j, i]$. □

Corollary 8.2.2 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current configuration, then* $\varphi_i(x)$ *denotes the set of participants that are waiting for any* $x \in I_\Sigma$ *to be executed.*

Theorem 8.2.3 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current configuration, then predicate* $Finished(\lambda, p, i)$ *is implemented by predicate* $\mathtt{Finished}(\vartheta_i, p)$.

Proof We prove this theorem by *reductio ad absurdum*, too. First, assume that predicate $\mathit{Finished}(\lambda, p, i)$ holds, whereas $\texttt{Finished}(\vartheta_i, p)$ does not hold, *i.e.*, $p \notin \vartheta_i$. According to Definition §6.2.5, if $\mathit{Finished}(\lambda, p, i)$ holds then p offered to participate in an empty set of interactions at a previous configuration C_j ($j < i$). Thus, the Termination Rule was applied, and it adds p to ϑ_{j+1} by means of function `AddFinished`. No other rule changes the value of the finished set. Therefore, it is impossible for p not to belong to set ϑ_i, which contradicts our hypothesis since $\mathit{Finished}(\lambda, p, i)$ should not hold.

Now, assume that $\mathit{Finished}(\lambda, p, i)$ does not hold, but $\texttt{Finished}(\vartheta_i, p)$ holds, *i.e.*, $p \in \vartheta_i$. This implies that process p has never produced an event of the form $p.\emptyset$ at any of the configurations before C_i. Then, the framework cannot have detected such an event, and p cannot belong to set ϑ_i since $\vartheta_0 = \emptyset$ and no rule other than the Termination Rule can add a participant to the finished set. □

Theorem 8.2.4 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current configuration, then predicate* $\mathit{Working}(\lambda, p, i)$ *is implemented by predicate* $\texttt{Working}(\varphi_i, \vartheta_i, p)$.

Proof According to Definition §6.2.5, $\mathit{Working}(\lambda, p, i)$ holds iff participant p is not finished and it is not waiting for an interaction to be executed at configuration C_i. The correctness of the implementation follows immediately from Theorems §8.2.1 and §8.2.3 since the definition of `Working` depends solely on `Waiting` and `Finished`. □

Theorem 8.2.5 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current configuration, then predicate* $\mathit{Enabled}(\lambda, x, i)$ *is implemented by predicate* $\texttt{Enabled}(\varphi_i, x)$.

Proof We prove this theorem by *reductio ad absurdum*, too. First, assume that predicate $\mathit{Enabled}(\lambda, x, i)$ holds, whereas $\texttt{Enabled}(\varphi_i, x)$ does not hold, *i.e.*, $\varphi_i(x) \neq \mathbb{P}(x)$. According to Definition §6.2.6, if $\mathit{Enabled}(\lambda, x, i)$ holds, then every participant of x is waiting for it to be executed, *i.e.*, every participant has offered to participate in x previously, but none of them has executed any interactions since then. Every time a participant produces an event of the form $p.\chi$ with $x \in \chi$, the Offer Rule is applied and it adds p to the offer map of every interaction in χ. Thus, if $\mathit{Enabled}(\lambda, x, i)$ holds, this means that every participant has been added previously to the offer map, which is a contradiction since then $p \in \varphi_i(x)$ should hold for every $p \in \mathbb{P}(x)$.

Now, assume that predicate $\mathit{Enabled}(\lambda, x, i)$ does not hold, whereas predicate $\texttt{Enabled}(\varphi_i, p)$ holds, *i.e.*, $\varphi_i(x) = \mathbb{P}(x)$. Note that this implies that there

is at least a participant p of interaction x that has not offered to participate in this interaction. Thus, if it has not produced an event of the form $p.\chi$ with $x \in \chi$, it is impossible for p to belong to $\varphi_i(x)$, which is a contradiction since this would imply that $\varphi_i(x) \neq \mathbb{P}(x)$. □

Theorem 8.2.6 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current configuration, then predicate* $\mathtt{Disabled}(\lambda, x, i)$ *is implemented by predicate* $\mathtt{Disabled}(\varphi_i, x)$.

Proof The proof follows from Theorem §8.2.5 since the definition of `Disabled` relies on `Enabled` only. □

Theorem 8.2.7 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current configuration, then predicate* $\mathtt{Stable}(\lambda, x, i)$ *is implemented by predicate* $\mathtt{Stable}(\varphi_i, \vartheta_i, x)$.

Proof The proof follows from Theorems §8.2.1 and §8.2.3 since the definition of $\mathtt{Stable}$ depends solely on `Waiting` and `Finished`. □

Theorem 8.2.8 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current configuration,* $\mathtt{StaLinked}(x)$ *is then an implementation of* $\mathtt{StaLinked}(x)$.

Proof The proof follows from the fact that `StaLinked` depends on the set of interactions and their participants only, which are constants. □

Theorem 8.2.9 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current configuration,* $\mathtt{DynLinked}(\varphi_i, x)$ *is then an implementation of* $\mathtt{DynLinked}(\lambda, x)$.

Proof Corollary §8.2.2 proves that $\varphi_i(x)$ denotes the set of participants that are waiting for interaction x to be executed; thus, $\varphi_i(x) \cap \varphi_i(y)$ denotes the set of participants that are waiting for either x or y to be executed at the i-th configuration. Therefore, $\{y \in \mathtt{StaLinked}(x) \mid \varphi_i(y) \cap \varphi_i(x) \neq \emptyset\}$ denotes the whole set of interactions that share a participant with x at run time. □

8.2.5 Preservation of safety properties

In this section, we prove that the transition rules by means of which we describe the semantics of our implementation preserve the safety properties in Section §3.2 if $\longrightarrow\!\!\!\!\!\!\rightarrow_L$ preserves them, which is absolutely necessary since, otherwise, the semantics of the underlying interaction language would be incorrect.

Theorem 8.2.10 $\mathtt{SafeOffers}(\lambda|_{\mathrm{IP}_{\mathrm{CORE}}})$ *holds for every execution* λ *generated by the implementation of our framework.*

Proof According to Definition §6.2.8, the SafeOffers safety property requires every participant that offers to participate in a subset of interactions $\chi \subseteq I_\Sigma$ to remain blocked until one of them is selected for execution. Assume that $\lambda|_{\mathrm{IP}_{\mathrm{CORE}}}$ does not satisfy this requirement. This implies that there exist two configuration $D_i = (C_i, \varphi_i, \vartheta_i)$ and $D_j = (C_j, \varphi_j, \vartheta_j)$ such that

$$\mathtt{Working}(\varphi_{i-1}, \vartheta_{i-1}, p) \wedge \exists \Upsilon \subseteq I_\Sigma \cdot \mathtt{Waiting}(\varphi_i, p, \Upsilon) \wedge \neg\, \mathtt{Waiting}(\varphi_j, p, \Upsilon)$$

and there is not any configuration $D_k = (C_k, \varphi_k, \vartheta_k)$ such that

$$k \in (i, j) \wedge C_k \xrightarrow{x}_L C_{k+1} \wedge x \in \Upsilon$$

Note that $\mathtt{Working}(\varphi_{i-1}, \vartheta_{i-1}, p) \wedge \exists \Upsilon \subseteq I_\Sigma \cdot \mathtt{Waiting}(\varphi_i, p, \chi)$ implies that participant p offers to participate in a subset of interactions $\chi \supseteq \Upsilon$ at the i-th configuration. Thus, the Offer Rule adds p to $\varphi_i(x)$ for all $x \in \chi$, and the only rule that can remove p from the offer map is the Synchronisation Rule. Therefore, if $\neg\, \mathtt{Waiting}(\varphi_j, p, \Upsilon)$ holds, it implies that there exist a configuration $k \in (i, j)$ at which the Synchronisation Rule is applied, which contradicts our hypothesis. □

Theorem 8.2.11 $\mathtt{SafeTerms}(\lambda|_{\mathrm{IP}_{\mathrm{CORE}}})$ *holds for every execution* λ *generated by the implementation of our framework.*

Proof According to Definition §6.2.9, the SafeTerms safety property requires every participant that finishes not to be able to offer to participate in any interaction again, and the interactions in which it can participate never to become enabled again.

Note that when participant p finishes, it produces an event of the form $p.\emptyset$ at the i-th configuration, and the Termination Rule can then be applied and it adds p to the finished set. No other rule can change this set. Since we assume that $\xrightarrow{}_L$ satisfies this property, this implies that no other event of the form $p.\chi$ or x for any $x \in \mathbb{I}(p)$ may be detected by the framework and, thus, $p \notin \varphi_j(x)$ for any $j > i, x \in I_\Sigma$. □

Theorem 8.2.12 $\mathtt{SafeSyncs}(\lambda|_{\mathrm{IP}_{\mathrm{CORE}}})$ *holds for every execution* λ *generated by the implementation of our framework.*

Proof According to Definition §6.2.9, the SafeSyncs safety property requires that an interaction cannot be executed unless it is enabled, and, after it is executed, all of the interactions that are linked to it (including itself) must become disabled and the processes that participate in it must begin working again. Let x be the interaction executed at the i-th configuration. Since we assume that $\longrightarrow_L$ satisfies this property, the check for enablement in the Synchronisation Rule is redundant, but guarantees that no interaction can be executed in the framework unless it is enabled. Note that the consequent of this rule guarantees that all of the participants of x are removed from φ_i, *i.e.*, $\texttt{Enabled}(\varphi_{i+1}, y)$ cannot hold for any $y \in \mathbb{I}(p)$, $p \in \mathbb{P}(x)$. Therefore, no interaction with which x shares a participant can be enabled. Furthermore, since no participant of x belongs to $\varphi_{i+1}(y)$ for any $y \in I_\Sigma$, $\texttt{Working}(\varphi_{i+1}, \vartheta_{i+1}, p)$ must hold for any $p \in \mathbb{P}(x)$. □

8.2.6 Fairness requirements for the implementation

The following theorem proves that our implementation does not require the underlying run-time system to implement any fairness notion, which implies that the decision on which rule must be applied in case of conflicts may be completely arbitrary.

Theorem 8.2.13 $\longrightarrow_{FW}$ *does not require any fairness assumption to be implemented by the underlying run-time system.*

Proof On the one hand, the Offer and the Termination Rules may not be applicable continuously since the number of participants that can finish or offer to participate in an interaction is finite; thus, after a finite number of applications, the Synchronisation Rule must necessarily be applicable or the system deadlocks. On the other hand, the Synchronisation Rule cannot be applied continuously since a new interaction cannot become enabled if its participants do not have an opportunity to offer to participate in it.

These are the reasons why the implementation of our theoretical framework does not need to rely on a fair criterion to select amongst conflicting rules, which concludes the proof. □

8.3 Implementing k-conspiracy-free fairness

In this section, we report on how to implement our notion by means of the implementation of our theoretical framework. We first present the configura-

tions on which the algorithm works, and then describe it by means of Plotkin's transition rules. Later, we prove that the implementation is correct.

8.3.1 Configurations

The configurations of our algorithm are three-tuples of the form (D, τ, ρ), namely: $D = (C, \varphi, \vartheta)$ denotes a framework configuration; τ is a queue in which we arrange the interactions according to their age, *i.e.*, $\tau\colon \mathbb{N} \to I_\Sigma$, $\operatorname{ran} \varphi = I_\Sigma$; and ρ is a map that counts the number of times each interaction is executed in a potentially conspiratorial situation, *i.e.*, $\rho\colon I_\Sigma \to \mathbb{N}$, $\operatorname{dom} \rho = I_\Sigma$.

It is worth mentioning that these configurations do not allow us to know how old an interaction is, but how relatively old it is with respect to the others. Note that an implementation in which the age of an interaction is recorded explicitly must be incomplete if the range of the counters is not wide enough to store the age of an interaction that is not executed frequently. Our configurations allow us to determine if an interaction is older than other by examining its position in queue τ because the semantics of our algorithm guarantee that interaction x is older than interaction y as long as $\tau^{-1}(x) < \tau^{-1}(y)$.

The configurations are updated and consulted by means of the following functions and predicates:

$$
\begin{aligned}
\texttt{MoveRear}([x_1, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots x_n], x_j) &= [x_1, \ldots, x_{j-1}, x_{j+1}, \ldots x_n, x_j] \\
\texttt{IncreaseConsp}(\rho, x) &= \{z \mapsto \rho(z) \mid z \in I_\Sigma \setminus \{x\}\} \cup \{x \mapsto \rho(x) + 1\} \\
\texttt{OldestEna}(\tau, \varphi) = x &\iff \texttt{Enabled}(\varphi, x) \Rightarrow \forall y \in I_\Sigma \setminus \{x\} \cdot \tau^{-1}(y) > \tau^{-1}(x) \\
\texttt{PotConsp}(\varphi, \vartheta, x) &\iff \exists y \in \texttt{StaLinked}(x) \cdot \neg\, \texttt{Stable}(\varphi, \vartheta, y)
\end{aligned}
$$

$\texttt{MoveRear}(\tau, x)$ is used every time interaction x is executed, and moves it to the rear of τ, but preserves the relative order of the remaining interactions. $\texttt{IncreaseConsp}(\rho, x)$ increases the potential conspiracy counter of interaction x when it is selected and there is an unstable interaction linked to it. $\texttt{OldestEna}(\tau, \varphi)$ returns the first enabled interaction in τ, *i.e.*, the one that is enabled whose age is minimum. $\texttt{PotConsp}(\varphi, \vartheta, x)$ holds if interaction x is executed in a potentially conspiratorial situation, which happens if there is at least a statically linked interaction that is not stable.

8.3.2 Semantics

The goal of our algorithm is to monitor the transitions of the framework to disallow the execution of an interaction if it does not satisfy the requirements of k-conspiracy-free fairness. The initial configuration is of the form

(D_0, τ_0, ρ_0), where D_0 is the initial framework configuration, $\operatorname{ran}\tau_0 = I_\Sigma$ (note that any initial permutation of I_Σ is valid), and $\rho(x) = 0$ for all $x \in I_\Sigma$. From this configuration onwards, the system is controlled by the Recording, the Conspiratorial Synchronisation, and the Stabilising Synchronisation Rules presented below:

i. The *Recording Rule* describes how the algorithm reacts on occurrence of an event of the form $p.\chi$. In such cases, we only need to record the changes the framework produces on its configuration. Formally,

$$\frac{D_i \xrightarrow{p.\chi}_{FW} D_{i+1}}{(D_i, \tau_i, \rho_i) \xrightarrow{p.\chi}_{CFF_k} (D_{i+1}, \tau_{i+1}, \rho_{i+1})}$$

where $\tau_{i+1} \triangleq \tau_i$
$\rho_{i+1} \triangleq \rho_i$

Note that both the cases $\chi \neq \emptyset$ and $\chi = \emptyset$ are dealt homogeneously since it is not necessary to change neither the queue nor the counters of potentially conspiratorial situations.

ii. The *Conspiratorial Synchronisation Rule* describes how the algorithm reacts when an interaction may be executed in a potentially conspiratorial situation, but it has not executed more than k times in such situations. In these cases, the algorithm does not disallow the interaction from executing, but records this fact.

$$\frac{D_i \xrightarrow{x}_{FW} D_{i+1}}{(D_i, \tau_i, \rho_i) \xrightarrow{x}_{CFF_k} (D_{i+1}, \tau_{i+1}, \rho_{i+1})} \text{ if } \texttt{PotConsp}(\varphi_i, \vartheta_i, x) \wedge \rho_i(x) < k$$

where $D_i \triangleq (C_i, \varphi_i, \vartheta_i)$
$\tau_{i+1} \triangleq \texttt{MoveRear}(\tau_i, x)$
$\rho_{i+1} \triangleq \texttt{IncreaseConsp}(\rho_i, x)$

Note that the interaction that is executed is moved to the rear of τ_i, which transforms it into the younger, and its counter of potentially conspiratorial situations is increased.

iii. The *Stabilising Synchronisation Rule* describes the behaviour of our algorithm when an enabled interaction is ready to be executed in a situation in which no conspiracy may happen, and it is the oldest amongst the set of enabled interactions. In such cases, it can be executed immediately.

$$\frac{D_i \xrightarrow{x}_{FW} D_{i+1}}{(D_i, \tau_i, \rho_i) \xrightarrow{x}_{CFF_k} (D_{i+1}, \tau_{i+1}, \rho_{i+1})} \quad \text{if } \neg \texttt{PotConsp}(\varphi_i, \vartheta_i, x) \wedge x = \texttt{OldestEna}(\tau_i, \varphi_i)$$

where $D_i \triangleq (C_i, \varphi_i, \vartheta_i)$
$\tau_{i+1} \triangleq \texttt{MoveRear}(\tau_i, x)$
$\rho_{i+1} \triangleq \rho_i$

Note that x is moved to the rear of τ_i, too, but no change to ρ_i is made since the interaction has executed in a situation in which the whole set of interactions that are linked to it are stable.

