Epistemic Strategies and Games on Concurrent Processes

Sophia Knight

Master of Science

Reasoning and Learning Lab
School of Computer Science

McGill University

Montréal,Québec

June 2009

A thesis submitted to McGill University
in partial fulfilment of the requirements of
the degree of Master of Science

ACKNOWLEDGEMENTS

I am grateful to everyone who has helped and supported me during my master's degree. Most of all, I would like to thank Prof. Prakash Panangaden, who has always been exceptionally encouraging, patient, generous, and positive about my interests and my work. I thank Prof. Catuscia Palamidessi and Kostas Chatzikokolakis for numerous enlightening discussions and ideas, and for their enthusiasm in answering all of my questions. I am very lucky to be able to work with Prakash, Catuscia and Kostas. I also thank Romain Beauxis, who has been inspiring and stimulating to work with. I am grateful to Prof. Samson Abramsky, Yannick Delbecque and Cait Phillips for helpful discussions and suggestions. I would like to thank Monica, Jesse, Jordan and Kamal for encouraging and entertaining me, and for all the coffee! I am most grateful to my husband Justin Dean for his understanding and constant support.

ABSTRACT

I develop a game semantics for process algebra with two independent, interacting agents. The purpose of the semantics is to make manifest the role of knowledge and information flow in the interactions between agents and to control the information available to interacting agents. I define games and strategies on process algebras, so that two agents interacting according to their strategies determine the execution of the process, replacing the traditional scheduler. I show that different restrictions on strategies represent different information being available to a scheduler. I also show that a certain class of strategies corresponds to the syntactic schedulers of Chatzikokolakis and Palamidessi, which were developed to overcome problems with traditional schedulers modelling interaction. The restrictions on these strategies have an explicit epistemic flavour. I extend these results to a probabilistic process algebra. Finally, I define a modal logic with both epistemic and temporal modalities to capture how information flows through the system. I give a logical characterization of a concept called introspection which is used to place restrictions on the allowed strategies of an agent. Thus, the logic makes precise what agents "know."

ABRÉGÉ

Je présente dans ce mémoire une sémantique des jeux pour l'algèbre de processus avec deux agents indépendants qui interagissent. Le but de cette sémantique est de permettre de comprendre le flux d'information et le rôle de la conaissance, ainsi que de contrôler l'information qui est disponible aux agents qui interagissent. Je définis les jeux et les stratégies pour les algèbres de processus, de facon à ce que les deux agents qui suivent ces stratégies lors de leurs interactions déterminent l'exécution du processus, remplaçant ainsi l'ordonnanceur traditionnel. Je démontre que différentes contraintes de stratégie représentent des informations différentes disponibles pour l'ordonnanceur. Je démontre aussi qu'une certaine classe de stratégies correspond aux les ordonnanceurs traditionnels qui modelisent les interactions entre agents. Les contraintes de stratégie ont des aspectes épistemiques explicites. Je généralise ces resultats à l'algèbre de processus. Enfin, je présente une logique modale avec des modalités épistémiques et temporales, afin de comprendre le flux d'information dans le système. Je présente une caractérisation logique d'un concept appelé l'introspection, qui est utilisé pour contraindre la stratégie d'un agent, faisant ainsi en sorte que la logique précise ce que les agents conaissent.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
Introduction

Concurrent processes are a natural and widely used model of interacting agents. Process algebra combines an operational semantics for processes with equational laws of process behaviour. Thus, process algebra has been widely used to model security protocols, which usually involve concurrent activity, interaction, and nondeterministic behaviour. For example, in [AG97], a special process calculus was developed for describing and analyzing cryptographic protocols, representing protocols as processes in order to understand their security properties. Also, in [SS96], a definition of anonymity as a kind of equivalence within a process calculus is proposed, so that anonymity properties can be formally proven about systems represented as processes. However, process algebra - as traditionally presented - has no explicit epistemic concepts, making it difficult to discuss what agents know and what has been successfully concealed. Epistemic concepts and indeed modal logics capturing "group knowledge" have proven very powerful in distributed systems [HM84, FHMV95]. Strangely, it has taken a long time for these ideas to surface in the process algebra community.

Epistemic concepts play a striking role in the resolution of nondeterministic choices. Typically one introduces a *scheduler* (or *adversary*) to resolve nondeterminism. This scheduler represents a single global entity that resolves all the choices. Furthermore, traditional schedulers are effectively omniscient: they may use the entire past history as well as all other information in order to resolve the choices. This is reasonable when one is reasoning

about correctness in the face of an unknown environment. In this case one wants a quantification over all possible schedulers in order to deliver strong guarantees about process behaviour.

In security, however, one comes across conditions where omniscient schedulers are unreasonably powerful, creating circumstances where one cannot establish security properties. The situation is as follows. One wants to set up protocols that *conceal* some action(s) from outside observers. If the scheduler is allowed to see these actions and reveal them through diabolical scheduling decisions there is no hope for designing a protocol that conceals the desired information. For example, randomness is often used as a way of concealing information; if the scheduler is allowed to see the results of random choices and code these outcomes through scheduling policies then randomness has no power to obfuscate data.

Consider for instance a voting system which collects people's votes for candidate a or b, and outputs in some arbitrary order the list of people who have voted (for example to check whether everyone has voted). Among the possible schedulers, there is the one which lists first all the people who voted for a. Clearly, this scheduler completely violates the desired anonymity property. Usually when we want a correctness property to hold for a nondeterministic system we require that it hold for *all* choices of the scheduler: there is no way such universally quantified statements will be true if we permit such omniscient schedulers.

How then is process algebra traditionally used to treat security issues? In fact scrutiny reveals that they do not have a completely demonic scheduler all the time. For example, Schneider and Sidiropoulos [SS96] argue that a system is *anonymous* if the set of (observable) traces produced by one

user is the same as the set of traces produced by another user. This is, in fact, an extremely *angelic* view of the scheduler. A perverse scheduler can most definitely leak information in this case by ensuring that certain traces never appear in one case *even though the operational semantics permits them*. Even a probabilistic (hence not overtly demonic) scheduler can leak information as discussed by Bhargava and Palamidessi[1] [BP05]. Anonymity is a problem where these issues manifest themselves particularly sharply.

Bisimulation is often used to handle nondeterminism in the analysis of security properties. Bisimulation is powerful for reasoning about concurrency because it combines algebraic and logical principles. In security, a common method is to prove that a protocol is bisimilar to a certain set of specifications with good security properties, thus proving that the protocol also has the correct security properties, as discussed in [CNP09]. Bisimulation can also be used to prove that different instances of a protocol are indistinguishable, which is useful for properties like secrecy and anonymity: if two instances of a protocol with two different agents responsible for some action like sending a message are bisimilar, then they are considered indistinguishable, meaning that the protocol preserves anonymity, as in [SS96]. Similarly, bisimulation can be used to show that a protocol preserves secrecy. Even bisimulation, however, does not treat non-determinism in a purely demonic way. If one looks at its definition, there is an alternation of quantifiers: s is bisimilar to t is *for every* $s \xrightarrow{a} s'$ *there exists* t' such that t

[1] They do not explicitly talk about schedulers in their paper but the import is the same.

$\xrightarrow{a} t'$ ... This definition implies that the scheduler that chooses the a transition for s is demonic whereas the scheduler that chooses the corresponding transition for t is *angelic*.

One approach to solve the problem of reasoning about anonymity in the presence of demonic schedulers has been suggested in [CP07]: the interplay between the secret choices of the process and the choices of the scheduler is expressed by introducing two *independent* schedulers and a framework that allows one to switch between them.

The ideas of demonic versus angelic schedulers, the idea of independent agents and the presence of epistemic concepts all suggest that *games* are a unifying theme. In this thesis we propose a game-based *semantic* restriction on the information flow in a concurrent process. We introduce a turn-based game that is played between two agents and define strategies for the agents. The game is played with the process as the "playing field" and the players' moves roughly representing the process executing an action. The information to which a player does not have access appears as a restriction on its allowed strategies. This is in the spirit of game semantics [AJ94, HO00, AJM00] where restrictions on strategies are used to describe limits on what can be computed. The restrictions we discuss have an epistemic character which we model using Kripke-style indistinguishability relations.

We show that there is a particular epistemic restriction on strategies that exactly captures the syntactic restrictions developed by Chatzikokolakis and Palamidessi [CP07]. It should be noted that this correspondence is significant since it only works with one precise restriction on the strategies, which characterizes the knowledge of the schedulers. This restriction is an

important achievement because although Chatzikokolakis and Palamidessi showed that with these schedulers certain equations hold, it was a purely informal argument that these equations had any epistemic significance. The treatment described in this thesis represents the first time that the epistemic qualities of these schedulers have been made explicit.

Finally, we introduce a modal logic with both epistemic and temporal modalities which can be used to reason about the labelled process calculus that we present. This logic allows us to explicitly discuss the knowledge of an agent in the system, as well as the effect of actions on agents' knowledge. Thus, our logic allows us to formalize the information flow in the system as well as the interaction between knowledge and actions. Furthermore, we are able to give a logical characterization of the introspection concept, which we used in the game semantics to restrict the allowed strategies for an agent. This characterization allows us to draw further conclusions about agents' knowledge in any situation.

The advantage to thinking in terms of strategies is that it is quite easy to capture restrictions on the knowledge of the agents as restrictions on the allowed strategies. For example, if one were to try to introduce some entirely new restriction on what schedulers "know" one would have to rethink the syntax and the operational semantics of the process calculus with schedulers and work to convince oneself that the correct concept was being captured. With strategies, one can easily add such restrictions and it is clear that the restrictions capture the intended epistemic concept. Furthermore, the strategies and game semantics framework allowed us to develop a formal logic to reason about knowledge and actions in our system. For instance, our notion of introspection makes completely manifest what the agents

know since it is couched as an explicit statement of what the moves can depend on, and thus, the agents' knowledge can be formally stated and reasoned about in our logic. On the other hand, previously one only had an intuitive notion of what the schedulers of [CP07] "knew" and it required some careful design to come up with the right rules to capture this in the operational semantics. Thus, strategies and restrictions are an excellent way to model interaction and independence in process algebra.

Related work

There are many kinds of games used in mathematics, logic and computer science. Games are also used widely in economics, although these are quite different from the games that we consider. Even within logic there is a remarkable variety of games. The logical games most related to our games are Lorenzen games. Lorenzen games are *dialogues* that follow certain rules about the patterns of questions and answers. There is a notion of winning and the main results concern the correspondence between winning strategies and the existence of constructive proofs. The idea of dialogue games appears in programming language semantics culminating with the deep and fundamental results of Abramsky, Jagadeesan, Malacaria [AJM00] and Hyland and Ong [HO00] on full abstraction for PCF. These games do not have a notion of winning. Rather the games simply delineate sets of possible plays and *strategies* are used to model programs. This has been a fruitful paradigm to which many researchers - far too many to enumerate - have contributed. It has emerged that games of this kind form a semantic universe where many kinds of language features coexist. Different features are simply modelled by different conditions on the strategies.

The games that we describe are most similar to these kinds of games in spirit but there are crucial differences. Our games are not dialogue games and there is no notion of question and answer, as a result, conditions like well-bracketing have no meaning in our setting. There is no notion of winning in our games either. Our games are specifically intended to model multiple agents working in a concurrent setting. While there have been some connections drawn between concurrent languages like the π-calculus and dialogue games [HO00] these are results that say that π-calculus can be used to describe dialogue games, not that dialogue games can be used to model π-calculus. The latter remains a fundamental challenge and one that promises to lead to a semantic understanding of mobility.

"Innocence" is an important concept pervading game semantics [HO00, DH01]. This is a very particular restriction on what the players know. In order to define innocence much more complex structures come into play; one needs special indicators of dependence (called "justification pointers") that are used to formalize a concept called the "view" of each process. In the end innocence, like introspection, is a statement about what knowledge the agents have. Our games have much less complicated structure because there are no issues with higher types and the introspection notion is relatively simple to define.

Outline

The outline of this thesis is as follows. In chapter 2, I give a brief outline of basic process calculus and a basic discussion of bisimulation and Hennessy-Milner logic. In chapter 3, I present a labelled process calculus with a special operator representing independent choice. In chapter 4, I define the execution of these processes as a two player game where each player

has a strategy, and I discuss certain restrictions on the players' strategies representing limitations on their knowledge. In chapter 5 I prove that execution of the processes according to the strategies defined in chapter 4 is equivalent to the execution of the processes with syntactic schedulers developed in [CPP06]. In chapter 6, I extend our results to processes with probabilistic choice, first defining games and strategies for these processes, then proving that the execution of these processes according to strategies is also equivalent to the execution of probabilistic processes with syntactic schedulers. In chapter 7, I present a modal logic with epistemic and temporal modalities, used to discuss the processes presented in chapter 3, and I use the logic to prove a result about the agents' knowledge in this context.

The main novel contributions of this thesis are reported in chapters 5 and 6. The results of chapter 5 were published in [CKP09].

CHAPTER 2
Background on Process Calculus

In this chapter I begin with a brief outline of traditional process calculus. Process calculi are used to model concurrent systems. They often model multiple agents or processes interacting and communicating. There is a great variety of process calculi used for different purposes, but I present a simple, basic process calculus based on CCS [Mil80]. I first introduce the syntax and operational semantics, then discuss bisimulation equivalence for these processes, and Hennessy-Milner logic, a special modal logic that characterizes bisimulation equivalence.