8.3.3 Correctness of predicates and functions

In this section, our goal is to prove that `OldestEna` and `PotConsp` implement $\mathit{OldestEna}$ and $\mathit{PotConsp}$ at the current configuration. We also prove that $\rho_i(x)$ is an implementation of $\sum_{j=0}^{i} \mathit{PotConsp}(\lambda, x, j)$ for all $x \in I_\Sigma$.

Theorem 8.3.1 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current framework configuration and* $E_i = (D_i, \tau_i, \rho_i)$ *is the current configuration of our algorithm,* $\mathit{OldestEna}(\lambda, x, i)$ *is then implemented by* $\texttt{OldestEna}(\tau_i, \varphi_i)$.

Proof According to the definition in Section §7.2, $\mathit{OldestEna}(\lambda, x, i)$ holds iff interaction x is the oldest amongst the interactions that are enabled at configuration C_i. Recall that interaction x is older than interaction y iff it was executed before or, otherwise, the initial arbitrary total order $\prec$ on which the definition of Age relies makes it older than y by definition.

Note that every time the Conspiratorial or the Stabilising Synchronisation Rules are applied, the interaction selected is moved to the rear of queue τ, whereas the relative order of the remaining interactions is maintained. This implies that the closer an interaction is to the rear, the less time has elapsed since it was executed for the last time. Furthermore, $\prec$ is an arbitrary total order, which implies that any initial τ_0 is valid and implements this arbitrary order.

Therefore, `OldestEna` cannot return an interaction other than the oldest amongst the interactions that are enabled at configuration D_i. □

Theorem 8.3.2 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current framework configuration and* $E_i = (D_i, \tau_i, \rho_i)$ *is the current configuration of our algorithm,* $\mathtt{PotConsp}(\lambda, x, i)$ *is then implemented by* $\mathtt{PotConsp}(\varphi_i, \vartheta_i, x)$.

Proof The proof follows from Theorems §8.2.8 and §8.2.7, where we proved that both the implementation of `StaLinked` and `Stable` are correct. □

Theorem 8.3.3 *If* $D_i = (C_i, \varphi_i, \vartheta_i)$ *is the current framework configuration and* $E_i = (D_i, \tau_i, \rho_i)$ *is the current configuration of our algorithm,* $\rho_i(x)$ *is then an implementation of* $\sum_{j=0}^{i} \mathtt{PotConsp}(\lambda, x, j)$ *at the current configuration.*

Proof Initially, $\rho_0(x) = 0$ for every $x \in I_\Sigma$. From the initial configuration onwards, the only rule that can change the value of this map is the Conspiratorial Synchronisation Rule, and it increases it by one every time an interaction is executed in a potentially conspiratorial situation. Since this is the only rule that requires PotConsp to hold and Theorem §8.3.2 proves that this predicate is implemented by `PotConsp`, it follows that $\rho_i(x)$ records the number of times interaction x has been executed in a potentially conspiratorial situation. □

8.3.4 Correctness and completeness of the algorithm

In this section, we prove that the set of executions our algorithm produces coincides with the set of k-conspiracy-free fair executions. As was the case before, if λ is one of the executions our algorithm produces, then $\lambda|_{\mathrm{IP}_{\mathrm{CORE}}}$ denotes its projection on $\mathrm{IP}_{\mathrm{CORE}}$.

Theorem 8.3.4 (Correctness) *Let* λ *be any of the executions our algorithm can produce, then* $\lambda|_{\mathrm{IP}_{\mathrm{CORE}}}$ *is* k*-conspiracy-free fair.*

Proof We proceed by *reductio ad absurdum*. Assume that there exists an index $p^* \geq 0$ such that our algorithm produces a prefix of an execution of the form

$$\begin{aligned}
&\Omega = E_0 \xrightarrow{e_1}_{\mathrm{CFF}_k} E_1 \xrightarrow{e_2}_{\mathrm{CFF}_k} \cdots \xrightarrow{e_{p^*}}_{\mathrm{CFF}_k} E_{p^*} \xrightarrow{e_{p^*+1}}_{\mathrm{CFF}_k} E_{p^*+1} \\
&E_i = (D_i, \tau_i, \rho_i) \text{ for all } i \in [0, p^* + 1] \\
&D_i = (C_i, \varphi_i, \vartheta_i) \text{ for all } i \in [0, p^* + 1]
\end{aligned}$$

Without loss of generality, we can assume that this prefix is minimum, *i.e.*, E_{p^*} is the first configuration at which it can be detected that the execution is not k-conspiracy-free fair since the execution of event e_{p^*+1} would violate the criterion. This implies that $e_{p^*+1} = x$ for some $x \in I_\Sigma$, which implies that we can distinguish between the following cases:

i. The potential conspiracy threshold can be exceeded. This cannot happen because the Conspiratorial Synchronisation Rule would not be applicable since the implementations of $\mathtt{PotConsp}$ and $\sum_{j=0}^{i} \mathtt{PotConsp}(\lambda, x, j)$ are correct according to Theorems §8.3.2 and §8.3.3.

ii. The interactions linked to it are stable, but it is not the oldest. This is impossible since the implementations of $\mathtt{OldestEna}$ and $\mathtt{PotConsp}$ are correct according to Theorems §8.3.1 and §8.3.2.

These contradictions conclude the proof because they imply that neither the Conspiratorial Synchronisation Rule nor the Stabilising Synchronisation Rule can allow an interaction to be executed unless the new prefix generated satisfies the requirements of k-conspiracy-free fairness. □

Theorem 8.3.5 (Completeness) *Let* λ *be any* k*-conspiracy-free fair execution for some* $k \in \mathbb{N}$*. Our algorithm can produce an execution* Ω *such that* $\Omega|_{IP_{CORE}} = \lambda$*.*

Proof We proceed by *reductio ad absurdum*. Assume that there exists a k-conspiracy-free fair execution λ that cannot be generated by our algorithm. This implies that there exists an index $p^* \geq 0$ such that

$$\lambda = C_0 \xrightarrow{e_1}_L C_1 \xrightarrow{e_2}_L \dots \xrightarrow{e_{p^*}}_L C_{p^*} \xrightarrow{e_{p^*+1}}_L C_{p^*+1}$$

but the algorithm produces

$$\begin{aligned}
&\Omega = E_0 \xrightarrow{e_1}_{CFF_k} E_1 \xrightarrow{e_2}_{CFF_k} \dots \xrightarrow{e_{p^*}}_{CFF_k} E_{p^*} \\
&E_i = (D_i, \tau_i, \rho_i) \text{ for all } i \in [0, p^*] \\
&D_i = (C_i, \varphi_i, \vartheta_i) \text{ for all } i \in [0, p^*]
\end{aligned}$$

and it paints itself into a corner at configuration E_{p^*} or the transition to the following configuration is of the form $E_{p^*} \xrightarrow{e'_{p^*+1}}_{CFF_k} E'_{p^*+1}$, but $e'_{p^*+1} \neq e_{p^*+1}$ or E'_{p^*+1} is of the form $((C'_{p^*+1}, \varphi'_{p^*+1}, \vartheta'_{p^*+1}), \tau'_{p^*+1}, \rho'_{p^*+1})$ and $C'_{p^*+1} \neq C_{p^*+1}$. We prove that this leads to contradictions by structural induction on the event e_{p^*+1}, namely:

i. If $e_{p^*+1} = p.\chi$ for some $p \in P_\Sigma$, $\chi \subseteq I_\Sigma$, then the Recording Rule should have been applied at configuration E_{p^*}, and E_{p^*+1} would be of the form $(D_{p^*+1}, \tau_{p^*+1}, \rho_{p^*+1})$ with $D_{p^*+1} = (C_{p^*+1}, \varphi_{p^*+1}, \vartheta_{p^*+1})$.

ii. If $e_{p^*+1} = x$ for some $x \in I_\Sigma$, then we can distinguish a couple of cases, namely: if C_{p^*} satisfies $\neg\, \mathtt{PotConsp}(\lambda, x, i) \wedge \mathtt{OldestEna}(\lambda, x, i)$, then the Conspiratorial Synchronisation Rule would have been applied and E'_{p^*+1} would be $((C_{p^*+1}, \varphi_{p^*+1}, \vartheta_{p^*+1}), \mathtt{MoveRear}(\tau_{p^*}, x), \rho_{p^*})$ since the implementations `PotConsp` and `OldestEna` are correct according to Theorems §8.3.1 and §8.3.2. If $\sum_{j=0}^{i} \mathtt{PotConsp}(\lambda, x, j) < k$ holds, then Theorem §8.3.3 proves that this predicate is implemented by $\rho_i(x) < k$ at E_i, which implies that E_{p^*+1} would be of the form $(D_{p^*+1}, \mathtt{MoveRear}(\tau_{p^*}, x), \mathtt{IncreaseConsp}(\rho_{p^*}, x))$ with $D_{p^*+1} = (C_{p^*+1}, \varphi_{p^*+1}, \vartheta_{p^*+1})$.

In both cases, there is a clear contradiction since the algorithm would react on the same event and would lead to the same configuration. □

8.4 Summary

In this chapter, we have proven that both our theoretical framework and our notion can be implemented by means of a finite-state scheduler, which is a marginal, but important conclusion.

Partial results were published at the PROLE'02 Conference [123], the Euro-Par'02 Conference [122], and Concurrency and Computation: Practice & Experience [125]. The details of our J♯ implementation were published as a technical report in Reference [118].

Chapter 9

Empirical evaluation

Experience teaches slowly
at the cost of mistakes

James A. Froude, 1818–1894
English historian

Most current implementations of fairness notions have not been analysed from an empirical point of view. In this chapter, we present a detailed study in which compare our implementation of k*-conspiracy-free fairness with most current algorithms to implement strong fairness and hyperfairness in the context of multiparty-based systems. In Section §9.1, we introduce the chapter and define some important concepts; in Section §9.2, we show how the selection and the execution times behave as the number of participants or the execution length increases; in Section §9.3, we study how different algorithms deal with conspiratorial situations; in Section §9.4, we present a simple method to compute a good value for* k *in the dining philosophers problem; finally, we summarise the chapter and our main contributions in Section §9.5.*

9.1 Introduction

In order to evaluate the implementation of our theoretical framework and our notion, we measured their performance and effectiveness using the dining philosophers system as a benchmark. We implemented it using the J♯ programming language, which is an efficient JAVA dialect for the .NET platform [130]. We run our tests on a WINDOWS XP PROFESSIONAL machine that was equipped with a 2.0 Ghz AMD ATHLON XP microprocessor, and 512 MB of DDR 266 Mhz RAM memory. During the tests, the machine was disconnected from the network, and WINDOWS services were reduced to the minimum in order to avoid as much exogenous interferences as possible. We did not turn off the .NET garbage collector since its interferences are unavoidable in real life. The details on this implementation may be found in Reference [118].

In this chapter, we compare our proposal (hereafter referred to as $\mathcal{CFF}_k$) with most of the results on implementing fair multiparty-based systems presented in Chapter §4, namely: the strongly fair proposal by Francez and Forman [58] (hereafter referred to as Fair), the incremental strongly fair algorithm by Corchuelo [35] (hereafter referred to as IncFair), the transformational strong hyperfair approach by Attie *et al.* [10] (hereafter referred to as HyperFair), and the randomised proposal by Joung [72] (hereafter referred to as TB). Note that the implementation of our theoretical framework is unfair but it deserves to be compared since it shows how a system runs when no fairness notion is implemented, *i.e.*, it allows to analyse the impact on the efficiency that the aforementioned proposals induce (hereafter we refer to it as Random). The results using Best's, Fagin and Williams's, or Olderog and Apt's algorithms [18, 19, 53, 103] were so similar to the results using Francez and Forman's algorithm that we decided not to show them explicitly.

We run the algorithms in several scenarios to show their impact on conspiracies and the efficiency, which we interpreted as a function of the time elapsed to run an experiment and the average selection time; we also evaluated the impact of k on the behaviour of a program. Note that the transformation by Attie *et al.* cannot be a applied to a system unless it is in interaction-driven normal formal form and every interaction is conspiracy resistant. The results on which we report were produced with Program §9.1. Note that interaction $Lunch_i$ simulates the execution of both interactions Get_i and Rel_i in sequence since otherwise the *Rel* interactions would not be conspiracy-resistant and the transformation could not be applied.

In the rest of this chapter, we use the following terms:

Length of an execution: we define it as the number of *Get* interactions that are executed, and refer to it as L. Note that an actual execution must include

CR-DINNER :: $[\|\|_{i=1}^{N} P_i \parallel F_i]$, **where**
P_i :: *[$\text{Lunch}_i\langle \textit{eat} \rangle \rightarrow \textit{think}$] ;
F_i :: *[$\text{Lunch}_i\langle\rangle \rightarrow$ **skip** [] $\text{Lunch}_{i+1}\langle\rangle \rightarrow$ **skip**] .

Program 9.1: *A conspiracy-resistant version of the dining philosophers problems in interaction-driven normal form.*

synchronisation events on both *Get* and *Rel* interactions, but the latter depend on the former to be executed; thus, they need not be taken into account explicitly. In the case of HyperFair, the length of the execution is defined as half the number of *Lunch* interactions that are executed.

Number of lunches: it represents the number of *Get* interactions each philosopher executes. We denote it as L_i, where i is the index corresponding to the i-th philosopher.

Distribution of lunches: it is a discrete function that relates every execution of a *Get* interaction with its corresponding philosopher. We represent it as a cloud of points in which axis $0X$ represents the execution of *Get* interactions and axis $0Y$ represents their corresponding philosophers.

Execution time: it is the time that elapses since the beginning of an execution until it reaches a predetermined length.

Selection time: it is the average time that elapses since an interaction has only one participant that is interested in synchronising on it until it is selected or the execution finishes, whichever happens before.

We often write "philosopher i has lunch" to mean that it executes interaction $\textit{Get}_i$. Note that lunches are not atomic since they require each philosopher to participate in two different interactions. However, "having lunch" may be seen as atomic since once a philosopher or a fork executes a *Get* interaction, they only action they can execute is the corresponding *Rel* interaction. We also identify "lunches" with the execution of *Get* interactions. Although this is an abuse of the language, we think that these expressions are worth since they are not misunderstanding and make the writing more fluent and intuitive.

9.2 Timings

To evaluate the behaviour of our implementation with respect to the time, we devised a scenario in which the time each philosopher spent at thinking or

eating was negligible with respect to the time needed to detect enablements, get mutual exclusion or select interactions. In this setting, each philosopher should be able to have lunch as much as the others during a long-enough fair execution.

On the one hand, we measured how long the algorithms under consideration took to distribute 5000 lunches when the number of philosophers ranged from 10 to 200 in steps of 10 units; on the other hand, we measured how long they took to execute a program with 10 philosophers when the execution length ranged from 100 to 5000 lunches in steps of 100 units. The first experiment is useful to characterise how the algorithms behave with respect to the number of participants and interactions. As we show in the following section, we obtained a second-grade polynomial behaviour. On the contrary, the second experiment was used to determine how the execution length affects the timings. In this case, our implementation behaves linearly.

In order to reduce the significance of possible outliers produced by occasional interferences with the operating system, we took averaged times in 100 runs and left out the maximum and the minimum timings for each experiment.

9.2.1 Timings as the number of participants increases

Figure §9.1 shows that Random is the fastest algorithm, whereas the slowest is $\mathcal{CFF}_k$ with $k = 0$. Note that IncFair performs better than Fair since, in the average, it need not examine the whole set of interactions before reaching a decision, but worse than the random proposal since the data structures it needs to maintain are more complex and the time needed to reach a decision depends on the probability of finding an enabled interaction and the average time they need to become stable.

The execution time of $\mathcal{CFF}_k$ depends on the value we assign to k. If k is minimum ($k = 0$), it performs worse than Fair because interactions are pairwise linked and this implies that every time an interaction is selected, the remaining need to be stable. This is very similar to Fair, but the data structures $\mathcal{CFF}_k$ needs to maintain are more complex, which justifies the slightly worse performance of $\mathcal{CFF}_k$ in this case. However, if k higher, its performance is similar to Random if the execution is not long enough. Note that Lemmata §7.3.1 and §7.3.2 guarantee that there exists a configuration such that every interaction that is selected from it onwards executes in non-potentially conspiratorial situations. This implies that from that configuration onwards, the behaviour of $\mathcal{CFF}_k$ with $k > 0$ tends to be the behaviour of $\mathcal{CFF}_k$ with $k = 0$.