2.1 Syntax and Semantics

We start with a set of actions, denoted a, b,... Each action has a unique co-action $\bar{a}$, $\bar{b}$,... There is also a unique silent action τ. In the following discussion, α and β represent an action, co-action, or silent action, and P and Q represent processes. Here is the syntax for a process:

$$P ::= a.P \mid P + Q \mid P|Q \mid (\nu a)P \mid !P \mid 0$$

In the syntax, 0 represents the empty process. From the syntax, a few examples of processes are $a.0 + (\bar{a}.0|(b.\bar{c}.0))$, $b.a.0$, and $(\nu b)!(c.0 + c.c.0)$. However, the empty process is not usually written since it is clear what is meant, so $b.a.0$ is written as $b.a$. The operational semantics, given in Fig. 2–1, explain the meanings of the operators. The operational semantics define the *transition relation*, which is of the form $P \xrightarrow{\alpha} Q$ where P and Q are processes and α is an action, co-action, or silent action.

$$\text{ACT}\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{PLUS1}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{PAR1}\ \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

$$\text{SYN}\ \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \text{RES}\ \frac{P \xrightarrow{\alpha} P' \quad \alpha \neq a, \bar{a}}{(\nu a)P \xrightarrow{\alpha} (\nu a)P'} \qquad \text{BANG}\ \frac{P|!P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

Figure 2–1: Operational Semantics

The transitions for the processes can be thought of as sending and receiving on channels: an a action represents receiving an input on channel a, and a $\bar{a}$ action represents broadcasting on channel a. The ACT rule is the simplest: the process $a.P$ must receive an input on channel a, then can act as process P. The process $P + Q$ can be thought of as a choice between processes P and Q: the PLUS1 rule says that process $P + Q$ can behave as process P, discarding Q. There is a symmetric rule PLUS2 saying that $P + Q$ can instead behave as Q and discard P. I have omitted this rule to save space. $P|Q$ means that both P and Q exist at the same time; the PAR1 rule says that $P|Q$ can do any actions that P can do, but in contrast to $P + Q$, when $P|Q$ executes an action from P, it keeps its ability to later execute an action from Q. There is a symmetric rule PAR2. SYN can be thought of as synchronization by sending and receiving: two processes are in parallel and one outputs on a certain channel and the other one receives the input on the channel, so both processes change, but the exchange is invisible to the outside environment, so only a silent τ action is seen. The ν operator restricts a certain channel: $(\nu a)P$ behaves as P, as long as it doesn't receive or broadcast on the a channel. Note that this does not prevent synchronization on channel a: $(\nu a)(a|\bar{a})$ can do a τ transition. Finally, $!P$ is replication. Informally, $!P$ behaves as $P|P|P...$, with arbitrarily many instances of the process in parallel.

The operational semantics only gives a single transition at a time for a process. Often, the transition relation is extended to multiple actions: if $P_1 \xrightarrow{\alpha_1} P_2 \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_n} P_{n+1}$, then we say that $P \xrightarrow{\alpha_1.\alpha_2...\alpha_n} P_{n+1}$. So $a.b \xrightarrow{ab} 0$. Also, if process P has no transitions available according to the operational semantics, then P is said to be blocked, which is written $P \not\rightarrow$. Of course, a process may have more than one transition available, and may even have different transitions available with the same action: $a.c + (a|b) \xrightarrow{a} c$ and $a.c + (a|b) \xrightarrow{a} 0|b$.

2.2 Bisimulation

Bisimulation is a widely used process equivalence and is in fact the finest equivalence commonly discussed [Par81]. It is important for security applications – in fact, for almost all applications – that the notion of equivalence be compositional: if two processes are equivalent and are put in the same context, the resulting processes will still be equivalent. Bisimulation for CCS was proven to be compositional [Mil89][1] .

It is easy to see that there are processes which are the same in every relevant aspect but are written differently. For example, $P + Q$ and $Q + P$ are clearly essentially equivalent, but they are not equal. Similarly, we see that $P + P$, $0|P$ and P behaves exactly the same way, and 0 and $(\nu a)(a)$ can be considered equivalent because neither of them can perform any transitions. Bisimulation formalizes all of these notions.

Definition 2.2.1 *P and Q are bisimilar, denoted $P \sim Q$, if for all a and for all P' such that $P \xrightarrow{a} P'$, there is a process Q' such that $Q \xrightarrow{a} Q'$ and*

[1] Actually, this result holds only for so-called strong bisimulation, which is what we discuss here.

$P' \sim Q'$, and for all a and for all Q' such that $Q \xrightarrow{a} Q'$, there is a process Q' such that $P \xrightarrow{a} P'$ and $Q' \sim P'$.

Thus, we say that two processes are bisimilar if each of them can "match" any transition that the other process can do. This idea is intuitively reasonable, but the definition appears circular at first glance. It can in fact be formalized as a coinductive definition, and I present two equivalent ways of doing this here. First we define *a* bisimulation relation.

Definition 2.2.2 *A bisimulation relation R is an equivalence relation on processes such that if PRQ and $P \xrightarrow{a} P'$ for any a then there exists a Q' such that $Q \xrightarrow{a} Q'$ and $P'RQ'$ and symmetrically with P and Q interchanged.*

Note that being a bisimulation relation is a property of R and is not circular. Now we can define P to be bisimilar to Q if we can find a bisimulation relation R such that PRQ.

The second way to formalize bisimulation is by using fixed points. Recall that any monotone function on a complete lattice has a greatest fixed point. We define a function $\mathcal{F}$ on the lattice $\mathcal{R}$ of equivalence relations (on processes) ordered by inclusion as follows. Given a relation R on processes, we define $P\mathcal{F}(R)Q$ to hold if whenever $P \xrightarrow{a} P'$ there exists Q' such that $Q \xrightarrow{a} Q'$ with $P'RQ'$, and symmetrically with P and Q interchanged. It is easy to verify that $\mathcal{F}$ is a monotone function and that $\mathcal{R}$ is a complete lattice. We define bisimulation to be the greatest fixed point of $\mathcal{F}$.

It is not hard to see that these definitions coincide.

2.3 Hennessy-Milner Logic

Hennessy-Milner logic is a special modal logic that characterizes bisimulation. This means that two processes are bisimilar if and only if they satisfy the same formulas in Hennessy-Milner logic. My presentation of this logic is based on [HM85].

2.3.1 Syntax and Semantics

Hennessy-Milner logic has two special operators: a box and a diamond, each labelled with an action.

$$\phi ::= \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [\alpha]\phi \mid \langle\alpha\rangle\phi \mid \top \mid \bot.$$

The semantics formalize the meanings of these operators. Essentially, $\langle\alpha\rangle\phi$ means that there is some transition α such that ϕ holds after α, and $[\alpha]\phi$ means that after any α transition, ϕ will be true.

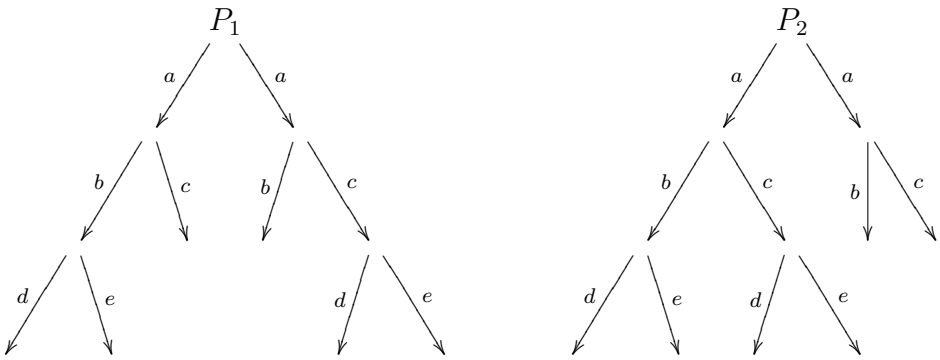
- $P \models \phi_1 \wedge \phi_2$ if $P \models \phi_1$ and $P \models \phi_2$.
- $P \models \phi_1 \vee \phi_2$ if $P \models \phi_1$ or $P \models \phi_2$.
- $P \models [\alpha]\phi$ if for all P' such that $P \xrightarrow{\alpha} P'$, $P' \models \phi$.
- $P \models \langle\alpha\rangle\phi$ if there is some P' such that $P \xrightarrow{\alpha} P'$ and $P' \models \phi$.
- $P \models \top$ for any P.
- no process satisfies $\bot$.

Note that $P \models \langle a\rangle\top$ means that P has some a transition available, and $P \models [a]\bot$ means that P has no a transitions.

Hennessy-Milner logic is useful because it characterizes bisimulation. From the definition of bisimulation, it may not be obvious how to show that two

processes are not bisimilar, but it is often easy to find a Hennessy-Milner formula that differentiates between them, as in the following example.

Example 2.3.1 *Consider the processes* $P_1 = a.(b.(d+e)+c)+a.(b+c.(d+e))$ *and* $P_2 = a.(b.(d+e)+c.(d+e))+a.(b+c)$.



It is somewhat difficult to use the definition of bisimulation to decide whether P_1 *and* P_2 *are bisimilar, but it is easy to see that* $P_2 \models \langle a\rangle((\langle b\rangle\langle d\rangle\top) \wedge (\langle c\rangle\langle d\rangle\top))$, *while* P_2 *does not satisfy this formula.*

CHAPTER 3
Labelled Process Calculus with Independence Operator

We now introduce our process calculus with labelled actions and a protection operator. This process calculus was first introduced in [CPP06]. The purpose is to model independent agents interacting with limited information. The labels on actions allow us to control what is visible to an agent about an action; if two actions have the same label then they are indistinguishable to an agent controlling the execution of the process. The protection operator, represented by curly brackets, indicates that the choice of the top-level action in the protected subprocess must be made independently from the choices concerning unprotected actions in the process. This idea is explained in more detail below.

We let l, j, and k represent labels, a and b actions, $\bar{a}$ and $\bar{b}$ co-actions, τ the silent action, and α and β generic actions, co-actions, or silent action. The syntax for a process is as follows:

$$P, Q ::= l : \alpha.P \mid P|Q \mid P + Q \mid (\nu a)P \mid l : \{P\} \mid 0$$

The operational semantics for this process calculus is shown in Fig. 3–1. The transition relation in the operational semantics includes both the action and the label for the action. In the case of synchronization, the labels for both synchronizing actions are included in the transition, and for the SWITCH rule, two labels are also included, one representing the fact that the protected process was chosen and one representing the action taken within the protected process. All the labels have an X or Y subscripted to

them, denoting whether the label was part of a protected choice (Y) or not (X). There are corresponding right rules for $+$ and $|$; these operators are both associative and commutative. All of the rules are analogous to those of traditional process algebra, except for the rule SWITCH, which requires that protected processes do a silent action. The reason for this restriction on the SWITCH operator is that this operator is intended to represent choices made independently from the other choices in the process. For example in the process $(l_1 : a \ + \ l_2 : b) \mid l_3 : \{k_1 : \tau \,.\, l_4 : a \ + \ k_2 : \tau \,.\, l_4 : b\}$, the left and right choices are represented as independent. This means that whatever agent controls whether the left part of the process performs an a or b action does not control how the choice on the right side of the process is resolved. This choice is resolved by an entity independent from the traditional scheduler. Therefore, we require that the protected subprocess do a silent action, because any other action would be observable to the outside world, and therefore observable to the scheduler, allowing it to base its decisions on the outcome of the protected choice, which would make this choice dependent on other choices. This independence is not a part of the operational semantics; rather, it represents the idea that the protected subprocess makes decisions independently from the main process. Furthermore, requiring the protected subprocess to do a silent action prevents synchronization between protected and unprotected parts of the process, since these two parts of the process should be independent.

From now on, we will only consider *deterministically labelled* processes: processes where there can never be more than one action available with the same label.

$$\text{ACT}\ \frac{}{l : \alpha \,.\, P \xrightarrow[l_X]{\alpha} P} \qquad \text{RES}\ \frac{P \xrightarrow[s]{\alpha} P' \quad \alpha \neq a, \bar{a}}{(\nu a)P \xrightarrow[s]{\alpha} (\nu a)P'} \qquad \text{SUM1}\ \frac{P \xrightarrow[s]{\alpha} P'}{P + Q \xrightarrow[s]{\alpha} P'}$$

$$\text{PAR1}\ \frac{P \xrightarrow[s]{\alpha} P'}{P|Q \xrightarrow[s]{\alpha} P'|Q} \qquad \text{COM}\ \frac{P \xrightarrow[l_X]{a} P' \quad Q \xrightarrow[j_X]{\bar{a}} Q'}{P|Q \xrightarrow[(l,j)_X]{\tau} P'|Q'} \qquad \text{SWITCH}\ \frac{P \xrightarrow[j_X]{\tau} P'}{l : \{P\} \xrightarrow[l_X . j_Y]{\tau} P'}$$

Figure 3–1: Operational semantics

Definition 3.0.2 *P is* deterministically labelled *if the following conditions hold:*

1. *It is impossible for P to make two different transitions with the same labels: for all strings s, if $P \xrightarrow[s]{\alpha} P'$ and $P \xrightarrow[s]{\beta} P''$ then $\alpha = \beta$ and $P' = P''$.*
2. *If $P \xrightarrow[l_X . j_Y]{\tau} P'$ then there is no transition $P \xrightarrow[l_X]{\alpha} P''$ for any α or P''.*
3. *If $P \xrightarrow[s]{\alpha} P'$ then P' is deterministically labelled.*

Note that any blocked[1] process is deterministically labelled, so since we only consider finite processes without recursion, this concept is well defined.