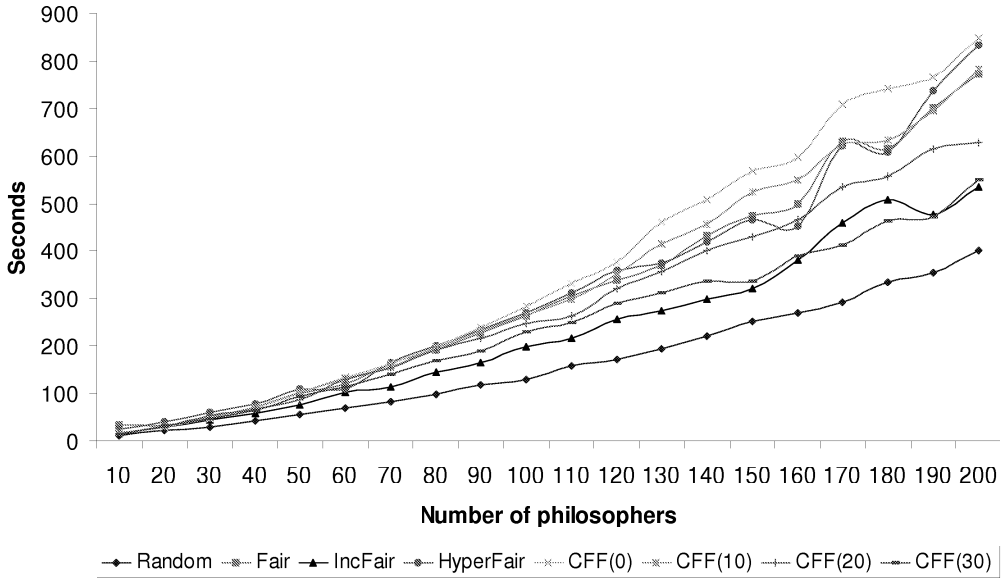


Figure 9.1: *Execution time as the number of participants increases.*

N	Random	Fair	HyperFair	IncFair	$\mathcal{CFF}_k(0)$	$\mathcal{CFF}_k(10)$	$\mathcal{CFF}_k(20)$	$\mathcal{CFF}_k(30)$
10	10.79	33.87	23.61	16.82	18.18	16.46	15.67	14.38
20	21.82	34.20	40.38	28.10	29.34	31.18	29.27	31.22
30	28.31	50.87	60.47	43.92	46.52	49.74	52.50	47.38
40	42.86	72.36	77.81	57.81	69.29	72.08	67.28	65.35
50	55.43	99.87	108.53	76.26	99.91	102.44	86.62	94.39
60	68.74	120.90	108.62	101.53	134.23	131.19	128.24	113.45
70	81.46	161.95	164.56	114.16	161.95	156.16	154.26	140.98
80	97.42	191.88	199.54	144.72	198.02	195.10	190.65	169.78
90	118.46	227.25	234.26	164.57	239.24	231.91	215.91	189.91
100	130.00	263.41	269.43	198.52	283.87	266.10	248.15	228.48
110	157.95	304.81	312.58	216.83	331.69	297.44	262.61	248.73
120	171.99	339.18	359.12	256.53	376.28	351.95	321.46	290.69
130	193.41	370.85	373.64	274.27	460.99	413.58	357.14	311.53
140	221.27	432.38	419.86	299.49	508.19	456.67	400.00	336.62
150	252.16	474.73	465.00	319.70	568.60	524.12	430.04	335.96
160	269.13	499.09	451.20	379.97	596.29	550.34	466.21	390.48
170	292.04	629.60	621.01	459.16	708.32	624.70	535.05	411.42
180	334.91	614.70	607.76	507.15	742.70	632.66	557.76	464.12
190	353.69	702.02	737.38	475.83	766.19	694.71	615.29	471.16
200	401.64	773.87	833.13	533.68	848.39	782.33	629.01	550.10

Table 9.1: *Execution time as the number of participants increases.*

Algorithm	Polynomial	(R^2)	Exponential	(R^2)
Random	$0.66x^2 + 6.02x + 6.67$	0.998	$10.31e^{0.16x}$	0.928
Fair	$1.30x^2 + 11.65x + 10.74$	0.995	$39.63e^{0.16x}$	0.946
IncFair	$0.84x^2 + 10.28x + 4.95$	0.989	$29.35e^{0.16x}$	0.928
HyperFair	$1.50x^2 - 8.00x + 24.00$	0.985	$39.73e^{0.16x}$	0.934
$\mathcal{CFF}_k(0)$	$1.31x^2 + 17.96x - 17.89$	0.996	$32.80e^{0.18x}$	0.920
$\mathcal{CFF}_k(10)$	$1.10x^2 + 17.33x - 10.99$	0.997	$33.79e^{0.17x}$	0.910
$\mathcal{CFF}_k(20)$	$0.71x^2 + 18.94x - 12.79$	0.997	$33.43e^{0.16x}$	0.896
$\mathcal{CFF}_k(30)$	$0.40x^2 + 18.67x - 8.02$	0.994	$34.88e^{0.15x}$	0.891

Table 9.2: *Execution time tendency as the number of participants increases.*

This is the behaviour we expected since k constraints the number of times that linked interactions do not have to wait for each other; thus, the smaller k is, the more they have to wait for each other. If there are 10 philosophers, for instance, the average number of interactions per second ranges from 550.05 int/sec in the worst case ($\mathcal{CFF}_k$ with $k = 0$) to 926.78 int/sec in the best case (Random proposal). On the contrary, if there are 200 philosophers, the performance falls down from 11.78 int/sec to 24.89 int/sec, respectively. (This amount is obtained as the quotient between 10 000 (5000 *Get* interactions plus 5000 *Rel* interactions) and the timings shown in Table §9.1.)

Table §9.1 lists the execution times in Figure §9.1. Note that when the system is composed of 10 philosophers, Random executes the test in 10.79 seconds and $\mathcal{CFF}_k$, with $k = 0$, in 18.18 seconds. Clearly, this time increases with the number of philosophers. We carried out an experimental analysis to determine if this behaviour is linear, logarithmic, polynomial, potential, or exponential, and the results are shown in Table §9.2. According to the standard R^2 value [99], we can conclude that the behaviour of the algorithms is second-grade polynomial in all cases.

Figure §9.2 shows the evolution of TB as the number of participants increases. Note that this time is much greater than the others, and its behaviour does not have a clear trend since it is not linear, polynomial, potential, exponential, or logarithmic. Furthermore, the times are expressed in hours and the execution time has a peak when the number of participants is 110 since TB needs over 14 hours to distribute 5 000 lunches. These results indicate that TB lacks practical interest, although it is the only existing wait-free algorithm that implements strong fairness with probability one.

Figure §9.3 shows the time the algorithms take to select interactions. The selection time of our algorithm depends on k again. If k is minimum, it is

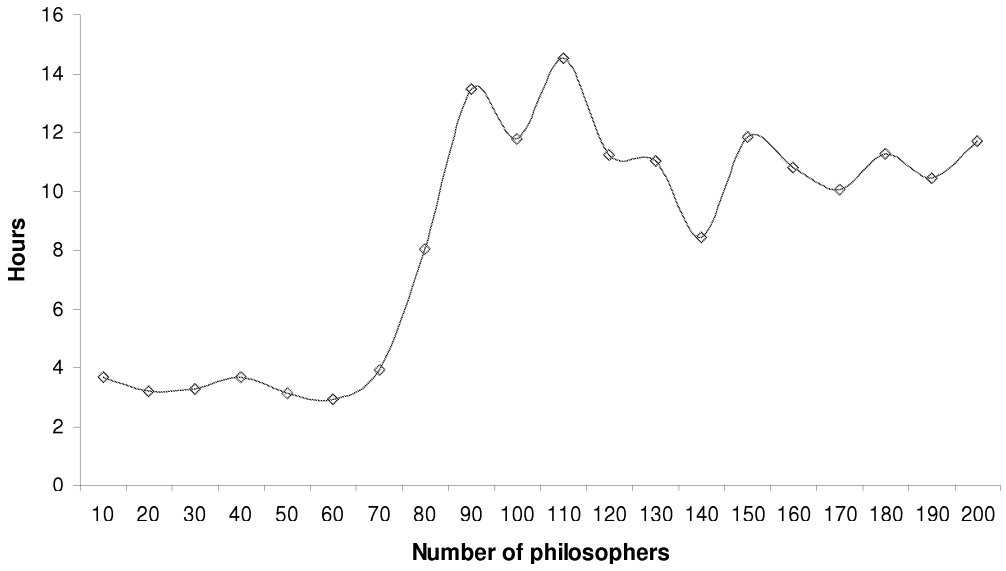


Figure 9.2: *TB's execution time as the number of participants increases.*

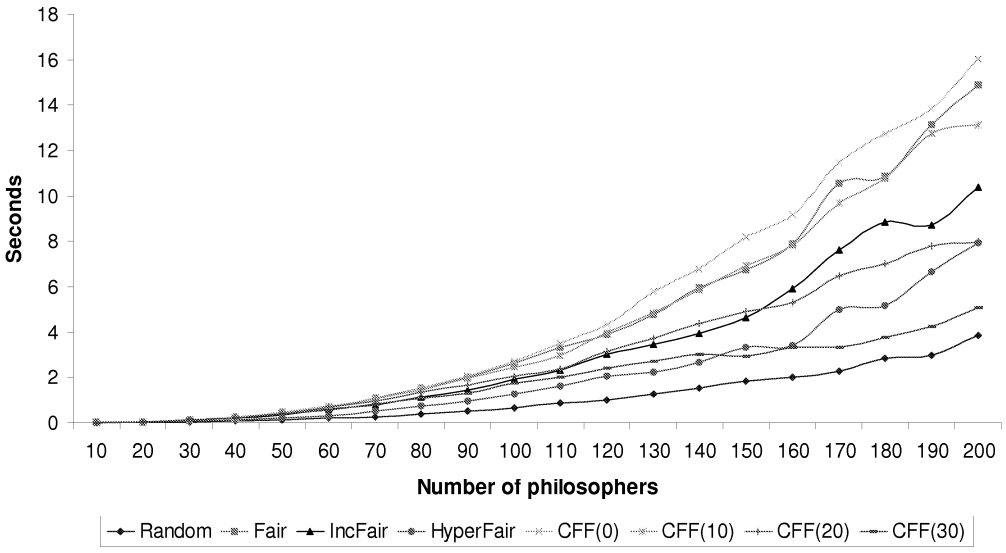


Figure 9.3: *Selection time as the number of participants increases.*

slightly worse than Fair, but it behaves like Random if k is large because almost no interaction reaches the conspiracy threshold and those that are linked to it almost never have to wait for each other. When there are 200 philosophers, the time needed to select interactions in Fair is a bit greater than 14 seconds. Depending on the value of k, the selection time of our algorithm ranges, for instance, from 5 seconds in case $k = 30$ to 16 seconds if $k = 0$. The selection time for TB ranges from 30 seconds to 14 minutes per interaction, but we omit the chart since it does not contribute significantly to this discussion.

9.2.2 Timings as the execution length increases

Figure §9.4 shows the execution times for Random, Fair, IncFair, HyperFair, and $\mathcal{CFF}_k$. The behaviour is quite good since it is linear. TB's times are omitted because they are very large and do not contribute significantly to this discussion.

Note that the value of k in this experiment does not remain constant, and it is expressed as a percentage. The reason is that it is suitable to define the number of potentially conspiratorial situations allowed as a percentage of the total amount of interactions executed. That is, the significance of allowing 10 potentially conspiratorial situations in an execution with 100 lunches is greater than in an execution with 10 000 lunches, *i.e.*, 10% versus 0.1% respectively.

Table §9.3 shows the execution times we obtained. Note that Random has an average execution time of 221 milliseconds for 100 lunches, whereas this figure is about 256 milliseconds if $\mathcal{CFF}_k$ with $k = 0$ is considered. These times increase with the number of lunches. We carried out an analysis to determine the tendency of this behaviour, and it is summarised in Table §9.4. As the figures show, all algorithms behave linearly.

Figure §9.5 shows graphically the selection times for Random, Fair, IncFair, HyperFair, and $\mathcal{CFF}_k$. Note that it remains almost constant when the number of lunches increases since the fluctuations are not significant. Figure §9.6 shows that the only algorithm whose selection time is affected by the execution length is TB. The reason is that the time each participant spends at monitoring other participants is highly dependent on external factors. For instance, a page fault or CPU peak may enlarge the monitoring windows unnecessarily, and this implies that the small interferences that inevitably happen during the execution of a program make the selection time longer as the execution length increases. The author proposed to reset the length of the monitoring windows from time to time, but we are not aware of any published result that describes when and how this can be done safely.

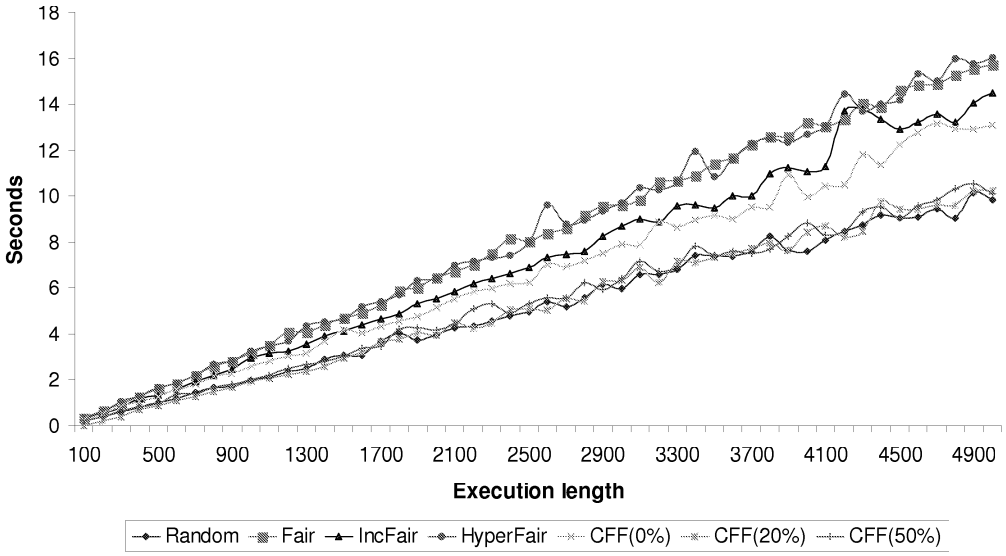


Figure 9.4: *Execution time as the execution length increases.*

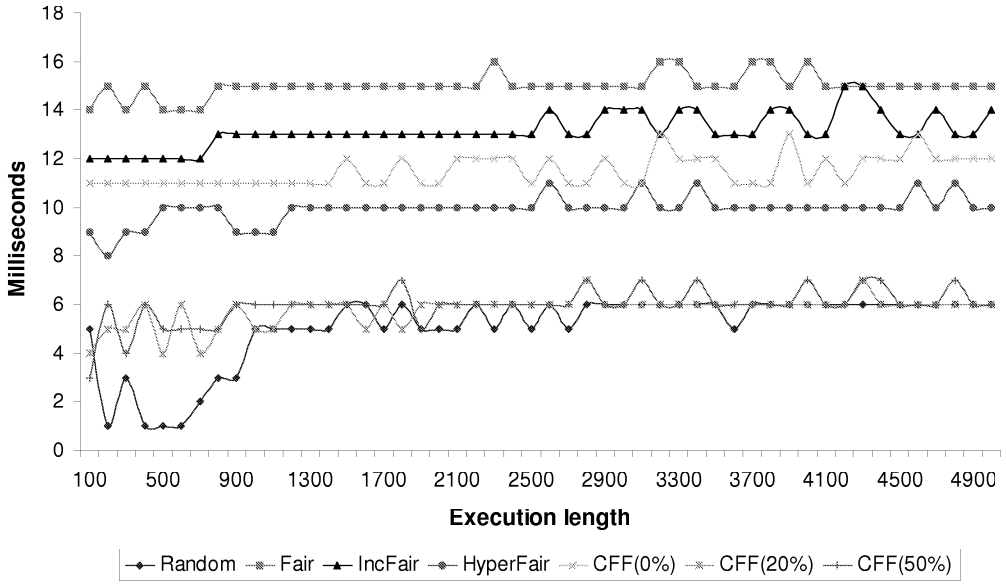


Figure 9.5: *Selection time as the execution length increases.*

N	Random	Fair	HyperFair	IncFair	$\mathcal{CFF}_k(0\%)$	$\mathcal{CFF}_k(20\%)$	$\mathcal{CFF}_k(50\%)$
100	221	336	367	271	256	216	215
200	386	677	601	571	526	415	401
300	611	951	1074	781	826	717	671
400	811	1262	1284	1167	1052	861	842
500	1017	1618	1683	1307	1267	1092	982
600	1167	1853	1840	1598	1588	1252	1347
700	1468	2179	2194	1913	1808	1497	1397
800	1668	2579	2674	2209	2163	1653	1678
900	1708	2794	2809	2444	2279	1923	1803
1000	1963	3160	3261	2925	2574	2068	2003
1100	2113	3521	3463	3180	2830	2244	2203
1200	2359	4072	3696	3250	3050	2384	2509
1300	2499	4072	4327	3556	3165	2599	2699
1400	2895	4412	4522	3911	3696	2950	2800
1500	3065	4682	4665	4137	4162	3220	3020
1600	3080	4902	5191	4382	4037	3651	3360
1700	3676	5253	5379	4657	4347	3786	3460
1800	4046	5885	5694	4877	4577	4046	4232
1900	3741	6030	6325	5308	4763	3957	4257
2000	3951	6450	6408	5544	5123	4492	4182
...	...	...	...	...	...	...	...
5000	9815	15 735	16 023	14 492	13 070	10 501	10 115