Roughly, this means that two enabled actions never have the same label. For example, $P = l : a + l : b$ is not deterministically labelled because $P \xrightarrow[l_X]{a} 0$ and $P \xrightarrow[l_X]{b} 0$ but $a \neq b$, violating the first condition. Also, $P = l_1 : a \ + \ l_1 : \{l_2 : \tau\}$ is not deterministically labelled since $P \xrightarrow[l_{1X} . l_{2Y}]{\tau} 0$ and $P \xrightarrow[l_1]{a} 0$, violating the second condition. Further, no process with this as a (reachable) subprocess is deterministically labelled. However, $l_1 : a \,.\, l_3 : b \ + \ l_2 : c \,.\, l_3 : d$ is deterministically labelled even though l_3 occurs

[1] A process is blocked if it cannot make any transition.

twice, since there is no series of transitions that will result in both l_3's being available simultaneously.

Also, $l_1 : a \mid l_2 : b \,.\, l_1 : c$ is not deterministically labelled because it can transition to $l_1 : a \mid l_1 : c$ which is not deterministically labelled.

Note, however, that $l : a \,.\, P + l : a \,.\, P$ is deterministically labelled. Even though l is available twice, $l : a \,.\, P + l : a \,.\, P \xrightarrow[l]{a} P$ is the only transition available labelled with l, so P is deterministically labelled.

CHAPTER 4
Games and Strategies

In this chapter we define two player games on deterministically labelled processes. One game is defined for each deterministically labelled process. The two players are called X and Y. The moves in the game are labels and pairs of labels. Moves represent an action being taken by the process. The player X controls all the unprotected actions, and the player Y is in charge of all the top level actions within the protected subprocesses. This makes it possible to represent the independent resolution of the two kinds of choice, by carefully defining the appropriate strategies for these games. A strategy is for one player and determines the moves the player will choose within the game. Games and strategies are both made up of *valid positions*, discussed in the next section.

4.1 Valid Positions

Valid positions are defined on a process and represent valid plays or executions for that process, with player X moving first. Every valid position is a string of moves (labels or pairs of labels from the process), each of which is assigned to a player X or Y, with player X moving first. The set of all valid positions for a process represents all possible executions of the process, including partial, unfinished executions.

Definition 4.1.1 *A* move *is anything of the form* l_X, l_Y, $(l,j)_X$, *or* $(l,j)_Y$ *where* l, *and* j *are labels.* l_X *and* $(l,j)_X$ *are called* X-moves *and* l_Y *and* $(l,j)_Y$ *are called* Y-moves

To define valid positions, we must define an extension of the transition relation.

Definition 4.1.2 *This extends the transition relation to multiple transitions, ignoring the actions for the transitions but keeping track of the labels.*

1. *For any process* P, $P \underset{\varepsilon}{\longrightarrow} P$.
2. *If* $P \underset{s}{\xrightarrow{\alpha}} P'$ *and* $P' \underset{s'}{\longrightarrow} P''$ *then* $P \underset{s.s'}{\longrightarrow} P''$.

Now we define valid positions.

Definition 4.1.3 *If* $P \underset{s}{\longrightarrow} P'$ *then every prefix of* s *(including* s*) is a* valid position *for* P.

In order for the set of valid positions to be prefix closed, we must explicitly include prefixes in the definition because of the SWITCH rule. For example, for the process $l : \{j : \tau\}$, the set of valid positions is $\{\varepsilon,\ l_X,\ l_X.j_y\}$, but if the condition about prefixes were not included in the definition of valid positions, l_X would not be a valid position, because the process does not have any transition with this label alone.

Example 4.1.4 *Consider the process*

$$P = (\nu b)\left(l_1 : \{k_1 : \tau \,.\, l_2 : a \,.\, l_3 : b + k_2 : \tau \,.\, l_2 : c \,.\, l_3 : b\} \mid l_4 : \bar{b} \,.\, (l_5 : d + l_6 : e)\right).$$

Here are some of the valid positions for P*:*

$$l_{1X}.k_{1Y}.l_{2X}.(l_3, l_4)_X.l_{5X}$$

$$l_{1X}.k_{1Y}.l_{2X}.(l_3, l_4)_X.l_{6X}$$

$$l_{1X}.k_{2Y}.l_{2X}.(l_3, l_4)_X.l_{5X}$$

$$l_{1X}.k_{2Y}.l_{2X}.(l_3, l_4)_X.l_{6X}$$

The prefixes of these valid positions are also valid positions.

4.2 Strategies

A strategy for a certain player is a subset of the valid positions, each valid position ending with a move made by that player. The idea behind a strategy is that if, for example, player X finds himself in position s and $s.m$ is in his strategy, then he will do move m.[1]

From now on, when we use m without a subscript to denote a move, it will mean a move including its player: a move of the form l_X, $(l_1, l_2)_X$, l_Y, or $(l_1, l_2)_Y$. When we use m_X, m_Y, or m_Z to denote a move, it means a move with the specified subscript, where Z represents X or Y.

Definition 4.2.1 *Let Z stand for either X or Y. In the game for P, a* strategy for Z *is a set S of valid positions such that ε is in S and if $s.m \in S$, then m is a Z move ($m = l_Z$ for some label or pair of labels l), and every prefix of s ending with a Z move is in S.*

Example 4.2.2 *For*

$$P = (\nu b)\left(l_1 : \{k_1 : \tau \,.\, l_2 : a \,.\, l_3 : b + k_2 : \tau \,.\, l_2 : c \,.\, l_3 : b\} \mid l_4 : \bar{b} \,.\, (l_5 : d + l_6 : e)\right),$$

[1] Unlike the usual game theoretic definition of strategy, our strategies are not functional, that is, they do not necessarily specify one move for the player to make in every situation. We can, however, put simple conditions on our strategies so that they are functional.

one strategy for X is:

$$\varepsilon$$
$$l_{1X}$$
$$l_{1X}.k_{2Y}.l_{2X}$$
$$l_{1X}.k_{2Y}.l_{2X}.(l_3, l_4)_X$$
$$l_{1X}.k_{2Y}.l_{2X}.(l_3, l_4)_X.l_{6X}$$

Another strategy for X is:

$$\varepsilon$$
$$l_{1X}$$
$$l_{1X}.k_{1Y}.l_{2X}$$
$$l_{1X}.k_{2Y}.l_{2X}$$

One strategy for Y is:

$$\varepsilon$$
$$l_{1X}.k_{1Y}$$
$$l_{1X}.k_{2Y}$$

This is not a strategy:

$$\varepsilon$$
$$l_{1X}$$
$$l_{1X}.k_{2Y}.l_{2X}.(l_3, l_4)_X$$

It is not a strategy because it contains $l_{1X}.k_{2Y}.l_{2X}.(l_3, l_4)_X$, which is a valid position ending in an X move, but does not contain $l_{1X}.k_{2Y}.l_{2X}$, a prefix of the aforementioned valid position ending in an X move, so it would have to be included for this set to be a strategy.

4.3 Execution of Processes According to Strategies

In this section we define the execution of a process with two strategies- one for each player. However, not every pair of strategies defines a unique

execution of a process. We define two simple restrictions on strategies, which together imply that executions are unique.

Definition 4.3.1 *A strategy S is* deterministic *if: $s.m_1 \in S, s.m_2 \in S$ implies $m_1 = m_2$.*

Example 4.3.2 *For*

$$P = (\nu b)\left(l_1 : \{k_1 : \tau \,.\, l_2 : a \,.\, l_3 : b + k_2 : \tau \,.\, l_2 : c \,.\, l_3 : b\} \mid l_4 : \bar{b} \,.\, (l_5 : d + l_6 : e)\right),$$

the above strategy for X,

$$\begin{aligned} &\varepsilon, \\ &l_{1X}, \\ &l_{1X}.k_{2Y}.l_{2X}, \\ &l_{1X}.k_{2Y}.l_{2X}.(l_3, l_4)_X, \\ &l_{1X}.k_{2Y}.l_{2X}.(l_3, l_4)_X.l_{6X} \end{aligned}$$

is deterministic. The strategy for Y, $\{\varepsilon,\ l_{1X}.k_{1Y},\ l_{1X}.k_{2Y}\}$. is not deterministic, because $l_{1X}.k_{1Y}$ and $l_{1X}.k_{2Y}$ are both included in the strategy.

The second restriction is called completeness; it means that a strategy prescribes a move for the player whenever the player has a move available. In order to define completeness, we start with two subsidiary definitions.

Definition 4.3.3 *Let V denote the set of valid positions for a process P. If s is a valid position for P, enabled(s) represents the set of moves available after s: define enabled$(s) = \{m | s.m \in V\}$. Also, define the X and Y moves available after s as, respectively, enabled$_X(s) = \{m_X | s.m_X \in V\}$ and enabled$_Y(s) = \{m_Y | s.m_Y \in V\}$.*

Note that a position can have X moves enabled or Y moves enabled, but not both. This is clear from the operational semantics.

Definition 4.3.4 *If s is a valid position for P and Z is a player, let $\bar{Z}$ denote the other player. We define $Z(s)$, the string of Z moves in s, inductively as follows:*

1. $Z(\varepsilon) = \varepsilon$.
2. $Z(s.m_Z) = Z(s).m_Z$.
3. $Z(s.m_{\bar{Z}}) = Z(s)$.

Example 4.3.5 *Continuing the example from above, $X(l_{1X}.k_{1Y}.l_{2X}.(l_3, l_4)_X.l_{5X}) = l_{1X}.l_{2X}.(l_3, l_4)_X.l_{5X}$, and $Y(l_{1X}.k_{1Y}.l_{2X}.(l_3, l_4)_X.l_{5X}) = k_{1Y}$.*

Definition 4.3.6 *For a nonblocked process with valid positions V, a strategy S for player Z is* complete *if for all $s \in S$, for every string s' such that $Z(s') = \varepsilon$ and $s.s' \in V$ and $enabled_Z(s.s') \neq \emptyset$, then $s.s'.m \in S$ for some move m.*

Completeness captures the idea that a player's strategy always dictates a move whenever it is that player's turn to play and a move is available. Note that if a deterministic strategy chooses a move m_1 at a particular point and another move m_2 is available to it, there is no need for a complete strategy to specify what happens after an m_2 move, since this move will not be chosen. The condition $Z(s') = \varepsilon$ means that all the moves in s' were opponent moves, so that a complete strategy can respond to any sequence of moves made by the opponent. On the other hand, we of course do not want to quantify over moves made by the strategy's own player.

Example 4.3.7 *The strategy S given above for X,*

$$\varepsilon$$
$$l_{1X}$$
$$l_{1X}.k_{2Y}.l_{2X}$$
$$l_{1X}.k_{2Y}.l_{2X}.(l_3, l_4)_X$$
$$l_{1X}.k_{2Y}.l_{2X}.(l_3, l_4)_X.l_{6X}$$

is not complete, because X cannot respond to Y choosing k_1: $l_{1X} \in S$, and $l_{1X}.k_{1Y} \in V$ and $enabled_X(l_{1X}.k_{1Y}) \neq \emptyset$, but there is no move m such that $l_{1X}.k_{1Y}.m \in S$. The strategy would be complete if, for example, the valid position $l_{1X}.k_{1Y}.l_{2X}.(l_3, l_4)_X.l_{5X}$ and all appropriate prefixes were added to the strategy.

Now we define the execution of a process with deterministic, complete strategies as the execution corresponding to the maximal execution that the two strategies "agree" on.

Definition 4.3.8 *Define the execution of a process P with deterministic, complete X and Y strategies S_1 and S_2 as follows: Let s be the maximal element in $S = \{s \in S_1 \cup S_2 \mid$ every prefix of s is in $S_1 \cup S_2\}$. The execution of P according to S_1 and S_2 is the sequence of processes $P, P_1 ... P_n$ such that $s = s_1 s_2 ... s_n$ where each s_i is either a single X move or an X move followed by a Y move, and*

$$P \xrightarrow[s_1]{} P_1 \xrightarrow[s_2]{} P_2 \xrightarrow[s_3]{} ... \xrightarrow[s_{n-1}]{} P_{n-1} \xrightarrow[s_n]{} P_n.$$

This represents the sequence of moves that will be chosen and processes that will be reached if labels are chosen according to the strategies S_1 and S_2.

In order to check that the string s in this definition exists and is unique, we need the following proposition.

Proposition 4.3.9 *Consider a process P, S_1 a deterministic, complete X strategy for P, and S_2 a deterministic, complete Y strategy for P. Let*

$$S = \{s \in S_1 \cup S_2 \mid \textit{every prefix of } s \textit{ is in } S_1 \cup S_2\}$$

Then the prefix ordering is a total order on S.

Proof . First, note that since every strategy for either player contains ε, S always contains ε.

Now we use induction on the length of the valid positions in S to prove that S has at most one element of any length: if $s.m \in S$ and the length of $s.m$ is $n+1$, then $s \in S$ since S is clearly prefix closed, so by assumption, s is the only element of length n in S, and any string of length $n+1$ must be of the form $s.m'$, again since S is prefix closed. Since a position cannot have both X and Y moves enabled, either $s.m, s.m' \in S_1$, or $s.m, s.m' \in S_2$. In either case, $m = m'$ by the determinacy of that strategy.

Therefore, every string s in S is a prefix of all longer strings in S, and every shorter string in S is a prefix of s, so the prefix order is a total order on S. ∎

Since the prefix order is a total order on S in definition 4.3.8 and S is a finite set, there is indeed a unique maximal element s.