Table 9.3: *Execution times as the execution length increases.*

Algorithm	Tendency	R^2
Random	$0.20x + 0.02$	0.994
Fair	$0.31x - 0.02$	0.998
IncFair	$0.29x - 0.20$	0.991
HyperFair	$0.32x - 0.01$	0.995
$\mathcal{CFF}_k(0\%)$	$0.26x - 0.10$	0.994
$\mathcal{CFF}_k(20\%)$	$0.21x - 0.18$	0.994
$\mathcal{CFF}_k(50\%)$	$0.21x + 0.31$	0.993

Table 9.4: *Execution time tendency as the execution length increases.*

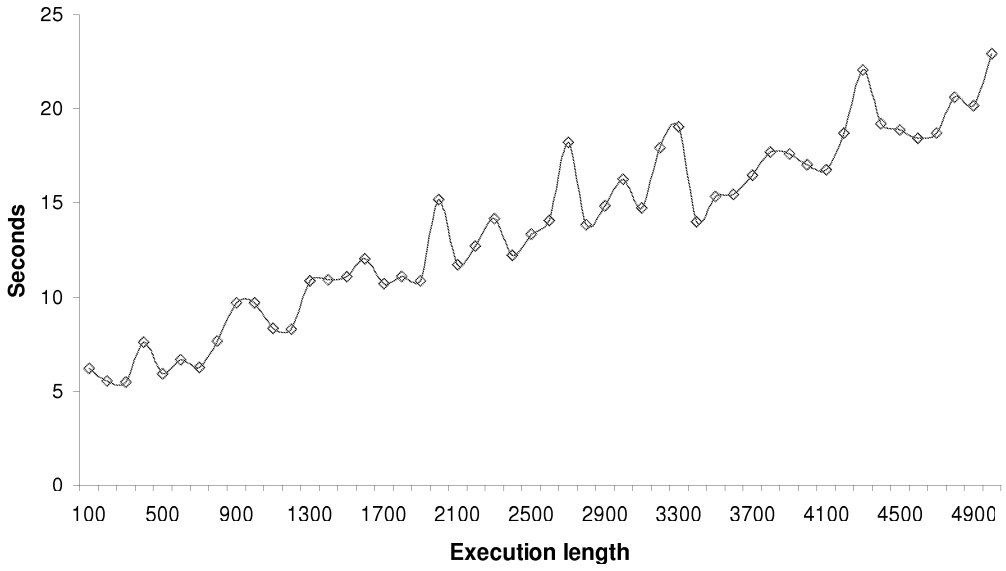


Figure 9.6: *TB's selection time as the execution length increases.*

DINNER :: $[|||_{i=1}^{N} P_i \parallel F_i]$, **where**
P_i :: *[$Get_i\langle\rangle \rightarrow$ *eat*; $Rel_i\langle\rangle$; *think*] ;
F_i :: [i mod 2 = 0 $\rightarrow$ *sleep for one second*];
*[$Get_i\langle\rangle \rightarrow Rel_i\langle\rangle \parallel Get_{i+1}\langle\rangle \rightarrow Rel_{i+1}\langle\rangle$] .

Program 9.2: *A program to force conspiracies.*

9.3 Addressing conspiracies

To see how k-conspiracy-free fairness solves conspiracies, we studied Program §9.2. It is a version of the dining philosophers problem in which the behaviour of the forks was changed so as to introduce a one-second phase difference between even and odd forks, *i.e.*, the odd philosophers always become enabled after the even ones. The timing diagram in Figure §9.7 shows how interactions Get_{i-1} and Get_{i+1} conspire against interaction Get_i because of the initial one-second phase difference ($i \in [1, N]$).

Figure §9.8 shows the histogram of lunches obtained with this experiment and Table §9.5 shows its particular values. Furthermore, Table §9.6 shows the time needed to run the different algorithms. Note that $\mathcal{CFF}_k$ with $k = 0$ solves conspiracies like Fair and the time it takes is slightly greater; on the

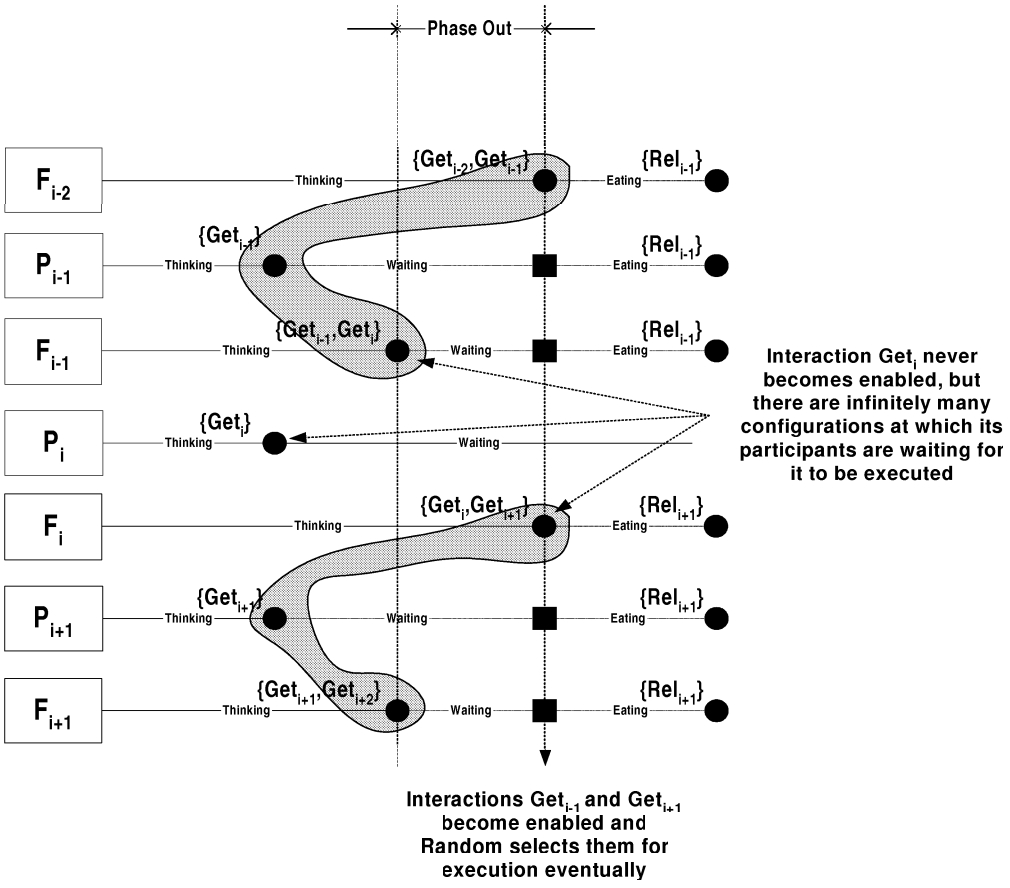


Figure 9.7: *A scenario to force conspiracies.*

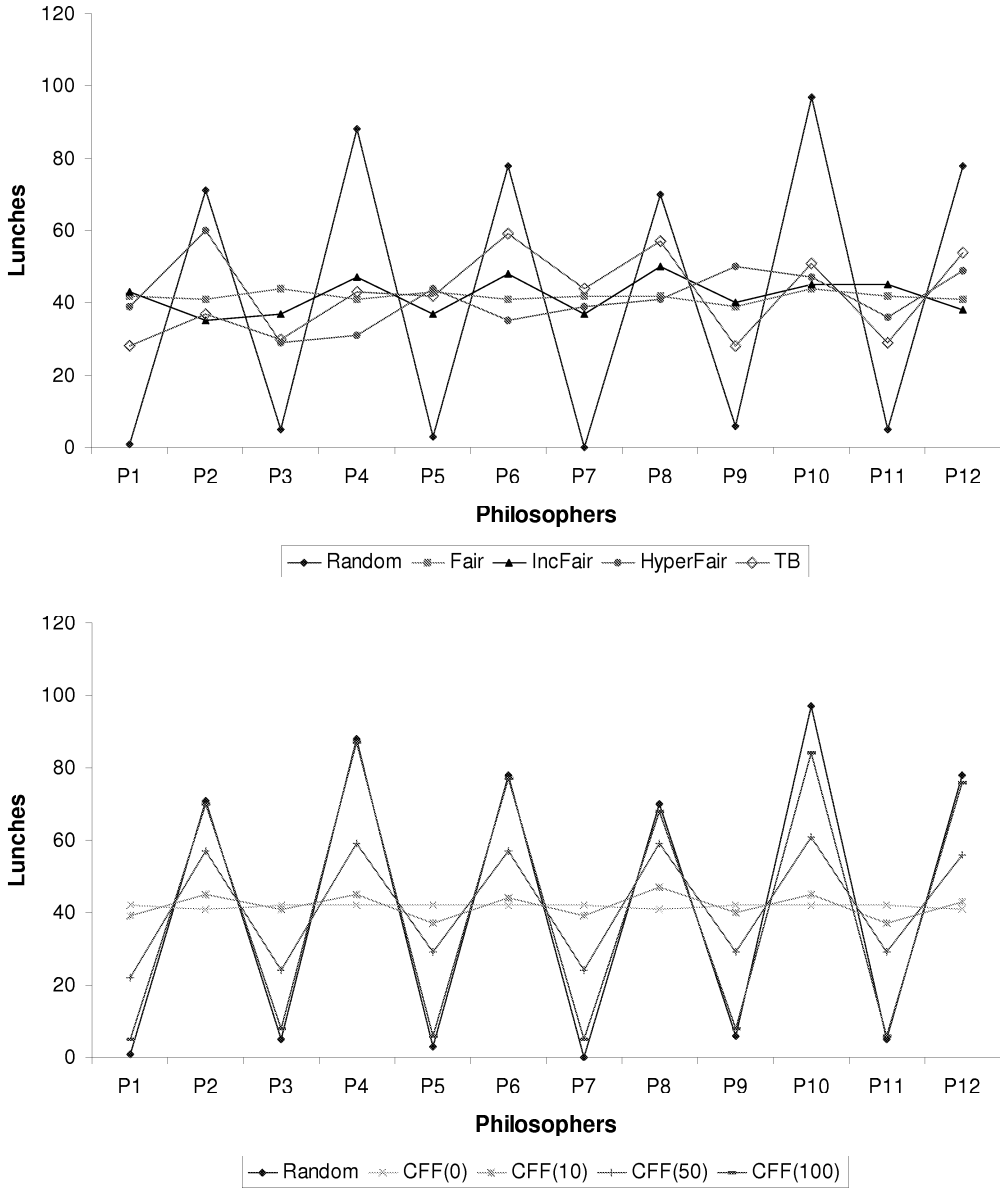


Figure 9.8: *Histograms when conspiracies are forced.*

P_i	Random	Fair	HyperFair	IncFair	TB	$\mathcal{CFF}_k(0)$	$\mathcal{CFF}_k(10)$	$\mathcal{CFF}_k(50)$	$\mathcal{CFF}_k(100)$
P_1	1	42	39	43	28	42	39	22	5
P_2	71	41	60	35	37	41	45	57	70
P_3	5	44	29	37	30	42	41	24	8
P_4	88	41	31	47	43	42	45	59	87
P_5	3	43	44	37	42	42	37	29	6
P_6	78	41	35	48	59	42	44	57	77
P_7	0	42	39	37	44	42	39	24	5
P_8	70	42	41	50	57	41	47	59	68
P_9	6	39	50	40	28	42	40	29	8
P_{10}	97	44	47	45	51	42	45	61	84
P_{11}	5	42	36	45	29	42	37	29	6
P_{12}	78	41	49	38	54	41	43	56	76

Table 9.5: *Histograms when conspiracies are forced.*

Algorithm	Execution time (ms)
Random	2033
Fair	14 108
HyperFair	12 042
IncFair	9855
TB	133 536
$\mathcal{CFF}_k(0)$	15 540
$\mathcal{CFF}_k(10)$	4054
$\mathcal{CFF}_k(50)$	3466
$\mathcal{CFF}_k(100)$	2007

Table 9.6: *Execution time when conspiracies are forced.*

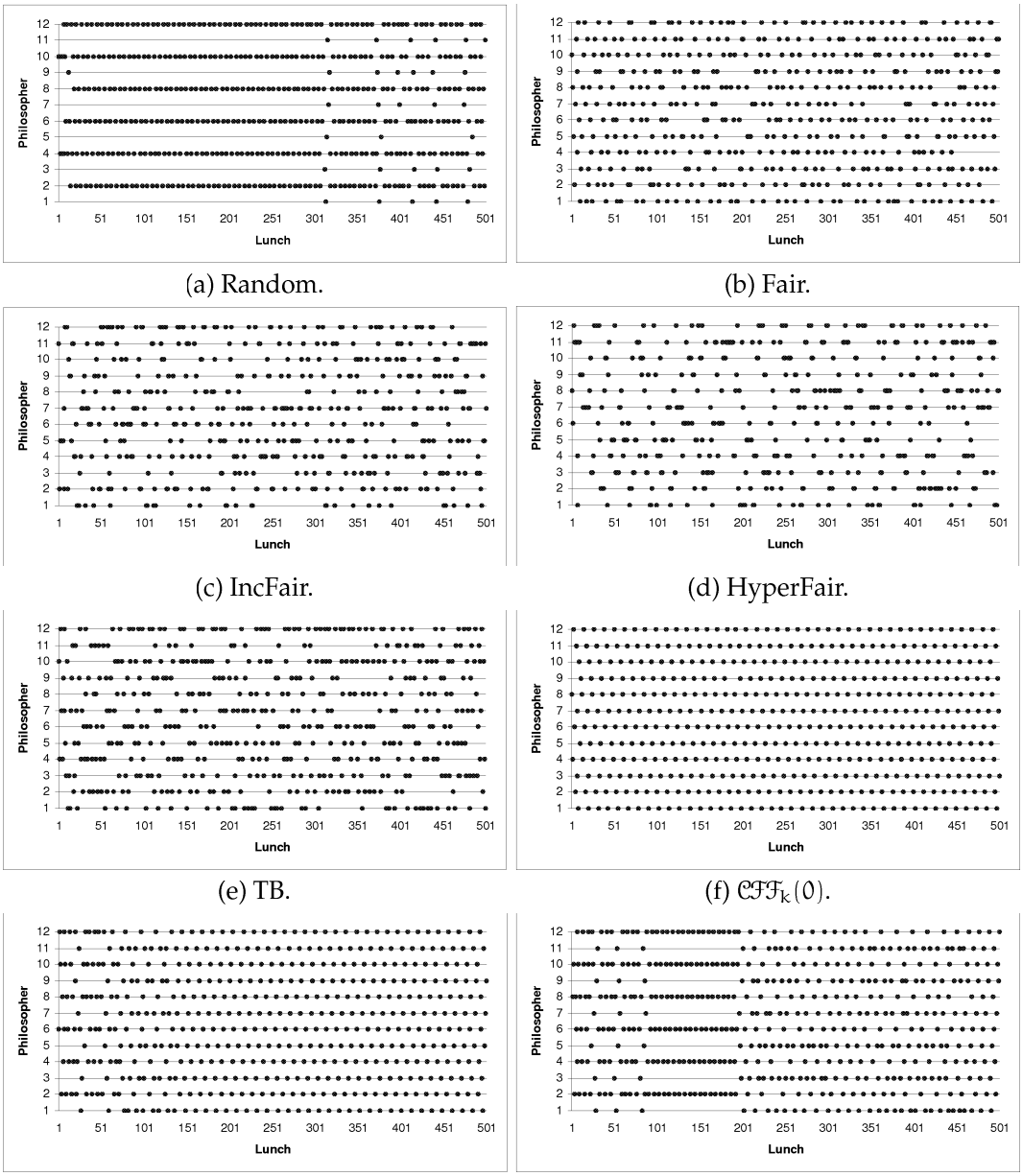
contrary, $\mathcal{CFF}_k$ with $k = 10$ alleviates the negative effects of potentially conspiratorial situations as Fair does (note that the maximum difference between two philosophers is roughly 8 with $\mathcal{CFF}_k$ and $k = 0$, but nearly 5 with Fair), and it improves its execution time from 15 540 to 4054 milliseconds.

However, these figures are not concluding. The charts in Figure §9.9 show the distribution of lunches produced with the algorithms we are evaluating. At a first glance, it might seem that the reason why Fair and IncFair produce a good distribution of lunches is that they require all or many interactions to be stable before one of them can be selected for execution; however, a careful study of the algorithms reveals that this behaviour depends completely on the features of the random number generator used, namely: the range of values it produces and its quality. In the previous experiments, we used Casale's random number generator [37] and configured it to produce numbers in the range $[0, 100]$. When it was configured to produce numbers in the range $[0, 2^{31} - 1]$, we obtained the distribution of lunches in Figures §9.10.(a) and §9.10.(b). These results are clearly very undesirable since they are even worse than the distribution of lunches with the Random algorithm, but more time consuming. The behaviour of TB seems very adequate, but it depends on the quality of the random number generator used. We substituted Casale's generator by the built-in generator .NET provides, and we obtained the results in Figure §9.10.(c). This distribution of lunches is very similar to the Random algorithm, but it is several hundred orders of magnitude more costly.

Contrarily to these schedulers, ours does not depend on a random number generator, and the notion it implements was designed to control conspiracies. If k is zero, no potentially conspiratorial situation is allowed, so the system is controlled so that every philosopher can have lunch as many times as the rest, which was the expected behaviour. As the value of k increases, more interactions can be executed in potentially conspiratorial situations, but they can never exceed the threshold. Thus, changing the value of k may help adjust the notion to the features of the system under consideration.

9.4 Tuning k in the dining philosophers problem

To tune the value of k and analyse its influence on the execution, we study Program §9.3. Now, the time the philosophers spend at eating is negligible with respect to the time they spend at thinking, which is given by function Think. In other words, we assume that the time the resources are used is negligible with respect to the time a process is working before it needs to compete for them. Note that $\frac{1}{\mathtt{Think(i)}}$ is the frequency at which philosopher i request its corresponding forks.



(a) Random.

(b) Fair.

(c) IncFair.

(d) HyperFair.

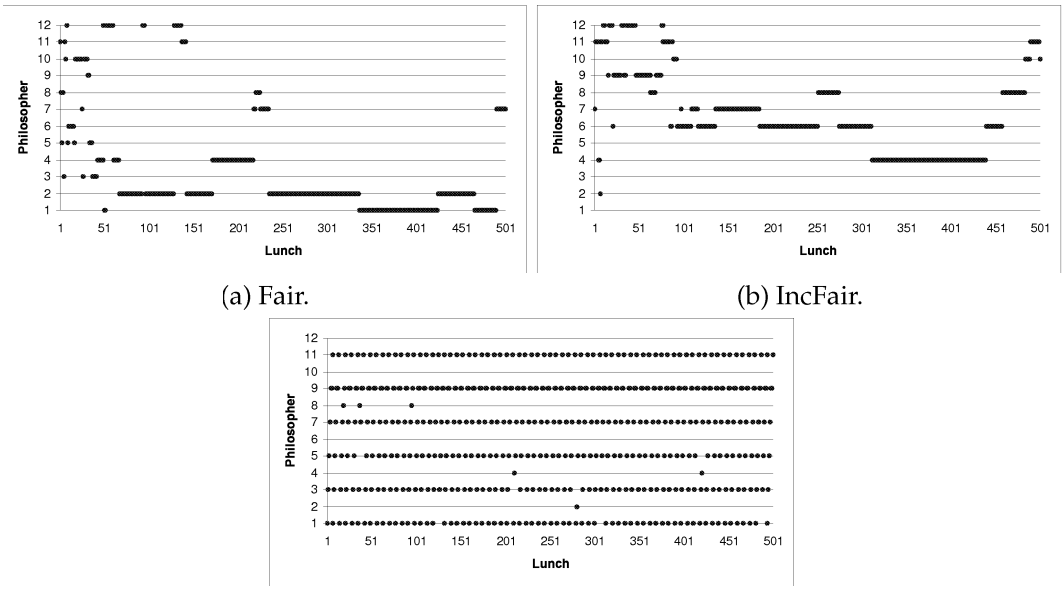
(e) TB.

(f) $\mathcal{CFF}_k(0)$.

(g) $\mathcal{CFF}_k(10)$.

(h) $\mathcal{CFF}_k(30)$.

Figure 9.9: *Distribution of lunches when conspiracies are forced.*



(a) Fair.

(b) IncFair.

(c) TB.

Figure 9.10: *Distribution of lunches using different random number generator.*

DINNER :: $[\|_{i=1}^{N} P_i \parallel F_i]$, **where**
P_i :: *[$Get_i\langle\rangle \rightarrow$ *eat*; $Rel_i\langle\rangle$; *wait for* $\mathtt{Think}(i)$ *units of time*] ;
F_i :: *[$Get_i\langle\rangle \rightarrow Rel_i\langle\rangle$ [] $Get_{i+1}\langle\rangle \rightarrow Rel_{i+1}\langle\rangle$] .

Program 9.3: *A sample program to show how to tune* k.

9.4.1 An analytical characterisation

Let L_i be the number of times philosopher P_i has lunch in an execution, and let L be its length; then, P_i spends $L_i\mathit{Think}(i)$ units of times at thinking, and $\sum_{i=1}^{N} L_i = L$. For instance, if there are three philosophers and $\mathit{Think}(i) = i$, then philosopher P_1 spends L_1 units of time at thinking, P_2 thinks for $2L_1$ units, and philosopher P_3 thinks for $3L_1$ units. Note that with an ideal interleaving and with a sufficiently large execution, we can assume that all of the philosophers spend the same time at working, *i.e.*, $L_1 = 2L_1 = 3L_1$. We can generalise this reasoning for any Think function and for every couple of philosophers so that

$$\forall i, j \in [1, N] \cdot (L_i\mathit{Think}(i) = L_j\mathit{Think}(j))$$

This formula can be re-written as the following equation system:

$$\begin{array}{ccc} L_1\mathit{Think}(1) & = & L_2\mathit{Think}(2) \\ L_1\mathit{Think}(1) & = & L_3\mathit{Think}(3) \\ \vdots & \vdots & \vdots \\ L_1\mathit{Think}(1) & = & L_N\mathit{Think}(N) \end{array}$$

Thus, we can express the times P_i has lunch as a function of the times that P_1 has lunch in order to find variable L_1 ($i \in [1, N]$):

$$L_2 = L_1\frac{\mathit{Think}(1)}{\mathit{Think}(2)}, L_3 = L_1\frac{\mathit{Think}(1)}{\mathit{Think}(3)}, \ldots, L_N = L_1\frac{\mathit{Think}(1)}{\mathit{Think}(N)}$$

Now, we can sum both sides of the equations so that:

$$\sum_{i=2}^{N} L_i = L_1\mathit{Think}(1)\sum_{i=2}^{N}\frac{1}{\mathit{Think}(i)}$$

Since $\sum_{i=2}^{N} L_i = L - L_1$, L_1 can be expressed as follows:

$$L_1 = \frac{L}{1 + \mathit{Think}(1)\sum_{i=2}^{N}\frac{1}{\mathit{Think}(i)}}$$

Degrees of freedom	Confidence level α				
	0.995	0.990	0.975	0.95	0.900
5	0.412	0.554	0.831	1.145	1.610
6	0.676	0.872	1.237	1.635	2.204
7	0.989	1.239	1.690	2.167	2.833
8	1.344	1.647	2.180	2.733	3.490
9	1.735	2.088	2.700	3.325	4.168
10	2.156	2.558	3.247	3.940	4.865
11	2.603	3.053	3.816	4.575	5.578
12	3.074	3.571	4.404	5.226	6.304
13	3.565	4.107	5.009	5.892	7.041
14	4.075	4.660	5.629	6.571	7.790
15	4.601	5.229	6.262	7.261	8.547
20	7.434	8.260	9.591	10.851	12.443
30	13.787	14.953	16.791	18.493	20.599
40	20.707	22.164	24.433	26.509	29.051
50	27.991	29.707	32.357	34.764	37.689
60	35.534	37.485	40.482	43.188	46.459
70	43.275	45.442	48.758	51.739	55.329
80	51.172	53.540	57.153	60.391	64.278
90	59.196	61.754	65.647	69.126	73.291
100	67.328	70.065	74.222	77.929	82.358
150	109.142	112.668	117.985	122.692	128.275
200	152.241	156.432	162.728	168.279	174.835

Table 9.7: *A portion of the standard χ^2 table.*

9.4.2 A simple method to select k

Once we have computed the expected distribution of lunches, we can use a goodness-of-fit method to compute the first value of k that allows to produce an actual distribution that does not diverge too much from the former. There are many such methods in the literature [68, 99], but we have selected the Chi-squared Test because it is simple, yet powerful and general. It can be applied to any system with more than two interactions that execute five times at least, which we think are very weak constraints.

Using this test, the simplest method to calculate the first value of k that allows to adjust the actual execution to the expected one is as follows:

Step 0: Initialise k to 0, and α to a standard confidence level. Let Exp_i be the expected number of times interaction x_i should be executed for all $i \in [1, N]$.

Step 1: Run a set of experiments and let Obs_i be the average number of times interaction x_i executes for all $i \in [1, N]$.

Step 2: Calculate X^2 as follows:

$$X^2 = \sum_{i=1}^{N} \frac{(Obs_i - Exp_i)^2}{Exp_i}$$

Step 3: Compare X^2 with $\chi^2(N-1, \alpha)$, where χ^2 denotes the standard Chi-squared table, which is partially presented in Table §9.7. If X^2 is greater or equal than $\chi^2(N-1, \alpha)$, then the execution is not good enough, so we need to increase k and return to Step 1; otherwise, this is the first value of k that produces an execution that complies with the expected distribution of interaction at confidence level α.

For instance, assume that $Think(i) = (i-6)^2+1$ and the confidence level is $\alpha = 0.995$. The expected number of lunches in a program with 12 philosophers and 500 lunches is shown in the second row of Table §9.8, whereas Table §9.9 shows an excerpt of the different actual distributions of lunches we obtained with different values of k and the value of X^2. Since there are 12 *Get* interactions, we need to iterate until we obtain the first value of X^2 less or equal than $\chi^2(11, 0.995) = 2.603$, which happens at $k = 161$. Note that this value is not the best possible, since $X^2 = 0.37$ if $k = 170$, which indicates that the execution is even better with this value. In general, we are not interested in finding an optimal value, but a value that allows us the consider the execution appropriate at a certain confidence level, and $\alpha = 0.995$ is quite a high confidence level.

Think(i)	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}
i	161	80	53	40	32	26	23	20	17	16	14	13
$(i-6)^2+1$	7	10	18	35	89	177	89	35	18	10	7	5
$\frac{1}{\sqrt{(i)}}$	17	24	29	34	38	41	45	48	51	54	56	59
$\frac{i}{\sin\frac{(2i+1)\pi}{2}}+12$	16	20	14	25	12	32	11	45	10	75	9	226

Table 9.8: *Expected number of lunches with several* Think *functions.*

k	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	X^2
0	42	42	42	42	41	42	42	42	42	42	41	42	1 055.71
10	37	39	42	44	47	48	47	44	42	39	37	35	818.55
20	34	37	41	46	52	52	52	46	40	37	34	29	658.35
30	25	33	40	47	64	65	64	47	40	33	25	18	379.54
40	20	30	43	53	65	65	65	52	43	31	20	14	318.29
50	13	20	34	58	81	81	80	58	35	20	13	9	146.76
60	11	17	28	55	90	91	90	55	28	17	11	7	89.95
70	11	17	28	51	91	92	91	56	28	17	11	8	87.37
80	10	15	26	50	97	98	97	50	26	15	10	7	64.71
90	9	14	24	48	101	102	101	48	24	14	9	6	52.90
100	9	14	23	45	104	105	104	45	23	14	9	6	47.24
110	8	13	21	42	105	116	105	43	21	13	8	6	33.24
120	8	13	21	42	103	120	103	42	21	13	8	6	28.78
130	8	12	21	41	100	130	101	41	21	12	8	5	19.63
140	8	12	20	40	98	140	98	39	20	12	8	5	12.23
150	8	12	19	38	95	150	95	38	20	12	8	5	6.75
160	7	11	19	38	93	160	93	37	19	11	7	5	2.63
161	7	11	19	38	93	161	92	37	19	11	7	5	2.35
170	7	11	18	36	90	171	90	36	18	11	7	5	0.37

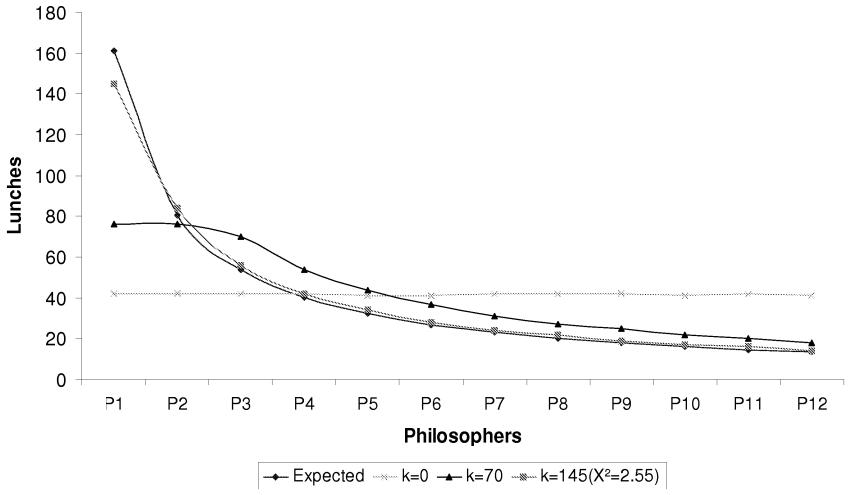
Table 9.9: *Finding an adequate value for* k *if* $\texttt{Think}(i) = (i-6)^2+1$.

Figures §9.11.(a), §9.11.(b), §9.12.(c), and §9.12.(d) show several charts in which the expected and the actual values are represented for several Think functions and values of k.

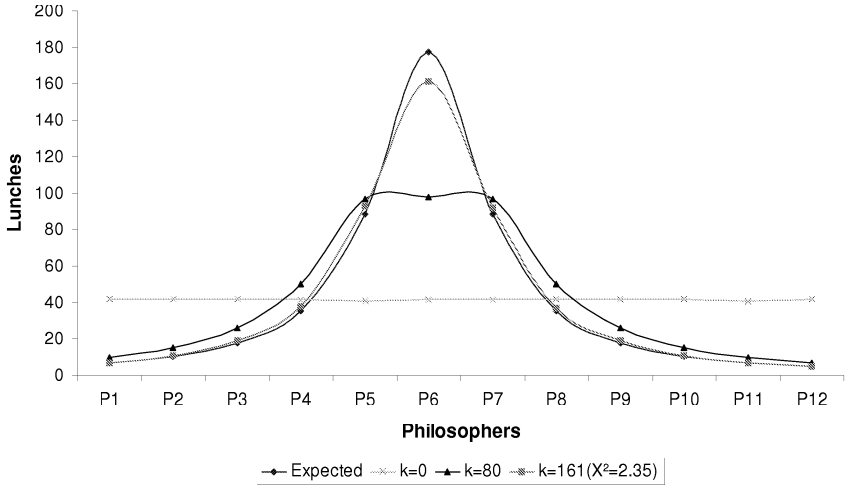
9.5 Summary

In this chapter, we have presented the results of a detailed experimental evaluation of our proposal. We have compared it with state-of-the art implementations of strong fairness and hyperfairness. The figures show that it helps solve conspiracies, but still performs comparably to other proposals that are less stringent. We have also presented a simple method to calculate a good value for k in the case of a dining philosophers system. This method can be easily adapted to other systems.

These results have been published partially in Concurrency and Computation: Practice & Experience [125].

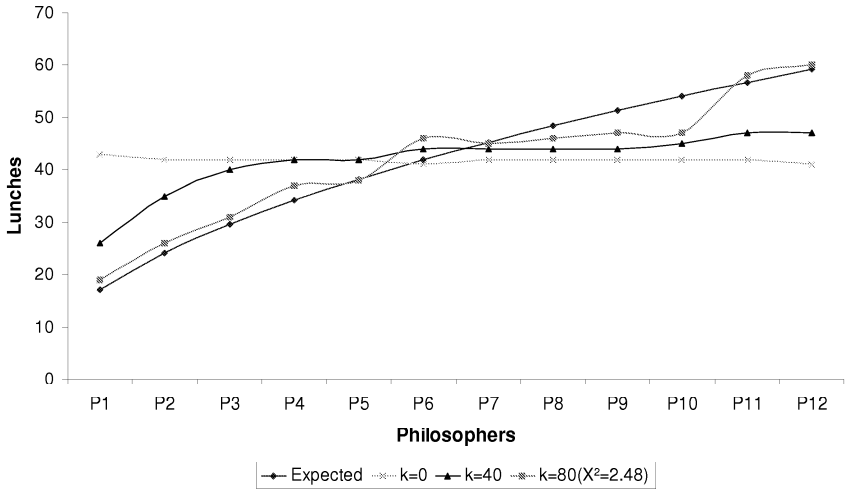


(a) $\mathrm{Think}(i) = i$.