4.4 Epistemic Restrictions on Strategies

Now that we have shown how properly specified strategies determine the execution of a process, we can consider epistemic restrictions on strategies, representing agents' actions when their knowledge is limited. In general, we impose epistemic conditions on strategies first by determining what knowledge is appropriate for each agent, that is, which sets of executions should be indistinguishable for him. Once the correct notion of the agent's

knowledge is determined, the condition on the strategy to enforce this knowledge is "if valid positions s_1 and s_2 are indistinguishable for player Z (Z's knowledge about s_1 and s_2 is the same), then for any move m, $s_1.m$ is in the strategy if and only if $s_2.m$ is in the strategy." In other words, Z must choose the same move whether s_1 or s_2 describes the execution of the process so far, because he does not know whether s_1 or s_2 has occurred. We call restrictions of this form *epistemic restrictions*.

For example, we could require that an agent only have knowledge of his own past moves. The epistemic restriction for a strategy S to satisfy this property is: if $Z(s_1) = Z(s_2)$, then for all moves m, $s_1.m \in S$ if and only if $s_2.m \in S$. Thus, this player responds the same way no matter what the other player does, because he does not have knowledge of the other player's actions. Similarly, we could require that an agent only know what moves are currently available to him. The epistemic restriction expressing this for a strategy S is: if $enabled_Z(s_1) = enabled_Z(s_2)$ then $s_1.m \in S$ if and only if $s_2.m \in S$.

We now single out a very important epistemic restriction, called introspection. An introspective strategy allows a player to "remember" not only his own history of moves, but also the moves that were available to him at every point in the past, including the current step. Introspective strategies are important because they exactly capture the intended independence requirement for the protection operator.

Definition 4.4.1 *For player Z, positions s_1 and s_2 are called Z* indistinguishable *if they satisfy the following conditions:*

1. $Z(s_1) = Z(s_2)$
2. $enabled_Z(s_1) = enabled_Z(s_2)$.

3. *For all prefixes s_1' of s_1 and s_2' of s_2, if Z has a move available at both s_1' and s_2' and $Z(s_1') = Z(s_2')$, then $enabled(s_1') = enabled(s_2')$.*

In this definition, two positions are indistinguishable if the player made the same series of moves to arrive at both positions, and at any point in the past where he had made a certain series of moves in both positions and had moves available, he had the same set of moves available in both positions.

Definition 4.4.2 *Given a process P, and S a strategy for player Z on P, S is* introspective *if for every Z indistinguishable pair of valid positions s_1 and s_2, $s_1.m \in S$ if and only if $s_2.m \in S$.*

In other words, the player chooses the move he makes at each step based on his past moves, the moves that are available to him, and the moves that were available to him at each point in the past. If these conditions are all the same at two positions, the player cannot distinguish them, so he makes the same move at both positions.

Example 4.4.3 *For*

$$P = (\nu b)\left(l_1 : \{k_1 : \tau \,.\, l_2 : a \,.\, l_3 : b + k_2 : \tau \,.\, l_2 : c \,.\, l_3 : b\} \mid l_4 : \bar{b} \,.\, (l_5 : d + l_6 : e)\right)$$

the deterministic strategy given above for X,

$$\begin{aligned} S \;=\; \{&\varepsilon,\\ &l_{1X},\\ &l_{1X}.k_{1Y}.l_{2Y},\\ &l_{1X}.k_{2Y}.l_{2X},\\ &l_{1X}.k_{1Y}.l_{2X}.(l_3,l_4)_X,\\ &l_{1X}.k_{2Y}.l_{2X}.(l_3,l_4)_X,\\ &l_{1X}.k_{1Y}.l_{2X}.(l_3,l_4)_X.l_{5X},\\ &l_{1X}.k_{2Y}.l_{2X}.(l_3,l_4)_X.l_{6X}\} \end{aligned}$$

is not introspective. This is because in order to satisfy the introspection condition, $l_{1X}.k_{1Y}.l_{2X}.(l_3,l_4)_X$ and $l_{1X}.k_{2Y}.l_{2X}.(l_3,l_4)_X$ should have the same moves appended to them in S, since they are X indistinguishable. However, $l_{1X}.k_{1Y}.l_{2X}.(l_3,l_4)_X.l_{5X} \in S$ and $l_{1X}.k_{2Y}.l_{2X}.(l_3,l_4)_X.l_{5X} \notin S$, and similarly, $l_{1X}.k_{2Y}.l_{2X}.(l_3,l_4)_X.l_{6X} \in S$ and $l_{1X}.k_{2Y}.l_{2X}.(l_3,l_4)_X.l_{5X} \notin S$.

An example of an introspective strategy for X is the set consisting of the strings $l_{1X}.k_{1Y}.l_{2X}.(l_3,l_4)_X.l_{5X}$ and $l_{1X}.k_{2Y}.l_{2X}.(l_3,l_4)_X.l_{5X}$ as well as the appropriate prefixes of either of these two strings. Here is an example showing why the prefixes of the valid positions are discussed in the definition of introspective. For readability, labels are replaced with superscript numbers preceding actions: ${}^1a.P$ represents $l_1 : a \,.\, P$.

Example 4.4.4 *Consider*

$$P \;=\; {}^0\{\; {}^1\tau \,.(\, {}^3c \,.(\, {}^6f + {}^7g\,) + {}^4d\,) + {}^2\tau \,.(\, {}^3c \,.(\, {}^6f + {}^7g\,) + {}^5e\,)\}.$$

Let X's strategy be

$$\begin{aligned} S \;=\; \{&\varepsilon,\\ &l_{0X},\\ &l_{0X}.l_{1Y}.l_{3X},\\ &l_{0X}.l_{2Y}.l_{3X},\\ &l_{0X}.l_{1Y}.l_{3X}.l_{6X},\\ &l_{0X}.l_{2Y}.l_{3X}.l_{7X}. \end{aligned}$$

This strategy is introspective. Even though $X(l_{0X}.l_{1Y}.l_{3X}) = X(l_{0X}.l_{2Y}.l_{3X})$ *and*
$enabled_X(l_{0X}.l_{1Y}.l_{3X}) = enabled_X(l_{0X}.l_{2Y}.l_{3X})$*, it is acceptable that the two strings have different moves appended to them, because* $enabled_X(l_{0X}.l_{1Y}) = \{l_{3X}, l_{4X}\}$ *and* $enabled_X(l_{0X}.l_{2Y}) = \{l_{3X}, l_{5X}\}$*. This can be thought of as X being able to distinguish between the two positions* $l_{0X}.l_{1Y}.l_{3X}$ *and* $l_{0X}.l_{2Y}.l_{3X}$ *because he remembers what moves were available to him earlier and is able to use this information to tell apart the two positions.*

The essence of the introspection condition is that a player knows what moves it has made in the past and knows what moves, if any, were available to it at each point in the past, but cannot see any moves that its opponent has made. Thus, each player must choose its moves based solely on its own past moves, the past moves that were available to it, and the moves available to it now.

CHAPTER 5
Correspondence between Strategies and Schedulers

In this chapter, I begin by reviewing the syntactic schedulers defined in [CP07], then I prove that deterministic complete introspective strategies correspond exactly to these schedulers. This result is important because these schedulers are defined purely syntactically, without any explicit reference to knowledge or equivalence between executions. Since the players' knowledge is explicit in the definition of introspective strategies, this equivalence explains the knowledge requirements underlying the syntactic schedulers, which had not been discussed before.

5.1 Background on Schedulers

The process calculus with schedulers uses the syntax for processes discussed above, with the protection operator, but also adds a new ingredient: explicit syntax for a pair of independent schedulers. The schedulers use labels, rather than actions, to interact with a process, making it possible to use labels to control a scheduler's "view" of a process. The schedulers choose a sequence of labels, to execute actions, or pairs of labels, to synchronize processes, and also can check whether a label or synchronization is available, using an if... then... else... construct. The two schedulers operate independently and do not communicate with one another, and each scheduler controls certain choices in the process. This makes it possible to represent independent choices in the process calculus. A *complete process* is an ordinary process augmented with a pair of schedulers. In this section, we also add the notion of *general labels*, either a single ordinary label or a pair of

ordinary labels. This convention is useful because an ordinary label and a pair of synchronizing ordinary labels both represent a single action by a scheduler. We let l and k represent ordinary labels and L and K represent general labels. The notations $\sigma(L)$, $\sigma(l)$, and $\sigma(l,k)$ are used to designate a choice made by a scheduler: $\sigma(l)$ means a single action will be executed, $\sigma(l,k)$ means that the scheduler will synchronize two actions, and $\sigma(L)$ can represent either of these cases. We let a and b represent actions, $\bar{a}$ and $\bar{b}$ co-actions, τ the silent action, α and β generic actions, co-actions, or silent action, P and Q processes, and ρ and η schedulers. The syntax for a complete process is as follows:

$$\begin{aligned} P,Q &::= l:\alpha.P \mid P|Q \mid P+Q \mid (\nu a)P \mid l:\{P\} \mid 0 \\ L &::= l \mid (l,k) \\ \rho,\eta &::= \sigma(L).\rho \mid \textbf{if } L \textbf{ then } \rho \textbf{ else } \eta \mid 0 \\ CP &::= P \parallel \rho,\eta \end{aligned}$$

The first scheduler is called the primary scheduler and the second scheduler is the secondary scheduler. The rules for the operational semantics of the process calculus with schedulers are in Fig. 5–1. Using the **if then else** construct (rules IF1, IF2), the scheduler can check whether a move is available and choose what to do based on that information. The SWITCH rule says that the curly brackets indicate a point where the secondary scheduler makes the next choice. After making this choice, control reverts to the primary scheduler. The choice made by the secondary scheduler must result in a τ observation because the process is encapsulated and cannot interact with the environment at this point. Of course, once control reverts to the primary scheduler, interactions with the external environment can indeed take place. The order in which the schedulers are written indicates

$$\text{ACT}\ \frac{}{l : \alpha.P \parallel \sigma(l).\rho, \eta \xrightarrow[l_X]{\alpha} P \parallel \rho, \eta}$$

$$\text{RES}\ \frac{P \parallel \rho, \eta \xrightarrow[s]{\alpha} P' \parallel \rho', \eta' \quad \alpha \neq a, \bar{a}}{(\nu a)P \parallel \rho, \eta \xrightarrow[s]{\alpha} (\nu a)P' \parallel \rho', \eta'}$$

$$\text{SUM1}\ \frac{P \parallel \rho, \eta \xrightarrow[s]{\alpha} P' \parallel \rho', \eta' \qquad \rho \neq \textbf{if } L \textbf{ then } \rho_1 \textbf{ else } \rho_2}{P + Q \parallel \rho, \eta \xrightarrow[s]{\alpha} P' \parallel \rho', \eta'}$$

$$\text{PAR1}\ \frac{P \parallel \rho, \eta \xrightarrow[s]{\alpha} P' \parallel \rho', \eta' \qquad \rho \neq \textbf{if } L \textbf{ then } \rho_1 \textbf{ else } \rho_2}{P|Q \parallel \rho, \eta \xrightarrow[s]{\alpha} P'|Q \parallel \rho', \eta'}$$

$$\text{SWITCH}\ \frac{P \parallel \eta, 0 \xrightarrow[j_X]{\tau} P'||\eta', 0}{l : \{P\} \parallel \sigma(l).\rho, \eta \xrightarrow[l_X.j_Y]{\tau} P' \parallel \rho, \eta'}$$

$$\text{COM}\ \frac{P \parallel \sigma(l).0, 0 \xrightarrow[l_X]{a} P' \parallel 0, 0 \qquad Q \parallel \sigma(j).0, 0 \xrightarrow[j_X]{\bar{a}} Q' \parallel 0, 0}{P|Q \parallel \sigma(l, j).\rho, \eta \xrightarrow[(l,j)_X]{\tau} P'|Q' \parallel \rho, \eta}$$

$$\text{IF1}\ \frac{P \parallel \rho_1, \eta \xrightarrow[s]{\alpha} P' \parallel \rho_1', \eta' \quad P \parallel \sigma(L).0, \theta \xrightarrow[s']{\beta} P'' \parallel 0, \theta' \quad \text{for some scheduler } \theta}{P \parallel \textbf{if } L \textbf{ then } \rho_1 \textbf{ else } \rho_2, \eta \xrightarrow[s]{\alpha} P' \parallel \rho_1', \eta'}$$

$$\text{IF2}\ \frac{P \parallel \rho_2, \eta \xrightarrow[s]{\alpha} P' \parallel \rho_2', \eta' \quad P \parallel \sigma(L).0, \theta \not\rightarrow \quad \text{for all schedulers } \theta}{P \parallel \textbf{if } L \textbf{ then } \rho_1 \textbf{ else } \rho_2, \eta \xrightarrow[s]{\alpha} P' \parallel \rho_2', \eta'}$$

Figure 5–1: Operational semantics for processes with schedulers

which one is to be regarded as primary. In the rules SUM1 and PAR1, we require that the primary scheduler not be of the form **if** L **then** ρ_1 **else** ρ_2 because the **if then else** construct allows a scheduler to check whether a label is available. Thus, the behaviour of a process P with primary scheduler **if** L **then** ρ_1 **else** ρ_2 may be different than the behaviour of process $P + Q$ with the same scheduler if the label L is available in process Q. The same condition applies to PAR1. The rules IF1 and IF2 check whether a process

can execute any transition with the one step primary scheduler $\sigma(L)$ and any secondary scheduler. If there is any transition that can occur for this complete process, then the first branch of the primary scheduler is activated, otherwise, the second branch occurs.