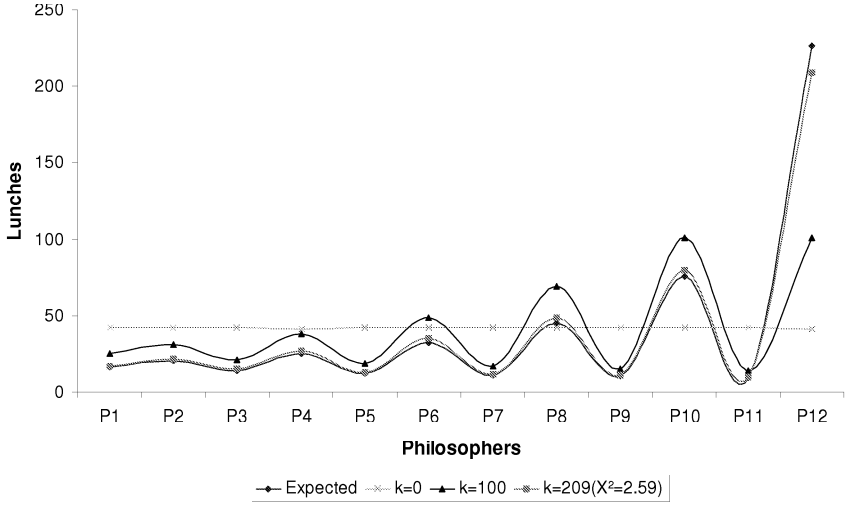


(b) $\mathrm{Think}(i) = (i-6)^2 + 1$.

Figure 9.11: *Tuning* k*-conspiracy-free fairness (I).*



(c) $\mathrm{Think}(i) = \frac{1}{\sqrt{i}}$.



(d) $\mathrm{Think}(i) = \frac{i}{\sin \frac{(2i+1)\pi}{2}} + 12$.

Figure 9.12: *Tuning* k*-conspiracy-free fairness (II).*

Part IV

Final remarks

Chapter 10

Conclusions and future work

When people agree, it is only in their conclusions; their reasons are always different.

Jorge A.N. de Santayana, 1863–1952
Spanish philosopher

Fairness is a key concept that allows programmers to abstract away from the many low-level details that concern the implementation and the execution of their programs. However, common notions in the field of multiparty interactions suffer from two problems known as fair finiteness and conspiracies, which constitute a major obstacle for their practical application. The implementation of some notions can deal with conspiracies using appropriate random number generators, but, in general, we do not think it is desirable since this makes the programs dependent on the specific scheduler use to run them.

Our goal in this thesis has been to devise a new fairness notion able to deal with both fair finiteness and conspiracies, so that programmers can think of how their programs run independently from the low-level details of the scheduler used. In this dissertation, we have shown that such a notion exists and can be implemented efficiently; furthermore, our empirical evaluation demonstrates that it can be adapted to the features of the system under consideration to a certain extent. As we mentioned in the abstract, fairness implies that every interaction should be treated conforming with the rules established in the program to which it belongs. This means that the goal is not to force every interaction to be executed as much as the rest, but to allow them to become enabled if they can and to execute sufficiently often. Amongst the notions we have analysed in this dissertation, k-conspiracy-free fairness is the only that can be adapted to the characteristics of the system under consideration

by tuning k appropriately, *i.e.*, independently from the actual scheduler used to implement it. To show the feasibility of our proposal, we have devised a theoretical framework to define it rigorously, as well as an efficient implementation using finite-state schedulers.

Anyhow, the results in this dissertation cannot be seen as the concluding end of a path, but as the motivation for further research on this topic. Amongst the many issues that remain open or can be improved, we think that the most exciting is to study how k, the threshold of potentially conspiratorial situations, may be adapted at run time. Currently, it is possible to determine which k best controls a finite execution whose length is known beforehand. If the actual execution is longer, the notion leads inevitably to a configuration from which no interaction can be selected unless the interactions that are linked to it are stable, which may be inefficient in some cases. It would be very interesting to be able to forecast if a conspiracy may happen and to adjust the value of k dynamically so that it can be avoided.

Part V

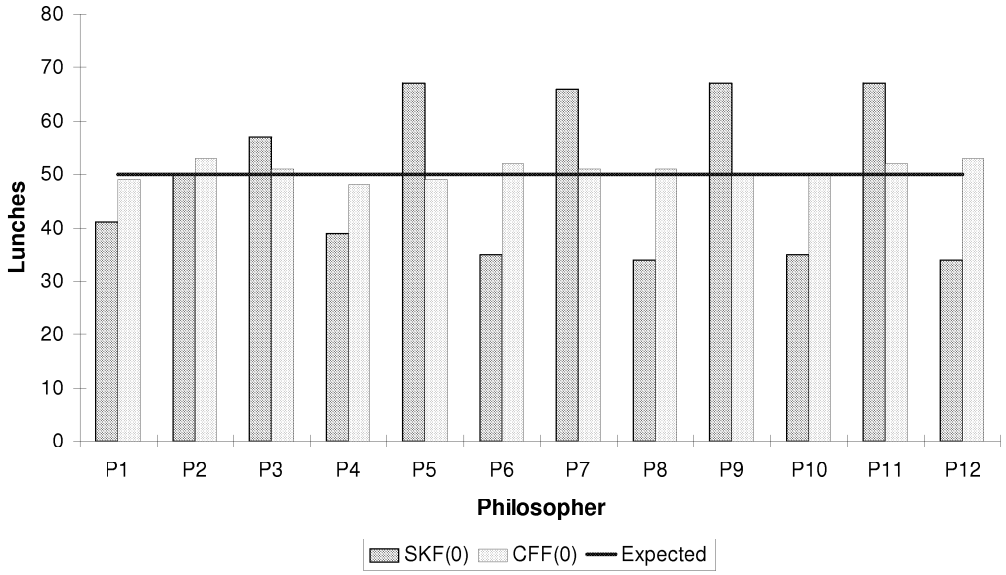
Appendices

Appendix A

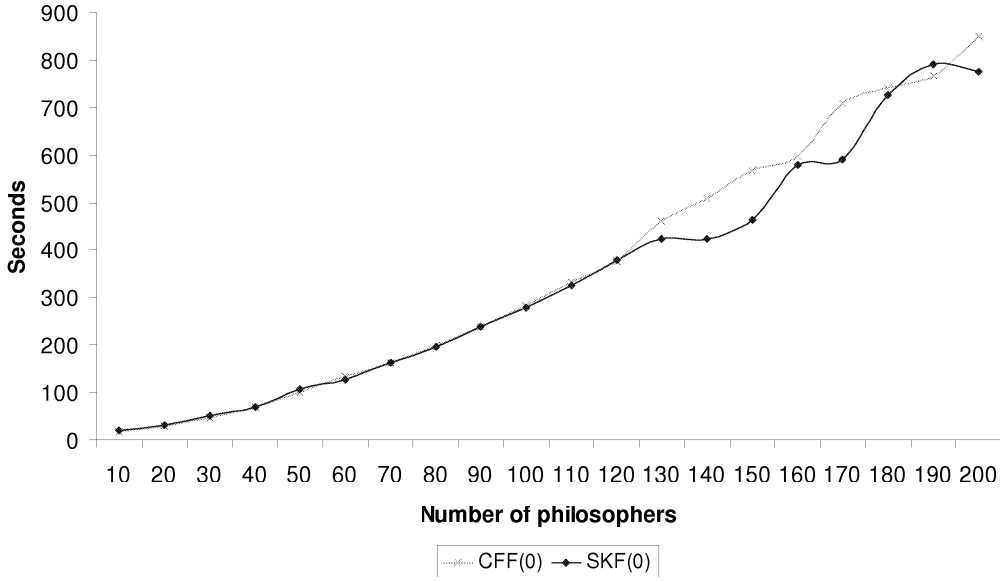
Strong k*-fairness*

A preliminary version of k-conspiracy-free fairness was presented at the Euro-Par'02 Conference [122]. There, we introduced a notion called *strong* k*-fairness*; it differs from the current proposal in that it attempts to detect potentially conspiratorial situations by analysing the interactions that are linked dynamically, instead of the interactions that are linked statically.

In spite of being so similar, the results are very different since k-conspiracy-free fairness can solve conspiracies whereas strong k-fairness can alleviate them only, *i.e.*, there are conspiratorial strongly fair executions that are strong k-fair. Figure §A.1 shows the results of an execution of the dining philosophers system in which all philosophers should have the same opportunities to have lunch. The figures show that k-conspiracy free fairness allows to control this execution better than strong k-fairness, but the effect on the efficiency is not significant.



(a) Distribution of lunches.



(b) Execution times.

Figure A.1: *Practical comparison between* k*-conspiracy-free fairness and strong* k*-fairness.*

Appendix B

The IP *language*

B.1 An overall picture

The IP language provides a high-level notation to model and design complex cooperations in concurrent and/or distributed systems. It was devised by Nissim Francez and Ira R. Forman, and it builds on their separate research ideas and precursory efforts on SCRIPT and RADDLE [59, 56]. Both proposals led to IP when the authors met at the MCC Software Technology Program [58].

IP provides three major programming constructs called multiparty interactions, teams, and superimposition. Multiparty interactions are adequate to describe complex cooperations amongst a fixed set of participating processes; they are high-level constructs that allow the programmer to abstract away from the low-level level protocols used to synchronise or communicate them since they can be generated automatically. Teams allow to encapsulate and reuse common cooperation patterns in which actual processes can participate by engaging into a role that allows them to interact with arbitrary processes. Superimposition is a construct that allows a process to observe and/or control the behaviour of another process; it is the cornerstone of a layered design method in which parts of a design may be reused very easily by superimposing a set of processes on it.

IP_{CORE} is a subset of IP that provides multiparty interactions only, *i.e.*, it does not provide any means for modularisation or abstraction other than processes. In this dissertation, we focus on IP_{CORE} instead of IP since our goal was to study multiparty interaction fairness, and the IP_{CORE} interaction model is conceptually simpler than the IP interaction model, although they both are quite similar in spirit. The results in this dissertation can be easily adapted to IP and other

languages that support similar interaction models, but the details fall outside our goals.

To the best of our knowledge, there are only two IP compilers available. One of them was designed by Francez and Forman's team; it runs on UNIX-like platforms and generates code for the MEIKO machine on either TRANSPUTER or i860 nodes. The other was designed by the author of this thesis in partial fulfilment for the requirements of his Master's Thesis; it runs on UNIX-like platforms and generates SR code as output [7]. Recently, several implementations of IP-like interaction models have appeared in the literature. For instance, a CORBA-based proposal was presented in Reference [108], and we describe our J♯-based implementation in Reference [118]. *Cf.* Section §2.1 for further references.

B.2 The syntax in a nutshell

In IP_{CORE}, systems are modelled as collections of cooperating sequential processes whose relationships are based on multiparty interactions. Each process executes a body that is composed of a sequence of commands. The general form of an IP_{CORE} program is the following:

SYSTEM :: $[P_1 \parallel P_2 \parallel \ldots \parallel P_n]$, **where**
P_1 :: *List of commands*$_1$;
P_2 :: *List of commands*$_2$;
⋮
P_n :: *List of commands*$_n$.

Although it is not usual, a program may be composed of one process only. In this case, the header can be omitted and the program is usually written as follows:

SINGLE_PROCESS :: *List of commands.*

If the name of this single process is not significant, the program can be reduced to its minimum form as follows:

List of commands.

The commands IP_{CORE} provides are assignments, interaction commands, multi-choice commands, and multi-choice loops. As usual, *assignments* are of the form $x := e$, where x denotes a local variable and e an expression over the local state of the process that executes this command. The null assignment is denoted as **skip**, and it may be used where a empty list of commands is needed. Arbitrary literals and operators are allowed to build expressions as

long as their semantics are well-defined in the context of a program. The variables need not be declared explicitly, but is is usual to list them as remarks between curly brackets.

Interaction commands are of the form $A\langle x_1 := e_1; \ldots; x_n := e_n\rangle$, where A is the name of an interaction and $x_1 := e_1; \ldots; x_n := e_n$ is an optional sequence of assignments that is referred to as the communication part. For an interaction to become enabled, all of its participants need to offer to participate in it by means of an interaction command. Once an interaction becomes enabled, it can be executed as long as its participants commit to it, *i.e.*, they do not withdraw their offers and commit to another interaction before. The communication part allows a process to retrieve data from other processes participating in interaction A since each e_i may refer to their local variables. Several improvements to this naive communication mechanism have been proposed in the literature [35, 38, 39, 141].

Multi-choice commands are of the form $[][]_{i=1}^{n}\ G_i \rightarrow S_i]$, where each G_i is a guard and S_i is a list of commands that can be executed as long as the guard is enabled. Typical guards are of the form $B\ \&A\langle x_1 := e_1; \ldots; x_n := e_n\rangle$, where B is a *guarding predicate* and the rest is an interaction command. They are enabled, *i.e.*, their corresponding commands can be executed, iff the guarding predicate holds and interaction A is enabled. If B & is omitted, then it is interpreted as *true* &; if $\&a\langle\overline{x := e}\rangle$ is omitted, it is interpreted as $\&\langle\rangle$, where $\langle\rangle$ denotes an anonymous, uniparty local interaction. Note that both parts of a guard cannot be omitted at a time.

Multi-choice loops are of the form $^*[\,[]\ _{i=1}^{n}\ G_i \rightarrow S_i\,]$. The semantics consists of repeating the multi-choice command until none of the guarding predicates is true. Note that they need not terminate if there is always an enabled guard.

B.3 Simple examples

In this section, we present several examples to illustrate IP_{CORE} basic features. Our goal is to give the reader a set of programs that may help him or her to understand IP_{CORE} commands more easily, but we do not go into many details about multiparty synchronisation and communication. *Cf.* Section §2.4 for more complex examples.

Example B.3.1 *The following program stores the maximum of vector v in variable result. Note that the variables are listed in a remark, but this is not strictly necessary. Since variable v is not initialised, we assume that its initial value is chosen*

randomly. No hypothesis may be made on the value of a variable whose initial value is not assigned explicitly.

```
{ result, i: natural; v: array (1..10) of natural }
result := 0; i := 1;
*[ i ≤ 10 →
    [ v(i) > result → result := v(i) ];
    i := i + 1
] .
```

Example B.3.2 *The following program computes the greatest common divisor of X and Y using the well-known Euclidean Algorithm. Note that we do not specify which the initial values of X and Y are since they are not important in this context, but both variables need to be initialised before the main loop begins executing.*

```
X := a value;
Y := a value;
*[ X < Y → Y := Y - X
[] X > Y → X := X - Y
]
gcd := X;
```

Example B.3.3 *In the following program, there are two processes whose goal is to exchange the values of their local states by means of interaction Exchange. Note that no two variables can have the same name, even if they belong to different processes. The reason is that* IP_{CORE} *provides a single global name space.*

```
EXCHANGE :: [P || Q], where
        P :: { x: natural }
             x := a value;
             Exchange⟨x := y⟩;
        Q :: { y: natural }
             y := a value;
             Exchange⟨y := x⟩.
```

Note that there is no need for a temporary variable since the semantics of multiparty communication guarantee that no race conditions may occur while an interaction is executing. When interaction Exchange takes place, a combined state is formed, and it contains a copy of the local state of every process participating in this interaction. The expressions on the communication part of an interaction command read data from this combined state, i.e., *they read data from the state before the interaction began. There is no way for a process to know how other processes are changing their state during an interaction, which prevents race conditions from occurring.*

Example B.3.4 *In the following program, we have modelled a typical system composed of a producer and a consumer that need to communicate indirectly by means of a fixed-sized buffer. This is a classical coordination problem in the literature.*

```
PROD-CONS :: [Producer || Consumer || Buffer], where
  Producer :: { x: Item }
              *[ true → x := Produce a new item; Produce⟨⟩ ] ;
  Consumer :: { y: Item }
              *[ true → Consume⟨y := Head(q)⟩; Consume item y ] ;
    Buffer :: { size: natural; MAX: natural; q: Queue of Item }
              size := 0; MAX := 10; q := NewQueue();
              *[ size < MAX & Produce⟨q := Enqueue(q, x)⟩ → size := size + 1
              [] size > 0 & Consume⟨q := Dequeue(q)⟩ → size := size - 1
              ] .
```

Note that when interaction Consume is executed, both the consumer and the buffer need to have access to the queue: the former needs to consult the head, whereas the buffer needs to eliminate it. Since the expressions on the right of the assignment commands in the communication part are evaluated in a combined state, this means that no race condition may occur in this example.

B.4 Interaction-driven programs

IP_{CORE} syntax is simple, but its ability to combine all commands orthogonally may lead to rather complex programs that may be difficult to analyse. In such cases, they may be transformed into equivalent so-called *interaction-driven normal form* programs that have the same behaviour, but flatten the commands in the body of every process so that they look as follows:

$$S_0; *[\,[]_{j=1}^{n_i} B_j \;\&\; a_j\langle v_j := e_j\rangle \rightarrow S_j\,]$$

where S_0 is an initialisation command that cannot contain any interaction command or loop, and each S_j is a command that contains local computation only. The authors of IP defined other normal forms, but we focus on the interaction-driven form only since it is required by hyperfairness, *cf.* Section §4.4.

Appendix C

Mathematical notes

C.1 Plotkin's method

We use the popular Plotkin's method to define our algorithms [112] since it is simple, yet powerful. It relies on inference rules of the following form:

$$\frac{\text{Antecedent}}{\text{Consequent}} \text{ if Applicability conditions}$$

where Definitions

To define an algorithm using this method, it is necessary to identify the data on which it works and model it as a tuple that is usually referred to as the configuration of the algorithm.

For instance, to model a simple producer/consumer system, we need configurations of the form (p, c, τ), where p denotes the state of the producer, c the state of the consumer, and τ is a fixed-sized queue that helps store the items that are ready to be consumed. Obviously, we also need a couple of rules denoted as $\longrightarrow_{\text{PROD}}$ and $\longrightarrow_{\text{CONS}}$ to describe how the producer and the consumer change their state, respectively. These rules may be left unspecified since the abstraction level at which we are describing the system does not require in-depth knowledge of their semantics. Using this information, the system can be described by means of the following rules:

i. The Production Rule, which controls how new items are stored in the queue as long if there is room for them. (We assume that MAX denotes the maximum number of items τ can store.)

$$\frac{p \xrightarrow{i}_{\text{PROD}} p'}{(p, c, \tau) \longrightarrow_{\text{PC}} (p', c', \tau')} \text{ if } |\tau| < MAX$$

where $c' \triangleq c$

$\tau' \triangleq Enqueue(\tau, i)$

$p \xrightarrow{i}_{\text{PROD}} p'$ means that the producer works locally to produce an item denoted as i, and this makes it to transit from state p to state p'. Note that the rule can be applied as long as $|\tau| < MAX$; otherwise, the producer has to wait until the consumer removes an item from the queue.

ii. The Consumption Rule, which controls how items are removed from the queue and consumed.

$$\frac{c \xrightarrow{i}_{\text{CONS}} c'}{(p, c, \tau) \longrightarrow_{\text{PC}} (p', c', \tau')} \text{ if } |\tau| > 0 \wedge i = Head(\tau)$$

where $p' \triangleq p$

$\tau' \triangleq Dequeue(\tau)$

Inference rules are very high-level and allow to model concurrent algorithms very precisely since they can be seen as the description of the atomic actions that can be performed in a concurrent system. The behaviour of the algorithm can thus be viewed as the set of all possible interleavings allowed by the inference rules used to describe it.

C.2 Büchi automata

Büchi automata were introduced in Reference [26] as an appropriate tool to recognise infinite words using finite-state automata. They may be viewed as four-tuples of the form (S, I, δ, F), where S denotes a finite set of states, $I \subseteq S$ a set of initial states, $\delta \subseteq S \times S$ a transition relation, and $F \subseteq S$ a set of final states. It is said that a sequence of states is accepted iff it contains infinitely many final states.

For instance, Figure §C.1 shows a simple non-deterministic Büchi automaton to recognise $(S_0 + S_1)^* S_2^\infty$. There exists procedures to minimise Büchi automata, but it is a well-known result that there does not exist an equivalent deterministic minimum Büchi automaton for every non-deterministic automaton. The procedures to minimise or compare these automata are exponential in general.

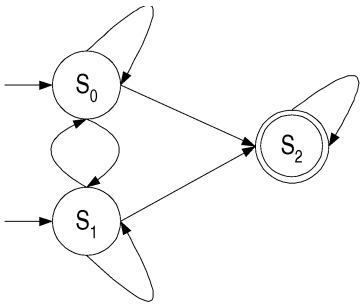


Figure C.1: *Büchi automaton to recognise* $(S_0 + S_1)^* S_2^\infty$.

These automata are important in theoretical computer science since it was shown that there exist automatic procedures to translate linear or branching temporal formulae into their equivalent Büchi automata [33]. In general, it is easier to study the properties of a Büchi automaton than the properties of its corresponding linear or branching temporal formula.

C.3 Framework notation

Table §C.1 shows the notation we have used to define the core of our theoretical framework and its implementation. Table §C.2 shows its predicates, and Table §C.3 presents its functions.

C.4 Mathematical notation

Table §C.4 summarises the mathematical notation we use through this dissertation.

Notation	Concept
Σ	A multiparty-based system
P_Σ	The set of processes in system Σ
I_Σ	The set of interactions in system Σ
$P, Q, \ldots, p, q, \ldots$	Sample processes
$A, B, \ldots, x, y, \ldots$	Sample interactions
$\mathbb{P}(x)$	The set of processes that may offer to participate in interaction x
$\mathbb{I}(p)$	The set of interactions in which process p can offer to participate
$p.\chi$	An event in which process p offers to participate in any of the interactions in set χ
$p.\emptyset$	An event by means of which process p indicates that it is finished
x	A synchronisation event in which interaction x is executed
λ	A generic execution
C_i, D_i	The i-th configuration of an execution
λ_α	The configuration trace of execution λ
λ_β	The event trace of execution λ
$\longrightarrow_L$	The rule that describes the semantics of the underlying interaction language
$\longrightarrow_{FW}$	The rule that describes the implementation of our framework
$\longrightarrow_{CFF_k}$	The rule that describes the implementation of k-conspiracy-free fairness

Table C.1: *Notation used to define our theoretical framework.*

Notation	Concept
$\mathit{Waiting}(\lambda, p, \Upsilon, i)$	It holds iff process p is waiting for any of the interactions in set $\Upsilon \subseteq \mathbb{I}(p)$ to be executed at the i-th configuration in execution λ
$\mathtt{Finished}(\lambda, p, i)$	It holds iff process p is finished at the i-th configuration in execution λ
$\mathtt{Enabled}(\lambda, x, i)$	It holds iff interaction x is enabled at the i-th configuration in execution λ
$\mathtt{Stable}(\lambda, x, i)$	It holds iff interaction x is stable at the i-th configuration in execution λ
$\mathtt{InfEna}(\lambda, x)$	It holds iff interaction x is enabled infinitely many times along execution λ
$\mathtt{InfSta}(\lambda, x)$	It holds iff interaction x becomes stable infinitely many times along execution λ
$\mathtt{InfExe}(\lambda, x)$	It holds iff interaction x is executed infinitely many times along execution λ
$\mathtt{InfOff}(\lambda, x, p)$	It holds iff process p offers to participate in interaction x infinitely many times during execution λ
$\mathtt{InfWait}(\lambda, x, p)$	It holds iff process p waits for interaction x to be executed at infinitely many configurations in execution λ
$\mathtt{SafeOffers}(\lambda)$	It holds iff the offer events in λ are safe
$\mathtt{SafeTerms}(\lambda)$	It holds iff the termination events in λ are safe
$\mathtt{SafeSyncs}(\lambda)$	It holds iff the synchronisation events in λ are safe

Table C.2: *Predicates used to define our theoretical framework.*

Notation	Concept
$\mathtt{StaLinked}(x)$	The set of interactions that are statically linked with x
$\mathtt{DynLinked}(\lambda, x, i)$	The set of interactions that are dynamically linked with x at the i-th configuration of λ
$\mathtt{EnaSet}(\lambda, x, i)$	The set of indices up to i that denote the configurations at which interaction x is enabled in execution λ
$\mathtt{StaSet}(\lambda, x, i)$	The set of indices up to i that denote the configurations at which interaction x is stable in execution λ
$\mathtt{ExeSet}(\lambda, x, i)$	The set of indices up to i that denote the configurations at which interaction x is executed in execution λ
$\mathtt{OffSet}(\lambda, \Upsilon, p, i)$	The set of indices up to i that denote the configurations at which process p offers to participate in any of the interactions in set Υ
$\mathit{WaitSet}(\lambda, x, p, i)$	The set of indices up to i that denote the configurations at which process p is waiting for any of the interactions in set Υ to be executed

Table C.3: *Functions used to define our theoretical framework.*

Notation	Concept
$\mathbb{N}$	The set of natural numbers
$\mathbb{N}^+$	The set of non-null, natural numbers
Ξ	A uniformly distributed random natural number
$[n_1, n_2]$	A closed natural interval
(n_1, n_2)	An open natural interval
$[n_1, n_2), (n_1, n_2]$	Semi-open natural intervals
$\{e_i\}_{i=1}^n$	A set defined by enumeration
$\{e \in A \mid P(e)\}$	A set defined by compression
$\lvert A\rvert$	The number of elements in set A
$A \cap B$	The intersection of sets A and B
$\bigcap_{i=1}^n A_i$	The distributive intersection of sets $A_1, A_2, \ldots, A_n$
$A \cup B$	The union of sets A and B
$\bigcup_{i=1}^n A_i$	The distributive union of sets $A_1, A_2, \ldots, A_n$
$A \setminus B$	The subtraction of set B from set A
2^A	The powerset of A
$\{a_i \mapsto b_i\}_{i=1}^n$	A map defined by enumeration
$\operatorname{dom}\sigma$	The domain of map σ
$\operatorname{ran}\sigma$	The image or range of map σ
$\sigma(a)$	The image of element a
$[e_i]_{i=1}^n$	A sequence defined by enumeration
$S_1 \frown S_2$	Concatenation of sequences S_1 and S_2
S^∞	The infinite concatenation $S \frown S \frown S \frown S \ldots$
$S(i)$	The element at the i-th position in sequence S
$\lvert S\rvert$	The number of elements in S
$\xrightarrow{\succ}_R$	A generic transition rule
$\sum_{i=1}^N P(i)$	Counts the number of times that $P(i)$ holds when i ranges in the interval $[1, N]$
$\forall^\infty$	For all, but finitely many
$\exists^\infty$	There exists infinitely many

Table C.4: *Mathematical notation.*

Bibliography

[1] N Abramson. The ALOHA system - Another alternative for computer communications. *AFIPS*, 37:281–285, 1970.

[2] MW Alford, JP Ansart, G Hommel, L Lamport, B Liskov, GP Mullery, and FB Schneider. *Distributed Systems - Methods and Tools for Specification - An Advanced Course*. Springer-Verlag, 1985.

[3] B Alpern and FB Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[4] B Alpern and FB Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[5] R Alur and TA Henzinger. Finitary fairness. *ACM Transactions on Programming Languages and Systems*, 20(6):1171–1194, 1998.

[6] E Andersen. *Conceptual Modeling of Objects: A Role Modeling Approach*. PhD thesis, University of Oslo, 1997.

[7] GE Andrews and RA Olson. *The SR Programming Language*. The Benjamin-Cummings Publishing Company, 1993.

[8] KR Apt, N Francez, and S Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2(4):226–241, 1988.

[9] F Arbab. A channel-based coordination model for component composition. Technical Report SEN-R0203, CWI, 2002.

[10] PC Attie, N Francez, and O Grumberg. Fairness and hyperfairness in multiparty interactions. *Distributed Computing*, 6(4):245–254, 1993.

[11] RJR Back and R Kurki-Suonio. Distributed co–operation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[12] RL Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, 1989.

[13] KE Batcher. Sorting networks and their applications. *Proceedings of AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[14] B Bauer, J Müller, and J Odell. Agent UML: A formalism for specifying multiagent interaction. *Lecture Notes in Computer Science*, 1957:91–103, 2001.

[15] B Bauer, J Müller, and J Odell. Agent UML: A formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.

[16] J Beauquier, AK Datta, M Gradinariu, and F Magniette. Self–stabilizing local mutual exclusion and daemon refinement. *Lecture Notes in Computer Science*, 1914:223–237, 2000.

[17] N Benton, L Cardelli, and C Fournet. Modern concurrency abstractions for C#. *Lecture Notes in Computer Science*, 2374:415–440, 2002.

[18] E Best. Erratum: Fairness and conspiracies. *Information Processing Letters*, 19(3):162, 1984.

[19] E Best. Fairness and conspiracies. *Information Processing Letters*, 18(4):215–220, 1984.

[20] E Best. *Semantics of Sequential and Parallel Programs*. Prentice Hall, New York, 1996.

[21] T Bolognesi and E Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems, North-Holland*, 14:25–59, 1987.

[22] G Booch. *Object-Oriented Design with Applications*. The Benjamin-Cummings Publishing Company, Redwood City, California, 1990.

[23] D Box, D Ehnebuske, G Kakivaya, A Layman, N Mendelsohn, HF Nielsen, S Thatte, and D Winer. Simple object access protocol (SOAP 1.1).

[24] E Brinksma. *On the design of Extended LOTOS – A specification language for open distributed systems*. PhD thesis, Department of Computer Science, University of Twente, 1988.

[25] M Broy and G Nelson. Adding fair choice to Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 16(3):924–938, May 1994.

[26] JR Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Method, and Philosophy of Science*, pages 1–12. Stanford University Press, 1962.

[27] G Caire, F Leal, P Chainho, R Evans, F Garijo, J Gómez, J Pavón, P Kearney, J Stark, and P Massonet. Agent oriented analysis using MESSAGE/UML. *Lecture Notes in Computer Science*, 2222:119–126, 2002.

[28] N Carriero and D Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35:97–107, 1992.

[29] D de Champeaux. Object-oriented analysis and top-down software development. *Lecture Notes in Computer Science*, 512:360–375, 1991.

[30] KM Chandy and J Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[31] A Charlesworth. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(2):350–366, 1987.

[32] P Chrząstowski-Wachtel. Generalisation of liveness and fairness properties. In *Proceedings of the* 8^{th} *Workshop on Applications and Theory of Petri Nets*, Zaragoza, Spain, 1987.

[33] EM Clarke, EA Emerson, and AP Sistla. Automatic verification of finite–state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 3(2):244–263, 1986.

[34] P Coad and E Yourdon. *Object-Oriented Analysis*. Yourdon Press, Englewood Cliffs, New Jersey, 1990.

[35] R Corchuelo. *Prototyping Constraint-Based Specifications of Distributed Systems*. PhD thesis, University of Seville, 1999.

[36] R Corchuelo, O Martín, M Toro, JL Arjona, and JM Prieto. Selección justa, optimación y superposición en el contexto de TESORO. In *Proceedings of the III Whorkshop MENHIR*, 1998.

[37] R Corchuelo and JA Pérez. Casale I: A new object–based language for discrete simulation. In *Proceedings of ECEC'98*, pages 310–312, 1998.

[38] R Corchuelo, JA Pérez, and A Ruiz-Cortés. Aspect-oriented interaction in multi-organizational web-based systems. *Computer Networks*, 41(4):385–406, 2003.

[39] R Corchuelo, JA Pérez, and M Toro. A multiparty coordination aspect language. *ACM Sigplan*, 35(12):24–32, 2000.

[40] R Corchuelo, JA Pérez, and M Toro. An order-based algorithm for implementing multiparty synchronisation. *Concurrency and Computation: Practice & Experience*, 2003. To be published.

[41] R Corchuelo, D Ruiz, M Toro, JL Arjona, and JM Prieto. A distributed solution to synchronous multiparty interaction. In *Proceedings of the 3RD IMACS/IEEE CSCC'99*, 1999.

[42] R Corchuelo, D Ruiz, M Toro, JL Arjona, and JM Prieto. Prototiping IPC on a network computer. In *Proceedings of the IV Whorkshop MENHIR*, 1999.

[43] R Corchuelo, D Ruiz, M Toro, and A Durán. Avances en la coordinación de objetos activos. *Novática*, 143:34–37, 2000.

[44] R Corchuelo, D Ruiz, M Toro, and A Ruiz. Implementing multiparty interactions on a network computer. In *Proceedings of the IEEE Euromicro'99*, Milan, Italy, 1999. IEEE Computer Society Press.

[45] R de Simone. Higher-level synchronization devices in MEIJE-SCCS. *Theoretical Computer Science*, 37:245–267, 1985.

[46] N Dershowitz, DN. Jayasimha, and S Park. Bounded fairness. *Lecture Notes in Computer Science*, 2772:To appear, 2003.

[47] EW Dijkstra. A class of allocation strategies inducing bounded delays only. *Manuscript, EWD 319*, 1971.

[48] EW Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[49] EW Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[50] EW Dijkstra. Position paper on "fairness". *Software Engineering Notes*, 3(2):18–20, 1988.

[51] DF D'Souza and AC Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Massachussets, 1999.