Clearly, if a process is blocked, then no transition is possible with any schedulers. On the other hand, it is possible for a process that is not blocked to have no transitions available with certain schedulers. For example, the process $l : a$ is not blocked, but no transitions are available for the complete process $l : a \parallel \sigma(j), 0$. Thus, it is useful to define the notion of a pair of schedulers being nonblocking for a certain process.

Definition 5.1.1 *For a process P which is not blocked, a pair of schedulers ρ, η are inductively defined as* nonblocking *if $P \parallel \rho, \eta \xrightarrow{\alpha} P' \parallel \rho', \eta'$ for some α, P', ρ', and η', and if P is not blocked, then ρ' and η' are non-blocking for P'.*

Since we consider only finite processes, this inductive definition characterizes all nonblocking scheduler pairs for processes that are not blocked.

We have defined a nonblocking scheduler pair as, essentially, a pair of schedulers that choose a move for the process whenever one is available. Now we define the concept of a single scheduler being nonblocking. We would like to say that a single primary or secondary scheduler for a process is nonblocking if it can be paired with any nonblocking secondary or primary scheduler (respectively) for the process and not cause the process to be blocked. Obviously, this would be a circular definition, so we define nonblocking first inductively for a secondary scheduler, and then for a primary scheduler, with reference to nonblocking secondary schedulers.

Definition 5.1.2 *If P is a deterministically labelled process and is not blocked, then a scheduler η is a* nonblocking secondary *scheduler for P if for every general label L such that for some η_1,*

$$P \parallel \sigma(L), \eta_1 \xrightarrow[s]{\alpha} P' \parallel 0, \eta_1'$$

(for some α, s, P', and η_1'), then

$$P \parallel \sigma(L), \eta \xrightarrow[s']{\beta} P'' \parallel 0, \eta'$$

(for some β, s', P'' and η'), and if P'' is not blocked, η' is a nonblocking secondary scheduler for P''.

If P is blocked, then any secondary scheduler is defined to be nonblocking.

First, note that this is a complete inductive definition because we only consider finite processes, so any process will be blocked after some finite number of steps. The meaning of this definition is the following: if there is a label that can be chosen by the primary scheduler and execute an action in conjunction with some arbitrary secondary scheduler, then a nonblocking secondary scheduler must also be able to execute an action in conjunction with the primary scheduler that chooses this label.

For a blocked process, all schedulers are considered to be nonblocking because it is not the scheduler that is preventing an action from occurring, but the process itself.

Definition 5.1.3 *If P is a deterministically labelled process that is not blocked, then primary scheduler ρ is* primary nonblocking *if for any nonblocking secondary scheduler η,*

$$P \parallel \rho, \eta \xrightarrow[s]{\alpha} P' \parallel \rho', \eta'$$

(for some α, s, P', ρ', η') and if P' is not blocked, then ρ' is a nonblocking primary scheduler for P'.

In other words, a primary scheduler is one that will schedule an action for the process no matter what nonblocking secondary scheduler it is paired with.

5.2 Correspondence Theorem

The main correspondence theorem can now be stated.

Theorem 5.2.1 *Given a deterministically labelled process P, a nonblocking primary scheduler ρ for P, and a nonblocking secondary scheduler η for P, there is a deterministic, complete, introspective X strategy S depending only on P and ρ, and a deterministic, complete, introspective Y strategy T depending only on P and η, such that the execution of $P \parallel \rho, \eta$ is identical to the execution of P with S and T.*

Furthermore, given a deterministically labelled process P, a deterministic, complete, introspective X strategy S for P, and a deterministic, complete, introspective Y strategy T for P, there is a nonblocking primary scheduler ρ depending only on S and P and a nonblocking secondary scheduler η depending only on T and P such that the execution of P with S and T is identical to the execution of $P \parallel \rho, \eta$.

Before we discuss the proof we make some observations on the quantifier structure of the statement of the theorem. One could imagine stating the first part as follows:

$$\forall P, \rho \exists S \text{ s.t. } \forall \eta \exists T \ldots$$

This is *apparently* stronger and certainly clearer than the original version which uses the clumsy phrase "depending only on..." However, this is not

the case; it is actually weaker. The "new improved" version allows T to depend on ρ, which the version stated in the theorem does not allow. There is in fact a formal logic called "Independence Friendly" (IF) logic which allows quantifiers to be introduced with independence statements; this is just what the version in the statement of the theorem does, without, of course, dragging in all the formal apparatus of IF logic. In fact, it can be proved that there are statements of IF logic than cannot be rendered in ordinary first-order logic; the statement of the theorem is an example.

In order to prove the theorem we need these two definitions.

Definition 5.2.2 *Given a process P, a valid position s and a strategy S, define $S/s = \{s' | s.s' \in S\}$.*

Definition 5.2.3 *A move l in process P is called a* switch *move if it chooses a label of the form $l : \{P'\}$ in P. Otherwise, it is called an* ordinary *move.*

Proof . The first step is to construct a strategy from a scheduler. Consider a deterministically labelled process P and let the scheduler be ρ. Z stands for either player X or Y. Let V be the set of valid positions for P. $Strat(\rho, V)$ gives the corresponding strategy. It is defined inductively.

Case 1 : $Strat(\sigma(m).\rho', V) = \{\varepsilon\} \cup \{s \in V | s = s_1.m.s_2, Z(s_1) = \varepsilon, \text{ and } s_2 \in Strat(\rho', V/s_1.m)\}$.

Case 2 : $Strat(\textbf{if } l \textbf{ then } \rho_1 \textbf{ else } \rho_2, V) = \{\varepsilon\} \cup \{s \in V | s = s_1.s_2, Z(s_1) = \varepsilon, l \in enabled_Z(s_1), \text{ and } s_2 \in Strat(\rho_1, V/s_1)\} \cup \{s \in V | s = s_1.s_2, Z(s_1) = \varepsilon, l \notin enabled(s_1), enabled_Z(s_1) \neq \emptyset, \text{ and } s_2 \in Strat(\rho_2, V/s_1)\}$.

Case 3 : $Strat(0, V) = \{\varepsilon\}$.

It is easy to see that this is correct by induction on the structure of the scheduler. The first case is correct because it chooses every instance of the move specified by the scheduler where that move is the first one available to the appropriate player. The second case is correct because if $Z(s_1) = \varepsilon$ and $l \in \mathit{enabled}(s_1)$, then it is possible for the opposite scheduler to choose the string of labels s_1 and then when control reverts to the scheduler we are considering, since $l \in \mathit{enabled}(s_1)$, the labels will be chosen according subscheduler ρ_1, which corresponds to the strategy $\mathit{Strat}(\rho_1, V/s_1)$ by the induction hypothesis. On the other hand, if $Z(s_1) = \varepsilon$ and $l \notin \mathit{enabled}(s_1)$, then the situation is similar, except that the labels will be chosen according to subscheduler ρ_2, by the semantics of the "**if then else**" construct. The third case is correct because the scheduler 0 does not choose any transition, and the strategy ε does not tell its player to make any move.

Next, we prove that Strat always defines deterministic strategies. The first case only adds strings ending in a single move m and strings from a recursive call to Strat. Assuming that the recursive call produces a deterministic strategy, the strings ending in m can't violate determinacy since they all end in the same move. The second case does not violate determinism because it adds strings of one of two forms: either $s_1.s_2$ where $l \in \mathit{enabled}(s_1)$ and s_2 comes from a recursive call to Strat or $s_1.s_2$ where $l \notin \mathit{enabled}(s_1)$, $\mathit{enabled}_Z(s_1) \neq \emptyset$ and s_2 is again from a recursive call to Strat. So assuming that recursive calls to Strat do not violate determinacy, the whole strategy will be deterministic because it is clear that the two cases for s_1 are mutually exclusive, so only one strategy can be concatenated to each string s_1, and the whole strategy will be deterministic.

It is easy to see that $Strat(\rho, V)$ is a complete strategy as long as ρ is nonblocking. From the argument that the strategy corresponds to the scheduler, the completeness of the strategy follows directly. Since the scheduler is assumed to be nonblocking, the strategy must give its player a response to every situation arising from any sequence of the other player's moves, meaning that it is complete.

Now we show that $Strat(\rho, V)$ is always introspective. The intuition behind this proof is that all decisions about what strings to include in the strategy are based on the moves available to Z, so the strategy must be introspective.

We begin by proving that if ρ is of the form $\sigma(m_1).\sigma(m_2)....\sigma(m_n).0$, with no occurrences of the "**if then else**" construct, then $Strat(\rho, V)$ is an introspective strategy. We prove this by induction on the length of ρ.

Base Case: $\rho = 0$. Then $Strat(\rho, V) = \{\varepsilon\}$, which is an introspective strategy.

Induction Step: $\rho = \sigma(m_1).\sigma(m_2)...\sigma(m_n).0$. Let $s_1.m'$ and $s_2.m''$ be in $Strat(\rho, V)$, and s_1 and s_2 be Z indistinguishable.

If $Z(s_1) = \varepsilon$, then from the description of $Strat$, $m' = m$, and since s_1 and s_2 are Z indistinguishable, $Z(s_2) = \varepsilon$, so $m'' = m$ as well, and this case cannot violate the introspective condition.

If $Z(s_1) \neq \varepsilon$, then from the definition of $Strat$, it is clear that $Z(s_1) = m_1.m_2...m_i$ for some $i < n$ and it is also clear that $m' = m_{i+1}$. Since s_1 and s_2 are assumed to be Z indistinguishable, $Z(s_2) = m_1.m_2....m_i$ also, and so from the definition of $Strat$, $m'' = m_{i+1}$. Therefore, the introspective condition is not violated.

Now we outline the proof for the case of a scheduler that uses the "**if then else**" construct an arbitrary number of times. To be completely formal one needs a structural induction on the scheduler, and nested inside it an induction on the length of the string. However, this would be notationally obscure and uninsightful to write out in complete detail. The argument as we have given it should be clear.

Suppose that $s_1.m'$ and $s_2.m''$ are in $Strat(\rho, V)$ and that s_1 and s_2 are Z indistinguishable. Since s_1 and s_2 are Z indistinguishable, for any pair of prefixes s_1' and s_2' of s_1 and s_2 respectively, if both prefixes have Z moves available, then both prefixes have exactly the same moves available. Thus, any time a subscheduler of the form **if** l **then** ρ_1 **else** ρ_2 is encountered by the algorithm $Strat$, either l will be in $enabled(s_1')$ and in $enabled(s_2')$ or it will not be in either set. Thus, if $s_1 = s_1'.s_1''$ and $s_2 = s_2'.s_2''$, then either s_1'' is in $Strat(\rho_1, V/s_1')$ and s_2'' is in $Strat(\rho_1, V/s_2')$, or s_1'' is in $Strat(\rho_2, V/s_1')$ and s_2'' is in $Strat(\rho_2, V/s_2')$. This is true no matter how many nested "**if then else**" statements occur, and eventually a scheduler of the form $\sigma(m).\rho'$ must be reached. At this point, the argument that $m' = m''$ is the same as the argument in the case of a scheduler without "**if then else**" statements, proving that the strategies are indeed introspective.

Now we give a procedure to get a scheduler corresponding to a deterministic, complete, introspective strategy. Let P be a deterministically labelled process, S a strategy for player Z, and V the set of valid positions for P.

First, consider all positions in S of the form $s.m$ for some s where $Z(s) = \varepsilon$. We will group these positions together by their Z indistinguishability: write the set of all such positions in the strategy as

$$\{s_{1,1}.m_1, s_{1,2}.m_1, ..., s_{1,n_1}.m_1,$$
$$s_{2,1}.m_2, s_{2,2}.m_2, ..., s_{2,n_2}.m_2, ...$$
$$s_{n,1}.m_n, s_{n,2}.m_n, ..., s_{n,n_r}.m_n\}$$

such that $enabled(s_{i,j}) = enabled(s_{i,k})$ for all i, j, k. This means that $s_{i,j}$ and $s_{i,k}$ are Z indistinguishable, since $Z(s_{i,j}) = Z(s_{i,k}) = \varepsilon$. So we know that they must be followed by the same move in the strategy, namely m_i.

If $n = 1$, that is if all positions where $Z(s) = \varepsilon$ are Z indistinguishable, then there is only one first move m that Z can choose. In this case, the scheduler is $\sigma(m).\rho'$, where ρ' is the scheduler constructed recursively from the strategy

$$\bigcup_j S/s_{1,j}.m$$

and the set of valid positions

$$\bigcup_j V/s_{1,j}.m.$$

If $n > 1$, we proceed as follows. First consider the special case where for some $i \in \{1, ..., n\}$, there is some move m_i^* such that $m_i^* \in enabled(s_{i,j})$ and $m_i^* \notin enabled(s_{l,k})$ for $l \neq i$. In this case, our scheduler will be **if** m_i^* **then** $\sigma(m_i).\rho_i$ **else** ρ' where ρ_i is a scheduler recursively constructed from the strategy $\bigcup_{j=1}^{n_i} S/s_{i,j}.m_i$ and the set of valid positions $\bigcup_{j=1}^{n_i} V/s_{i,j}.m_i$, and ρ' is the scheduler constructed from the strategy S with all the positions beginning with $s_{i,j}.m_i$ removed and the set of valid positions with the same strings removed.

Of course, it may not be true that there is a single move like m_i^*. However, the *set of moves available* after $s_{i,j}$ is unique for each i. If we were to extend the scheduler syntax to allow us to say **if** $l_1 \wedge l_2 \wedge \ldots \wedge l_k$ **then** ρ_1 **else** ρ_2

it would be easy to define the requisite scheduler. However, this syntactic extension can easily be coded up as

if l_1 **then**
    if l_2 **then**
        ⋮
            if l_k **then** ρ_1
                else ρ_2
        ⋮
        else ρ_2
    else ρ_2

Since we can use this construction to distinguish any set of available moves, we can construct the correct scheduler the same way as in the previous case. ∎

CHAPTER 6
Games for Processes with Probabilistic Choice

In this chapter, I discuss labelled processes equipped with a probabilistic choice operator and a single scheduler or player that resolves all nonprobabilistic choices. In some ways, this situation is similar to the two-agent situation; the single nondeterministic agent interacts with the outcomes of probabilistic choices in much the same way as it interacts with the outcome of choices made by the other player in the two-player situation. On the other hand, the probabilistic choice cannot be said to be resolved according to a strategy since it is, of course, resolved completely probabilistically, according to the distributions built into the process definition.

I begin by giving background on probabilistic processes. Next, I discuss games, strategies and epistemic restrictions for these processes. Finally, I prove that these introspective strategies for processes with probabilistic choice are equivalent to the schedulers defined for processes with probabilistic choice defined in [CP07].

6.1 Syntax and Semantics for Probabilistic Processes

The syntax of these processes is almost the same as the syntax of processes with an independence operator. The only difference is that the brackets signifying an independent choice are replaced with a labelled probabilistic choice operator.

$$P, Q ::= 0 \mid l : \alpha.P \mid P + Q \mid l : \sum_i l_i : p_i P_i \mid P|Q \mid (\nu a)P$$

$$\text{ACT}\ \frac{}{l : \alpha\,.\,P \xrightarrow[l_X]{\alpha}_1 P} \qquad \text{RES}\ \frac{P \xrightarrow[\lambda]{\alpha}_p P' \quad \alpha \neq a, \bar{a}}{(\nu a)P \xrightarrow[\lambda]{\alpha}_p (\nu a)P'}$$

$$\text{SUM1}\ \frac{P \xrightarrow[\lambda]{\alpha}_p P'}{P + Q \xrightarrow[\lambda]{\alpha}_p P'} \qquad \text{PAR1}\ \frac{P \xrightarrow[\lambda]{\alpha}_p P'}{P|Q \xrightarrow[\lambda]{\alpha}_p P'|Q}$$

$$\text{COM}\ \frac{P \xrightarrow[l_X]{a}_1 P' \quad Q \xrightarrow[jX]{\bar{a}}_1 Q'}{P|Q \xrightarrow[(l,j)_X]{\tau}_1 P'|Q'} \qquad \text{PROB}\ \frac{}{l : \sum_i l_i : p_i P_i \xrightarrow[l_X.l_i]{\tau}_{p_i} P_i}$$

Figure 6–1: Operational semantics for processes with probabilistic choice

For a process of the form $l : \sum_i l_i : p_i P_i$, we also require that $\sum_i p_i = 1$.

The operational semantics for labelled processes with probabilistic choice, shown in Fig. 6–1, is generally similar to the operational semantics without probability, but with two significant changes. First, each transition between two processes now has a probability assigned to it, in addition to an action and string of labels like in the other operational semantics. Second, the SWITCH rule is replaced with the PROB rule, representing probabilistic choice; the choice is resolved by the process doing a silent transition to one of the subprocesses, with the probability indicated in the original process. The other rules are straightforward analogues of the traditional process algebra rules. Note that only a τ transition can have a probability other than one. This is why in the COM rule we require that the transitions taken by P and Q have probability one; in fact, this is the only possibility for these transitions. In the strings of labels, a label can either have a subscript X, if it is not a label on a branch of a probabilistic choice, or no added subscript, if it is a label on a branch of a probabilistic choice.

We will only consider deterministically labelled processes: processes where every transition has a unique string of labels.

Definition 6.1.1 *A probabilistic process P is* deterministically labelled *if the following conditions hold:*

1. *It is impossible for P to make two different transitions with the same labels: if $P \xrightarrow[s]{\alpha}_{p_1} P'$ and $P \xrightarrow[s]{\beta}_{p_2} P''$ then $\alpha = \beta$, $p_1 = p_2$, and $P' = P''$.*

2. *If $P \xrightarrow[l_X.l']{\tau}_{p} P'$ then there is no transition $P \xrightarrow[l_X]{\alpha} P''$ for any α or p or P''.*

3. *Whenever $P \xrightarrow[s]{\alpha}_{p} P'$ then P' is deterministically labelled.*

Finally, since we are considering probabilities, we must discuss how they are composed in transition sequences of process. To construct transition sequences, we assume that the probabilities at every step are independent from one another. Thus, the probability of a sequence of transitions is just the product of the probabilities of each transition in the sequence. This is formalized below.

6.2 Games, Valid Positions and Strategies

In this section, we define games and strategies on probabilistic labelled processes. The construction of games and strategies is similar to the two player construction, since the player interacts with the probabilistic choices in a way similar to how the two players interact in the nonprobabilistic case.

6.2.1 Valid Positions

First we define the extension of the transition relation to allow sequences of transitions, by concatenating the label strings and multiplying the probabilities.

Definition 6.2.1 *For any process P, $P \underset{\varepsilon}{\longrightarrow}_1 P$, and if $P \underset{s}{\overset{\alpha}{\longrightarrow}}_{p_1} P'$ and $P' \underset{s'}{\longrightarrow}_{p_2} P''$, then $P \underset{s.s'}{\longrightarrow}_{p_1 \cdot p_2} P''$.*

Now we define valid positions.

Definition 6.2.2 *If $P \underset{s}{\longrightarrow}_{p} P'$ then every prefix of s, including s, is a* valid position *for P.*

6.2.2 Strategies

Besides there only being one player, the definition of a strategy and the restrictions on strategies are quite similar to the two player case. We recall all the definitions here only for convenience.

We start by defining player moves and probabilistic moves.

Definition 6.2.3 *If $s.m_X$ is a valid position for P, then m_X is a* player move *in this valid position. If $s.l$ is a valid position for P, then l is a* probabilistic move *in this valid position.*

Now we can define strategies.

Definition 6.2.4 *In the game for process P, a* strategy *S is a set of valid positions for P such that $\varepsilon \in S$ and if $s.m \in S$ then m is a player move and every prefix of s ending with a player move is also in S.*

Now we define restrictions on strategies. These are the same as in the two player case.

Definition 6.2.5 *A strategy S is* deterministic *if whenever $s.m_1 \in S$ and $s.m_2 \in S$ then $m_1 = m_2$.*

To define completeness for strategies, we need the following definitions first.

Definition 6.2.6 *Let s be a valid position for P. We define $X(s)$, the string of player moves in s, inductively as follows:*

1. $X(\varepsilon) = \varepsilon$.
2. $X(s.m_X) = X(s).m_X$.
3. $X(s.l) = X(s)$.

Definition 6.2.7 *If V is the set of valid positions for a process and $s \in V$ then define $enabled(s) = \{m | s.m \in V\}$. Define $enabled_X(s) = \{m | s.m \in V$ and m is an X move$\}$.*

Now we define a complete strategy.

Definition 6.2.8 *Let V be the set of valid positions for some process. Strategy S is* complete *if for all $s \in S$, for every probabilistic move l such that $s.l \in V$ and $enabled(s.l) \neq \emptyset$ then $s.l.m \in S$ for some move m.*

To define introspective strategies, we first define indistinguishable positions for the player.

Definition 6.2.9 *Valid positions s_1 and s_2 are called player indistinguishable if they satisfy the following conditions:*

1. $X(s_1) = X(s_2)$.
2. $enabled_X(s_1) = enabled_X(s_2)$.

3. *For all prefixes s_1' of s_1 and s_2' of s_2, if there is a player move available at s_1' and at s_2', then $enabled_X(s_1') = enabled_X(s_2')$.*

Now we define introspective strategies.

Definition 6.2.10 *For process P, strategy S is introspective if for every pair of player indistinguishable positions s_1 and s_2, $s_1.m \in S$ if and only if $s_2.m \in S$ for all moves m.*

6.2.3 Execution of a probabilistic process with a strategy

Since a deterministic, complete strategy resolves all the nonprobabilistic choices in a probabilistic process, a process paired with a deterministic complete strategy gives a normalized distribution on possible executions of the process.

We cannot define a probability measure on the set of all valid positions for several reasons. First, the probability assigned to a valid position must be based on the probability of that execution of the process occurring, but not all valid positions actually represent possible executions. For example, for the process

$$l : (l_1 : .5(l' : a) + l_2 : .5(l'' : b))$$

l_X is a valid position, but there is no reasonable way to assign a probability to this valid position because alone, it does not represent a partial execution of the process. Furthermore, the fact that some valid positions represent partial executions and the combination of probabilistic and nonprobabilistic choice means that the sum of the probabilities of all the valid positions will usually be more than one. Thus, we will only define the probability measure on a special, restricted set of valid positions.

First, we define the notion of a *final* valid position: a valid position with no possible continuations.

Definition 6.2.11 *Let V be the set of all valid positions for a process. Define the set of* final *valid positions as $V_f = \{s | s \in V \text{ and } s.m \notin V \text{ for all } m\}$. s is a* final valid position *if $s \in V_f$.*

Next we will define the set of final continuations of valid positions in a deterministic complete strategy S.

Definition 6.2.12 *Let V be the set of valid positions for a process P and let S be a strategy for P. Define*

$$final(S) = \{s \in V_f | s \in S \text{ or } s = s'.l \text{ for some label } l \text{ and } s' \in S\}.$$

Since a deterministic complete strategy resolves all nonprobabilistic nondeterminism, and taking only the final valid positions removes all partial executions, this definition gives us a set on which a probability measure can be defined.

Definition 6.2.13 *If S is a deterministic, complete strategy for process P, define $\mu_P : final(S) \longrightarrow [0,1]$ as follows: for $s \in final(S)$, if $P \underset{s}{\overset{}{\longrightarrow}}_{p} P'$, then $\mu_P(s) = p$.*

We will prove that μ_P is indeed a probability measure, but first we need an auxiliary definition.

Definition 6.2.14 *For S a deterministic, complete strategy, define $S/s = \{s' | s.s' \in S\}$.*

Theorem 6.2.15 *If S is a deterministic, complete strategy for P, then $\mu_P : final(S) \longrightarrow [0,1]$ is a probability measure.*

Proof . Since μ_P is defined on singletons and then extended in the evident way to arbitrary sets and the overall space is finite it is clear that μ_P is additive. Thus, all we have to show is that

$$\mu_P(final(S)) = \sum_{s \in final(S)} \mu_P(s) = 1$$

This will be proved by induction on the length of the maximal element in $final(S)$.

Base Case : P is blocked. Then ε is the only valid position for P, so $\varepsilon \in V_f$ and $S = \{\varepsilon\}$ by definition of strategy, so $final(S) = \{\varepsilon\}$. And for any process P, $P \underset{\varepsilon}{\longrightarrow}_1 P$, so $\mu_P(\varepsilon) = 1$.

Case : S starts by choosing a move m that does not label a probabilistic choice, resulting in P going to P'. Then it is easy to see that S/m is a deterministic complete strategy for P', so by the induction hypothesis, $\mu_{P'}(final(S/m)) = 1$. Note, that every element of $final(S)$ is of the form $m.s$ where s is in $(S/m)_{co}$, by determinacy of S. Furthermore, since $P \underset{m}{\longrightarrow}_1 P'$, we see from the definition of μ_P that if $m.s \in final(S)$ then $\mu_P(m.s) = \mu_{P'}(s)$. Therefore, $\mu_P(final(S)) = \mu_{P'}(final(S/m)) = 1$.

Case : S starts by choosing a label l of a probabilistic move of the form $l : \sum_{i=1}^{n} l_i : p_i P_i$. For $i = 1$ to n, let

$$S_i = \begin{cases} S/(l.l_i) & \text{if } P_i \text{ is not blocked} \\ \{\varepsilon\} & \text{otherwise} \end{cases}$$

Then since S is complete, it is easy to see that for all i, S_i is a deterministic, complete strategy for P_i. Now, for a string s and a set

S', let $s \odot S' = \{s.s' | s' \in S'\}$. Then it can be shown that

$$final(S) = \bigcup_{i=1}^{n} l.l_i \odot final(S_i).$$

Furthermore,

$$\mu_P(l.l_i \odot final(S_i)) = \sum_{s' \in final(S_i)} \mu_P(l.l_i.s'),$$

but since $P \xrightarrow[l.l_i]{}_{p_i} P_i$, by definition 6.2.1, we have that $\mu_P(l.l_i.s') = p_i \cdot \mu_{P_i}(s')$. So altogether,

$$\begin{array}{lcll} \sum_{s \in final(S)} \mu_P(s) & = & \sum_{i=1}^{n} \mu_P(l.l_i \odot final(S_i)) & \\ & = & \sum_{i=1}^{n} p_i \cdot \mu_{P_i}(final(S_i)) & \\ & = & \sum_{i=1}^{n} p_i & \text{by induction hypothesis} \\ & = & 1 & \text{by definition} \end{array}$$

■

Since we have shown that for process P and deterministic complete strategy S, μ_P is a probability measure on $final(S)$, it is now simple to define the execution of a probabilistic process with a deterministic complete strategy. For simplicity, we only discuss complete executions: executions ending in a blocked process. Since we are only considering finite processes, this is sufficient. It would be straightforward to extend these results to partial executions, but this extension would require more formalism. Therefore, we define executions as follows:

Definition 6.2.16 *The execution of process P with deterministic, complete strategy S is the set $final(S)$ with the probability measure μ_P.*

6.3 Schedulers for Probabilistic Processes

We present the syntax and semantics for explicit schedulers for probabilistic processes, as developed in [CP07]. We will then be able to prove that deterministic, complete, introspective strategies are equivalent to these schedulers, just like in the nonprobabilistic processes.