[52] R Englander. *Java and SOAPi*. O'Reilly & Associates, 2002.

[53] R Fagin and JH Williams. A fair carpool scheduling algorithm. *IBM Journal of Research and Development*, 27(2):133–139, 1983.

[54] M Fayad. E–Frame: A process-based, object-oriented framework for e-commerce. In *Proceedings of IC'01*, pages 124–128, Las Vegas, Nevada, 2001. CSREA Press.

[55] P Felber and MK Reiter. Advanced concurrency control in Java. *Concurrency and Computation: Practice & Experience*, 14(4):261–285, 2002.

[56] IR Forman. Design by decomposition of multiparty interactions in Raddle. 5^{th} *International Workshop on Software Specification and Design*, pages 2–10, 1989.

[57] N Francez. *Fairness*. Springer-Verlag, 1986.

[58] N Francez and I Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison-Wesley, 1996.

[59] N Francez, BT Hailpern, and G Taubenfeld. Script: A communication abstraction mechanism and its verification. *Science of computer programming*, 6(1):35–88, 1986.

[60] N Francez and A Pnueli. A proof method for ciclyc programs. *Acta Informatica*, 9:133–157, 1978.

[61] DM Gabbay, A Pnueli, S Shelah, and J Stavi. On the temporal basis of fairness. In *Proceedings of ACM POPL'80*, pages 163–173. ACM Press, 1980.

[62] MS Day GC Roman. Multifaceted distributed systems specification using processes and event synchronization. In *Proceedings of the IEEE ICSE'84*, pages 44–55. IEEE Computer Society Press, 1984.

[63] SM German. A language for specifying synchronization. Technical Report TM-0509-02-92-152, GTE Laboratories, 1992.

[64] SJ Hartley. *Operating Systems Programming: The SR Programming Language*. Oxford University Press, 1997.

[65] GH Hilderink. Communicating Java threads reference manual. In *Proceedings of WoTUG'20*, pages 283–325, Twente, Netherlands, 1997. IOS Press, Netherlands.

[66] CAR Hoare. *Occam programming manual*. Prentice Hall, 1984.

[67] CAR Hoare. *Communicating sequential processes*. Prentice Hall, 1985.

[68] RV Hogg and AT Craig. *Introduction to Mathematical Statistics*. MacMillan, 5^{th} edition, 1994.

[69] DN Jayasimha and N Dershowitz. Bounded fairness. Technical Report 615, Center for Supercomputing Research and Development, University of Illinois, 1986.

[70] Y-J Joung. *On the design and implementation of multiparty interaction*. PhD thesis, University of New York, 1992.

[71] Y-J Joung. Characterizing fairness implementability for multiparty interaction. *Lecture Notes in Computer Science*, 1099:110–121, 1996.

[72] Y-J Joung. Two decentralized algorithms for strong interaction fairness for systems with unbounded speed variability. *Theoretical Computer Science*, 243(1–2):307–338, 2000.

[73] Y-J Joung and SA Smolka. A completely distributed and message-efficient implementation of synchronous multiprocess communication. In *Proceedings of ICPP'90*, pages 311–318, Urbana-Champaign, Illinois, 1990. Pennsylvania State University Press.

[74] Y-J Joung and SA Smolka. Coordinating first-order multiparty interactions. *ACM Transactions on Programming Languages and Systems*, 16(3):954–985, 1994.

[75] Y-J Joung and SA Smolka. A comprehensive study of the complexity of multiparty interaction. *Journal of the ACM*, 43(1):75–115, 1996.

[76] Y-J Joung and SA Smolka. Strong interaction fairness via randomization. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):137–149, 1998.

[77] A Keen, T Ge, J Maris, and R Olsson. JR: Flexible distributed programming in an extended Java. In *Proceedings of the IEEE ICDCS'01*, pages 575–584, Los Alamitos, California, 2001. IEEE Computer Society.

[78] E. Kindler and W. van der Aalst. Liveness, fairness, and recurrence in petri nets. *Information Processing Letters*, 70(6):269–274, 1999.

[79] E Kindler and R Walter. Mutex needs fairness. *Information Processing Letters*, 62(1):31–39, 1997.

[80] D Kumar. An implementation of n-party synchronization using tokens. In *Proceedings of CDCS'90*, pages 320–327. IEEE Computer Society Press, 1990.

[81] MZ Kwiatkowska. Infinite behaviour and fairness in concurrent constraint programming. In *Semantics: foundations and applications*, number 666 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[82] L Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[83] L Lamport. Fairness and hyperfairness. *Distributed Computing*, 13:239–245, 2000.

[84] L Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison–Wesley, 2003.

[85] G Le Lann. Distributed systems: Towards a formal approach. In *Proceedings of IFIP'77*, pages 155–160, 1977.

[86] D Lea. *Concurrent Programming Using Java: Design Principles and Pattern*. Addison-Wesley, 1999.

[87] D Lea, J Bowbeer, B Goetz, D Holmes, C McCorvey, and T Peierls. JSR 166: Concurrency Utilities, 2002.

[88] DJ Lehmann, A Pnueli, and J Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. *Lecture Notes in Computer Science*, 115:264–277, 1981.

[89] F Leymann and IBM Software Group. Web services flow language (WSFL 1.0). Available at `http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

[90] P Li. *BizTalk Server Developer's Guide*. Osborne McGraw–Hill, 2001.

[91] NA Lynch. Fast allocation of nearby resources in a distributed system. In *Proceedings of ACM STC'80*, pages 70–81. ACM Press, April 1980.

[92] NA Lynch, M Merritt, WE Weihl, and A Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, 1994.

[93] TW Malone. What is coordination theory? Technical Report 2051-88, MIT Sloan School of Management, 1988.

[94] TW Malone and K Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26:87–119, 1994.

[95] Z Manna and A Pnueli. *The Temporal Logic of Reactive and Concurrent Systems-Specification*. Springer-Verlag, 1992.

[96] O Martín. *Quality-Aware Software Procurement*. PhD thesis, University of Seville, Under development.

[97] GJ Milne. CIRCAL and the representation of communication, concurrency, and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–289, 1985.

[98] R Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[99] MJ Norusis. *SPSS 11.0 Guide to Data Analysis*. Prentice Hall, 2002.

[100] J Odell, HVD Parunak, and B Bauer. Representing agent interaction protocols in UML. *Lecture Notes in Computer Science*, 1957:121–140, 2001.

[101] US Department of Defense. Reference manual for the Ada programming language. Technical Report ANSI/MIL-STD-1815A, US Goberment, 1983.

[102] S Older. Strong fairness and full abstraction for communicating processes. *Information and Computation*, 163(2):471–509, 2000.

[103] E Olderog and KR Apt. Fairness in parallel programs: The transformational approach. *ACM Transactions on Programming Languages and Systems*, 10(3):420–455, 1988.

[104] G Papadopoulos and F Arbab. Coordination models and languages. In *Advances in Computers*, volume 46. Academic Press, 1998.

[105] J Peña. *The Joint Processes Specification: A Methodological Fragment*. PhD thesis, University of Seville, Under development.

[106] J Peña, R Corchuelo, and JL Arjona. Towards interaction protocol operations for large multiagent systems. *Lecture Notes in Computer Science*, 2699:79–91, 2002.

[107] J Peña, R Corchuelo, and JL Arjona. A top down approach for mas protocol descriptions. In *Proceedings of the ACM SAC'03*, pages 45–49, Melbourne, Florida, 2003. ACM Press.

[108] JA Pérez. *An aspect-oriented framework to describe the coordinated behaviour. An application to multiorgisational systems.* PhD thesis, Facultad de Informática y Estadística. Dpto. de Lenguajes y Sistemas Informáticos. University Of Seville, 2001.

[109] JA Pérez, R Corchuelo, D Ruiz, and M Toro. A framework for aspect–oriented multiparty coordination. In *DAIS*, volume 198 of *IFIP Conference Proceedings*, pages 161–173. Kluwer, 2001.

[110] JA Pérez, R Corchuelo, D Ruiz, and M Toro. An enablement detection algorithm for open multiparty interactions. In *Proceedings of the ACM SAC'02*, pages 378–384, Madrid, Spain, 2002. ACM Press.

[111] JA Pérez, R Corchuelo, D Ruiz, and M Toro. An order–based, distributed algorithm for implementing multiparty interactions. *Lecture Notes in Computer Science*, pages 250–257, 2002.

[112] GD Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[113] JP Queille and J Sifakis. Fairness and related properties in transition systems – A temporal logic to deal with fairness. *Acta Informatica*, 19(3):195–220, 1983.

[114] MJ Quinn. *Designing Efficient Algorithms for Parallel Computers*. MacGraw Hill, 1987.

[115] I Rammer. *Advanced .NET Remoting*. APress, 2002.

[116] T Reenskaug, P Wold, and OA Lehne. *Working With Objects. The OOram Software Engineering Method*. Manning Publications Co., 1995.

[117] A Ruiz, R Corchuelo, JA Pérez, A Durán, and M Toro. An aspect–oriented approach based on multiparty interactions to specifying the behaviour of a system. In *Proceedings of PLI'99*, pages 56–65, Paris, France, 1999.

[118] D Ruiz and R Corchuelo. A J♯ framework to implement multiparty interactions. Technical Report 03-001, The Distributed Group, 2003.

[119] D Ruiz, R Corchuelo, JA Pérez, O Martín-Díaz, and A Ruiz-Cortés. Avances en la implementación del modelo de interacción entre múltiples participantes. In *Proceedings of the 5*$^{\text{t}}$*h Workshop MENHIR*, pages 110–117, Granada, Spain, 2000.

[120] D Ruiz, R Corchuelo, JA Pérez, and M Toro. Sincronización multiproceso en programas concurrentes: Selección completamente k–justa de interacciones. In *Proceegings of PROLE'01*, pages 279–294, Almagro, Spain, 2001.

[121] D Ruiz, R Corchuelo, JA Pérez, and M Toro. Una nueva noción de selección de interacciones. In *Proceedings of SI + T'01*, pages 389–401, A Coruña, Spain, 2001.

[122] D Ruiz, R Corchuelo, JA Pérez, and M Toro. An algorithm for ensuring fairness and liveness in non–deterministic systems based on multiparty interactions. *Lecture Notes in Computer Science*, 2400:563–572, 2002.

[123] D Ruiz, R Corchuelo, JA Pérez, and M Toro. A theoretical, multiparty framework to define new fairness notions. In *Proceegings of PROLE'02*, pages 23–38, San Lorenzo del Escorial, Spain, 2002.

[124] D Ruiz, R Corchuelo, and A Ruiz-Cortés. Vivacidad y justicia en entornos no deterministas. *Revista Colombiana de Computación*, 3(1):73–92, 2002.

[125] D Ruiz, R Corchuelo, and M Toro. Fairness in systems based on multiparty interactions. *Concurrency and Computation: Practice & Experience*, 15(11–12):1093–1116, 2003.

[126] D Ruiz, R Corchuelo, M Toro, A Durán, and JL Arjona. Implementación de interacciones entre múltiples participantes sobre una maquina virtual en red. In *Proceedings of SEID'99*, pages 183–193, Santiago de Compostela, Spain, 1999.

[127] A Ruiz-Cortés. *Una Aproximación Semicualitativa al Tratamiento Automático de Requisitos de Calidad*. PhD thesis, University of Sevilla, 2002.

[128] J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorensen. *Object–Oriented Modelling and Design*. Prentice Hall, Schenectady, New York, 1991.

[129] J Rumbaugh, I Jacobson, and G Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, Reading, Massachussets, 1999.

[130] PG Sarang, E Adatia, and B Jouhier. *J#*. Wrox Press, Inc., 2002.

[131] F B Schneider and L Lamport. Another position paper on "fairness". *Software Engineering Notes*, 13(3):1–2, 1988.

[132] BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems. Business process execution language for web services version 1.1. Available at `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/`.

[133] C Szyperski, D Gruntz, and S Murer. *Component Software*. Component Software Series. Addison–Wesley, 2 edition, 2002.

[134] AS Tanenbaum. *Computer Networks*. Prentice Hall, 1988.

[135] P Tang and Y Muraoka. Parallel programming with interacting processes. In *Proceedings of LCPC'99*, pages 201–218, 1999.

[136] P Tang and Y Muraoka. Parallel programming with interacting processes. *Lecture Notes in Computer Science*, 1863:201–218, 2000.

[137] S Thatte and Microsoft. XLANG: Web services for business process design. Available at `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`.

[138] YK Tsay and RL Bagrodia. Some impossibility results in interprocess synchronization. *Distributed Computing*, 6(4):221–231, 1993.

[139] R Wirfs-Brock and B Wilkerson. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[140] P Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[141] AF Zorzo and RJ Stroud. A distributed object-oriented framework for dependable multiparty interactions. *ACM Sigplan*, 34(10):435–446, 1999.

Index

artefacts, 4

bad things, 31
bounded eventuality, 66

$\mathcal{CFF}_k$, 93
Conspiratorial, 86
commands
  assignment, 148
  compound, 16
  interaction command, 149
  multi-choice command, 149
  multi-choice loops, 149
concurrency
  competitive, 4
  cooperative, 4
  element, 33
concurrent system, 4
configuration, 82
configuration trace, 82
conspiracies, 68, 86, 96
  inherent, 69
  non-inherent, 69
  resistance, 58
Conspiratorial Synchronisation Rule, 110
coordination, 5

Deadlock, 86
Disabled, 83
DynLinked, 83
Dovetail operator, 48

EnaSet, 84
Enabled, 83
ExeSet, 84
event trace, 82
events
  $p.\chi$, 82
  $p.\emptyset$, 82
  χ, 82
eventuality, 66
execution, 32
  enabled set, 84
  $\longrightarrow\!\!\!\!\!\rightarrow_L$, 82
  λ_α, 82
  λ_β, 82
  offer set, 84
  partial, 32
  stable set, 84
  synchronisation set, 84

Finished, 83
fair finiteness, 68, 95
fairness, 17, 19, 20
  0-fair, 37
  0-fairness, 53
  ∞-, 52
  ∞-fairness, 57
  k-bounded, 37
  k-conspiracy-free, 8, 90, 93
  k-fair, 35
  absolute, 36
  all levels, 36
  bounded-delay, 36, 43
  economy-based, 36, 52
  eventuality-based, 36
  finitary, 37
  hyperfairness, 35, 57
  interactions, 33
  oracle-based, 35
  order-based, 36
  processes, 33

quantitative-based, 36
strong, 35, 37, 40, 87
strong k-fairness, 90, 145
strong fairness, 96
strong interaction fairness, 52
top level, 36
unconditional, 36, 86
void notion, 43
weak, 37, 40, 87
weak finitary, 52
weak-k-bounded, 52
framework
finished set, 100
offer map, 100

good things, 31
googolplex, 68
guarding predicate, 149

I_Σ, 81
interaction, 81
disabled, 78, 83
dynamically linked, 78, 83
enabled, 78, 83
language, 5
linked, 83
models, 5
participants, 78
stable, 78, 83
statically linked, 78, 83
unstable, 78, 83
interaction-driven normal form, 151

k-activation, 34

EnablementDegree, 87
RejectionRatio, 87
StaLinkageDegree, 88
liveness
absolute, 33
uniform, 33
liveness properties, 31
logical assumptions, 80

metrics
enablement degree, 87
linkage degree, 88
rejection ratio, 87
minimal progress property, 33, 80
multiparty interaction, 6, 78
primitives, 14
multiparty-based system, 78, 81
execution, 82
interaction, 83
participant, 83
safe offers, 85
safe synchronisations, 85
safe terminations, 85
semantics, 82
trasition rule, 82

OffSet, 84
SafeOffers, 85
Offer Rule, 102
offering events, 78, 82

P_Σ, 81
participant, 78, 81
physical assumptions, 80
process state
blocked, 78
finished, 78, 83
local computation, 83
waiting, 78, 83
working, 78
properties
compassion, 35, 53
deadlock, 86
finitary liveness, 43
justice, 37
liveness, 86
minimal progress, 37
safety, 86

Recording Rule, 110
Right implementation, 43

SF, 87
SafeSyncs, 85
SafeTerms, 85
StaLinked, 83
StaSet, 84
Stable, 83
safety properties, 31

Stabilising Synchronisation Rule, 110
strictness levels, 34
sufficiently often, 33
synchronisation events, 78, 82
Synchronisation Rule, 103
system
    configuration, 78
    execution, 78

temporal logic
    branching, 41
    linear, 41
    of actions, 41
Termination Rule, 102
Theory of Big Numbers, 56

UF, 86
unbounded non-determinism, 33, 37

void obligation, 93

WF, 87
WaitSet, 84
Waiting, 83
Working, 83

This document was typeset on 1999/10/15 using RC-BOOK α2.2 for LaTeX$_{2\varepsilon}$.
Should you want to use this document class, please send mail to
contact@tdg-seville.info.