The schedulers for probabilistic processes are the same as the schedulers for nondeterministic processes, but rather than two independent schedulers, there is only one scheduler. The single scheduler interacts with the outcomes of probabilistic choices the same way that the primary scheduler interacts with the secondary scheduler in the nonprobabilistic case. This makes it possible to use these syntactic schedulers to represent the independent resolution of probabilistic and nonprobabilistic nondeterminism in processes with probabilistic choice.

6.3.1 Syntax and Semantics for Probabilistic Processes with Schedulers

The syntax for these schedulers is the same as the syntax for the schedulers discussed in Chapter 4, but now a complete process has only one scheduler instead of two.

Let L represent a general label, a and b represent actions, $\bar{a}$ and $\bar{b}$ co-actions, τ the silent action, α and β generic actions, co-actions, or silent action, P and Q processes, and ρ and η schedulers. The syntax for a complete process is as follows:

$$
\begin{aligned}
P, Q &::= 0 \mid l : \alpha.P \mid P + Q \mid l : \sum_i l_i : p_i P_i \mid P|Q \mid (\nu a)P \\
\rho, \eta &::= \sigma(L).\rho \mid \textbf{if } L \textbf{ then } \rho \textbf{ else } \eta \mid 0 \\
CP &::= P \parallel \rho
\end{aligned}
$$

The operational semantics for probabilistic processes with schedulers is given in Fig. 6–2. The rules are analogous to those for processes without probability, except for the PROB rule, which is straightforward. Note that in the rules RES, SUM1, and SUM2, the scheduler is required to be of the form $\sigma(L).\rho$, because a scheduler beginning with an if statement may behave differently in a restricted process, a sum, or a parallel process by checking what actions are available.

Transition sequences for processes with schedulers are produced in the same way as those without schedulers; namely, by multiplying the probabilities along the path.

6.4 Correspondence Theorem

Now we can state and prove the main theorem of this chapter, relating schedulers to deterministic complete introspective strategies for the probabilistic case.

Theorem 6.4.1 *Given a deterministically labelled, probabilistic process P and a deterministic, complete, introspective strategy S for P, there is a scheduler ρ such that the probability μ_P of any possible complete execution of P with S is equal to the probability of the same execution of $P \parallel \rho$, according to the semantics of probabilistic schedulers.*

$$\text{ACT}\ \frac{}{l : \alpha\,.\,P \parallel \sigma(l).\rho \xrightarrow[l_X]{\alpha}_1 P \parallel \rho} \qquad \text{RES}\ \frac{P \parallel \sigma(L).\rho \xrightarrow[\lambda]{\alpha}_p P' \parallel \rho' \qquad \alpha \neq a, \bar{a}}{(\nu a)P \parallel \sigma(L).\rho \xrightarrow[\lambda]{\alpha}_p (\nu a)P' \parallel \rho'}$$

$$\text{SUM1}\ \frac{P \parallel \sigma(L).\rho \xrightarrow[\lambda]{\alpha}_p P' \parallel \rho'}{P + Q \parallel \sigma(L).\rho \xrightarrow[\lambda]{\alpha}_p P' \parallel \rho'} \qquad \text{PAR1}\ \frac{P \parallel \sigma(L).\rho \xrightarrow[\lambda]{\alpha}_p P' \parallel \rho'}{P|Q \parallel \sigma(L).\rho \xrightarrow[\lambda]{\alpha}_p P'|Q \parallel \rho'}$$

$$\text{COM}\ \frac{P \parallel \sigma(l) \xrightarrow[l_X]{a}_1 P' \parallel 0 \qquad Q \parallel \sigma(j) \xrightarrow[jX]{\bar{a}}_1 Q' \parallel 0}{P|Q \parallel \sigma(l,j).\rho \xrightarrow[(l,j)_X]{\tau}_1 P'|Q' \parallel \rho}$$

$$\text{PROB}\ \frac{}{l : \sum_i l_i : p_i P_i \parallel \sigma(l).\rho \xrightarrow[l_X.l_i]{\tau}_{p_i} P_i \parallel \rho}$$

$$\text{IF1}\ \frac{P \parallel \rho_1 \xrightarrow[\lambda]{\alpha}_{p_1} P' \parallel \rho' \qquad P \parallel \sigma(L) \xrightarrow[\lambda']{\beta}_{p_2} P'' \parallel 0}{P \parallel \textbf{if } L \textbf{ then } \rho_1 \textbf{ else } \rho_2 \xrightarrow[\lambda]{\alpha}_{p_1} P' \parallel \rho'}$$

$$\text{IF2}\ \frac{P \parallel \rho_2 \xrightarrow[\lambda]{\alpha}_{p_1} P' \parallel \rho' \qquad P \parallel \sigma(L) \not\rightarrow}{P \parallel \textbf{if } L \textbf{ then } \rho_1 \textbf{ else } \rho_2 \xrightarrow[\lambda]{\alpha}_{p_1} P' \parallel \rho'}$$

Figure 6–2: Operational semantics for processes with probabilistic choice and schedulers

Similarly, given any nonblocking scheduler ρ for P, there is a corresponding deterministic, complete, introspective strategy S for P such that the probability of any complete execution of $P \parallel \rho$ according to the semantics is equal to the probability μ_P of the corresponding valid position occurring when P is executed with S.

Proof . We begin by giving the constructions of the schedulers and the strategies asserted to exist in the statement of the theorem.

Let P be a deterministically labelled process, S a strategy, and V the set of valid positions for P. We will define a scheduler from S. If the strategy is $\{\varepsilon\}$ then the scheduler is 0. Now consider a nontrivial strategy. All of the positions in S must start with the same move which we will call m. Therefore, the scheduler must begin with $\sigma(m)$.

If m does not label a probabilistic move in P, then since P is deterministically labelled, $P \xrightarrow[m]{}_1 P'$ for exactly one process P'. In this case, let ρ be the scheduler corresponding to the deterministic, complete, introspective strategy S/m for P' (recall that $S/m = \{s' | m.s' \in S\}$). Then it is clear that the scheduler for P corresponding to S is $\sigma(m).\rho$.

If m labels a probabilistic move in P, then $P \xrightarrow[m.m_i]{}_{p_i} P_i$ for each i in the probabilistic sum. Since S is an introspective strategy, if $S/m.m_j \neq S/m.m_k$, then $m.m_j$ and $m.m_k$ are not X indistinguishable, which must mean that $enabled(m.m_j) \neq enabled(m.m_k)$. So we can take the equivalence classes of indistinguishable positions of the form $m.m_i$ and we will have one strategy for each equivalence class, by the introspective property of S. Thus, by the induction hypothesis there is a scheduler for each equivalence class. And it is clear that each equivalence class has a unique set of moves available after the initial move m has been chosen, so we can use the "**if**

then else" construct to combine these schedulers so that each one will be applied to the correct branches of the probabilistic choice.

The second construction gives a strategy from a scheduler.

Consider a deterministically labelled process P with scheduler ρ and let V be the set of valid positions for P. $Strat(\rho, V)$ gives the corresponding scheduler. It is defined inductively.

Case 1 : $Strat(0, V) = \{\varepsilon\}$.

Case 2 : $\rho = \sigma(m).\rho'$. Then $Strat(\sigma(m).\rho', V) = \{\varepsilon\} \cup \{m\} \cup \{s \in V | s = m.s' \text{ and } s' \in Strat(\rho', V/m)\} \cup \{s \in V | s = m.m'.s' \text{ and } m' \text{ is not a player move and } s' \in Strat(\rho', V/m.m')\}$. m must be in the strategy because strategies are prefix closed for strings ending with player moves. The set $\{s \in V | s = m.s' \text{ and } s' \in Strat(\rho', V/m)\}$ covers the case where m is not a probabilistic move, and $\{s \in V | s = m.m'.s' \text{ and } m' \text{ is not a player move and } s' \in Strat(\rho', V/m.m')\}$ adds the correct strings if m labels a probabilistic choice. Note that one of these last two sets must always be empty, depending on what m is.

Case 3 : $\rho =$ **if** l **then** ρ_1 **else** ρ_2. If $l.s \in V$ for any s, then $Strat($**if** l **then** ρ_1 **else** $\rho_2, V) = Strat(\rho_1, V)$. If there is no string in V of the form $l.s$, then $Strat($**if** l **then** ρ_1 **else** $\rho_2 = Strat(\rho_2, V)$.

The proof that such a strategy is complete, deterministic and introspective is exactly analogous to the non-probabilistic case.

To prove that the probability of a particular execution of a process is the same with a strategy as with its corresponding scheduler, we begin by defining the execution tree of a process with a scheduler. The execution tree

has two kinds of edges: action edges, representing actions in the execution of the process with the scheduler, and probabilistic edges, representing the branches of a probabilistic choice in the execution of the process with the strategy. Since the scheduler resolves all nonprobabilistic choice, the only branching in this tree is probabilistic branching. Each edge in the tree has a label and a probability. The labels of action edges are the same as the labels of their corresponding actions in the process, and their probabilities are all 1. Each probabilistic edge is labelled with the label of the corresponding branch of the probabilistic choice in the process, and its probability is the probability of its branch of the probabilistic choice.

It is clear that every node in the tree corresponds to an execution, complete or partial, of the process with the specified scheduler. Furthermore, it is clear that the probability of each execution is the product of all the probabilities along the path from the root to that node. The probability of the root is defined to be 1; this represents the fact that ε is a prefix of every execution, so it must occur with probability 1. Note that this is not a probability measure when it is considered on all the nodes, but it is a probability measure on the complete executions of the process with the scheduler.

Let $s \in final(S)$ where S is a deterministic, complete, introspective strategy for process P. Let ρ be the scheduler constructed from S. We prove that the probability of s in the execution tree of $P \parallel \rho$ is the same as $\mu_P(s)$ by induction on the length of s.

Case 1 : $s = \varepsilon$. ε is in every execution tree and its probability is defined to be 1, and $\mu_P(\varepsilon)$ is also 1 by definition, which completes the proof for this case.

Case 2 : $s = m.s'$. We will prove this case by subcases based on what kind of move m is.

Case 2 (a) : m does not label a probabilistic choice, so $P \xrightarrow[m]{}_1 P'$ and $\rho = \sigma(m).\rho'$ where ρ' is the scheduler corresponding to S/m. Then by the induction hypothesis s' is in the execution tree for $P' \parallel \rho'$ and its probability is equal to $\mu_{P'}(s)$. This means that the probability of $m.s$ in the execution tree of $P \parallel \rho$ is the same as $\mu_P(s)$.

Case 2 (b) : m is l, where l labels a probabilistic choice of the form $l : \sum_i l_i : p_i P_i$. Therefore the complete valid position beginning with m must be of the form $l.l_j.s'$ for some j and some string s'. It is clear from the construction of the scheduler that $l.l_j.s'$ is in the execution tree of $P \parallel \rho$. Furthermore, the probability assigned to $l.l_j.s'$ in the execution tree is p_j times the probability of s', which is equal to $\mu_{P_j}(s')$ by the induction hypothesis. And by definition of μ, we know that $\mu_P(m.m_j.s') = p_j \cdot \mu_{P_j}(s')$, so the two probabilities are equal.

Now we will prove that when a strategy is constructed from a scheduler according to this description, the probability of each execution according to the strategy is the same as the probability according to the scheduler from which it was constructed.

Let P be a process, ρ a nonblocking scheduler for P, and S the strategy constructed from ρ. We must prove that for a leaf in the execution tree of $P \parallel \rho$, if s is the path from the root to the leaf, then $s \in final(S)$, and the probability of s computed from the execution tree is equal to $\mu_P(s)$.

We prove this by induction on the length of the execution sequence.

Case 1 : $s = \varepsilon$. Since we assume that ε is a complete execution, P must be blocked, so ρ must be 0, and $S = \{\varepsilon\}$. The probability of ε according to the execution tree is defined to be 1, and $\mu_P(\varepsilon) = 1$ also by definition, which completes the proof for this case.

Case 2 : $s = m.s'$. We will prove this case by induction on the structure of the scheduler.

Case 2 (a) : $\rho = \sigma(m).\rho'$, so the strategy is constructed to be the set of all valid positions for P beginning with m and ending with a string which is in the strategy for ρ' and V/m or in the strategy for ρ' and $V/m.m_i$, if m labels a probabilistic choice.

Case 2 (a) (i) : m does not label a probabilistic move. Then $P \parallel \rho \xrightarrow[m]{}_1 P' \parallel \rho'$ and the probability of $m.s'$ in the execution tree is equal to the probability of s' in the execution tree, since the edge labelled with m must have probability 1. By the induction hypothesis, the probability of s according to the execution tree is equal to $\mu_{P'}(s')$, and by definition of μ_P, we know that $\mu_P(m.s') = \mu_{P'}(s')$. Therefore, the probability of $m.s'$ in the execution tree is equal to $\mu_P(m.s')$.

Case 2 (a) (ii) : $m = l$ where l labels a probabilistic choice of the form $l : \sum_{i=1}^{n} l_i : p_i P_i$. So for $i = 1$ to n, $P \parallel \rho \xrightarrow[l.l_i]{}_{p_i} P_i \parallel \rho_i$. Then s must be of the form $m.l_i.s'$ for some l_i, where s' is in the execution tree of $P_i \parallel \rho'$. By the first case of the definition of *Strat*, the string $m.l_i.s'$ is in the strategy for ρ. And the probability of $m.l_i.s'$ in the execution tree is the probability of the edge labelled m, which is 1, times the probability of the edge labelled l_i,

which is p_i, times the probability of s' in the execution tree, which is by the induction hypothesis equal to $\mu_{P_i}(s')$. Thus, $\mu_P(m.l_i.s') = p_i \cdot \mu_{P_i}(s')$, so the two probabilities are equal.

Case 2 (b) : $\rho =$ **if** m **then** ρ_1 **else** ρ_2. The strategy corresponding to this scheduler is either the strategy for ρ_1 or the strategy for ρ_2. In either case, the result holds by the induction hypothesis.

∎

CHAPTER 7
Epistemic Logic

In this chapter we present a dynamic epistemic logic intended to reason about games on processes, particularly knowledge, information flow, and the effects of actions on knowledge. At the moment, this is a somewhat preliminary investigation.

We consider two-player processes with a switch operator rather than probabilistic processes. We take the set of all valid positions for a process as our universe. Our logic will allow us to discuss several aspects of any given valid position:

- Which player made the last move and what the last move was,
- What moves are available and what player they belong to,
- What formulas are satisfied by specific continuations of the current valid position,
- What formulas are satisfied by specific prefixes of the current valid position,
- The knowledge of each player in the current state, according to the introspective indistinguishability condition discussed in Chapter 3, and
- What formulas were satisfied by the state immediately after either player's last move.

Our logic not only allows us to discuss players' knowledge according to the correct equivalence relation on states, but also allows us to characterize logically the introspective indistinguishability relation itself.

7.1 Syntax and Semantics

We take the valid positions for a certain process as our states. For a valid position s and a formula ϕ, we say that $s \models \phi$ if ϕ is true at s.

Let L represent a general label (a single label or a synchronizing pair of labels), m a move (a general label together with a player), let X and Y be the two players, and let Z represent either X or Y.

$$\phi ::= C_Z(L) \mid A_Z(L) \mid \bigcirc_m \phi \mid \ominus\phi \mid K_Z\phi \mid @_Z\phi \mid \phi \wedge \phi \mid \neg\phi \mid \top.$$

The formula $C_Z(L)$ means that the last move taken was L_Z (general label L was chosen by player Z). $A_Z(L)$ means that move L_Z is available at the current valid position. $\bigcirc_m\phi$ means that m is an available move at the current valid position and ϕ will be true after m is played. $\ominus\phi$ means that ϕ was true at the previous valid position before this one. $K_Z\phi$ means that ϕ is true at every state that player Z considers equivalent to the current state-the usual Kripke interpretation of knowledge, as in [Kri63]. If a fact is true at every state equivalent to the current one, then it is true at every state that the agent considers to be possible, so the agent is said to know the fact. Finally, $@_Z\phi$ means that ϕ was true immediately after player Z's last move. These meanings are formalized in the semantics:

1. $s.L_Z \models C_Z(L)$.
2. $s \models A_Z(L)$ if $s.L_Z \in V$.
3. $s \models \bigcirc_m\phi$ if $s.m \in V$ and $s.m \models \phi$.

4. $s.m \models \ominus\phi$ if $s \models \phi$.

5. $s \models K_Z\phi$ if for all $s' \sim_Z s$, $s' \models \phi$.

6. $s \models @_Z\phi$ if $s = s'.L_Z$ and $s \models \phi$ or $s = s'.L_Z.L_{\bar{Z}}^1.L_{\bar{Z}}^2...L_{\bar{Z}}^n$ and $s'.L_Z \models \phi$.

7. $s \models \phi_1 \wedge \phi_2$ if $s \models \phi_1$ and $s \models \phi_2$.

8. $s \models \neg\phi$ if it is not the case that $s \models \phi$.

9. $s \models \top$ for all s.

Most of the connectives are straightforward, but the knowledge operator requires more discussion. In particular, we must define the equivalence relation $\sim_Z$. In fact, any equivalence relation on valid positions would be consistent with our syntax, but we will use the specific indistinguishability relation discussed above in the definition of an introspective strategy. In the following definition, recall that $Z(s)$ means the string of Z moves from the valid position S, $enabled_Z(s) = \{L | s.l_Z \in V\}$ where V is the set of valid positions for the relevant process, and $\leq$ refers to the prefix ordering.

Definition 7.1.1 *$s_1 \sim_Z s_2$ if all of the following conditions hold:*

1. $Z(h_1) = Z(h_2)$

2. $enabled_Z(s_1) = enabled_Z(s_2)$

3. *For all $s_1' \leq s_1$, $s_2 \leq s_2'$, if $Z(s_1') = Z(s_2')$ then $enabled_Z(s_1') = enabled_Z(s_2')$ or $enabled_Z(s_1') = \emptyset$ or $enabled_Z(s_2') = \emptyset$.*

Finally, note that in the syntax and semantics we only discuss the traditional logical connectives $\wedge$ and $\neg$, so that the notation is concise. However, from now on we will use $\phi_1 \vee \phi_2$ as shorthand for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi_1 \Rightarrow \phi_2$ for $\neg\phi_1 \vee \phi_2$, and $\phi_1 \Leftrightarrow \phi_2$ for $(\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)$. On the other hand, we

do not actually need the operator $A_Z(L)$ since it is equivalent to $\bigcirc_{L_Z}\top$ but we leave it in our syntax and semantics anyway, to make the explanations simpler.

7.2 Logical Characterization of Indistinguishability

One interesting use for our logic is to find a class of formulas so that two valid positions are indistinguishable in the introspection sense if and only if the two valid positions agree on all formulas in this class. In other words, we will define two sets of formulas Φ_X and Φ_Y such that for all valid positions s and t, $s \sim_Z t$ if and only if for all formulas ϕ in Φ_Z, $s \models \phi \iff t \models \phi$.

Theorem 7.2.1 *$s \sim_Z t$ if and only if s and t agree on all formulas of the form*

$$(@_Z \ominus)^n @_Z C_Z(L)$$

for $n \geq 0$, and for any L, and also agree on all formulas of the form

$$(@_Z \ominus)^n A_Z(L)$$

for $n \geq 0$ and for any L.

Proof . s and t agreeing on all formulas of the form $(@_Z \ominus)^n @_Z C_Z(L)$ is equivalent to $Z(s) = Z(t)$, because $s \models @_Z C_Z(L)$ if and only if L is the last Z move in s, and $s \models @_Z \ominus @_Z C_Z(L')$ if and only if L' is the second to last Z move in s, and so on. So if two valid positions agree on all such formulas, they must have the same Z moves in the same order.

s and t agreeing on all formulas of the form $A_Z(L)$ (i.e. $(@_Z \ominus)^0 A_Z(L)$) clearly means that $enabled_Z(s) = enabled_Z(t)$, and s and t agreeing on all formulas of the form $(@_Z \ominus)^n A_Z(L)$ is equivalent to the third condition in the indistinguishability definition. This is because we have already ensured

that $Z(s) = Z(t)$ so $(@_Z\ominus)^n$ means counting backwards n Z moves and n contiguous series of $\bar{Z}$ moves, and then checking that $enabled_Z$ is the same in the two strings. This shows that two valid positions agree on all formulas of the specified forms if and only if they are Z-indistinguishable. ∎

The fact that there is a class of formulas characterizing the equivalence relation used to define knowledge has interesting implications for our logic. For example, the fact that whenever $s \sim_Z t$, s and t necessarily agree on certain formulas, combined with our definition of knowledge as anything that is true at all equivalent states, means that whenever one of the specified formulas is true, the agent knows this fact.

Theorem 7.2.2 *For any player Z, for all labels L and for all $n \geq 0$, the following formulas are true at all valid positions.*

1. $(@_Z\ominus)^n @_Z C_Z(L) \Rightarrow K_Z((@_Z\ominus)^n @_Z C_Z(L))$
2. $(@_Z\ominus)^n A_Z(L) \Rightarrow K_Z((@_Z\ominus)^n A_Z(L))$
3. $(\neg(@_Z\ominus)^n @_Z C_Z(L)) \Rightarrow K_Z(\neg((@_Z\ominus)^n @_Z C_Z(L))$
4. $(\neg(@_Z\ominus)^n A_Z(L)) \Rightarrow K_Z(\neg(@_Z\ominus)^n A_Z(L))$
5. $K_Z((@_Z\ominus)^n @_Z C_Z(L)) \vee K_Z(\neg((@_Z\ominus)^n @_Z C_Z(L))$
6. $K_Z((@_Z\ominus)^n A_Z(L)) \vee K_Z(\neg(@_Z\ominus)^n A_Z(L))$

Proof . First, recall that $s \models K_Z\phi$ if for all $s' \sim_Z s$, $s' \models \phi$. And we showed in theorem 7.2.1 that whenever $s \sim_Z s'$, s and s' agree on all formulas of the form $(@_Z\ominus)^n @_Z C_Z(L)$. Therefore, if $s \models (@_Z\ominus)^n @_Z C_Z(L)$, then for all $s' \sim_Z s$, $s' \models (@_Z\ominus)^n @_Z C_Z(L)$, which means that $s \models K_Z(@_Z\ominus)^n @_Z C_Z(L)$. Similar reasoning applies to the second formula.

The third formula follows from the fact that if $s \models \neg(@_Z \ominus)^n @_Z C_Z(L)$ then $s \not\models (@_Z \ominus)^n @_Z C_Z(L)$, so for all $s' \sim_Z s$, $s' \not\models (@_Z \ominus)^n @_Z C_Z(L)$ and therefore $s' \models \neg(@_Z \ominus)^n @_Z C_Z(L)$, so that $s \models K_Z(\neg(@_Z \ominus)^n @_Z C_Z(L))$. The fourth formula is simple.

The fifth formula follows directly from the fact that $(@_Z \ominus)^n @_Z C_Z(L) \vee \neg(@_Z \ominus)^n @_Z C_Z(L))$, and from the first and third formulas. The last formula is similar. ∎

7.3 Axioms

One of the uses of this logic is to allow us to determine some axioms that are true at every state. Now that we have a formal logic for discussing valid positions, it is easier to determine examples of such axioms, allowing us better to understand the properties of valid positions and the relationships between them. The four formulas discussed at the end of the last section are axioms, and now we will survey some others.

Here are a few examples of axioms:

1. $\bigcirc_m \phi \Rightarrow \neg \bigcirc_m \neg\phi$.

   This formula is true because we require our processes to be deterministically labelled. Thus, there is at most one state that any valid position can transfer to for any given move m, and any formula that can possibly hold after m therefore must hold after m.

2. $\phi \Rightarrow \neg \bigcirc_m \neg \ominus \phi$.

   This formula is true because our states have a tree structure: there is at most one immediate previous state for any valid position.

3. $C_Z(L) \Rightarrow \ominus A_Z(L)$.

This formula says that if a move was chosen in the previous state, it must have been available there.

4. $A_Z(L) \Rightarrow \bigcirc_{L_Z} C_Z(L)$.

   This formula says that if a move is enabled, then there is a next state where that move was chosen. The last two formulas seem obvious, but formal expressions of the relationships between the operators are often useful, and are necessary to give a complete axiomatization for the logic.

Since we define knowledge using an equivalence class on states in the normal Kripke way, we automatically know that the knowledge axioms as discussed, for example, in [Kri63], are true:

1. $K_Z\phi \Rightarrow \phi$.

   This can be interpreted as saying that knowledge is true.

2. $K_Z\phi \Rightarrow K_Z K_Z\phi$.

   This means that the agents are aware that they know what they know.

3. $(K_Z(\phi \Rightarrow \psi) \wedge K_Z\phi) \Rightarrow K_Z\psi$.

   Agents can reason and form new knowledge from what they know.

4. $\neg K_Z\phi \Rightarrow K_Z \neg K_Z\phi$.

   If an agent does not know something, he is aware of this fact.

We have no reason to believe that the axioms we have given form a complete set of axioms for our logic, but having these axioms already gives us information about our framework that would have been quite difficult to

prove or even to state without the formalism of this logic. In the future, we hope to develop a complete axiomatization for this logic.

CHAPTER 8
Conclusion

In this paper we have given a semantic treatment of a process algebra with two kinds of choice in terms of games and strategies. This gives a semantic understanding of the "knowledge" possessed by schedulers when they resolve choices. This epistemic aspect is captured by restrictions on what the schedulers can see when they execute their strategies. In this short version we have not discussed the probabilistic case; we have, however, developed the theory for that case as well and have proved the correspondence theorem.

As far as we know there has been no work on a game semantics for process algebras with the notion of multiple schedulers. This work is a first step toward a systematic game-semantic exploration of concurrency. First of all, we would like to develop a new paradigm for process algebra that is more naturally adapted to games. This will lead to richer notions of interactions between agents than synchronization and value or name passing.

Second, we would like to enrich the epistemic aspects of the subject. In particular, we would like to move toward an explicit combination of modal process logic and epistemic logic so that we can describe in a compositional process-algebraic way how agents learn and exchange knowledge. This has already been partly done but we are far from a complete axiomatization. We would like to carry out a much more thorough development of the modal logic from chapter 7. We are especially interested in completeness.

Third, we would like to explore more subtle notions of transfer of control between the agents. Thus, for example, there could be a protracted dialogue between the agents before they decide on a process move. This could conceivably be fruitful for incorporating higher-order or mobile processes.

Finally, we would like to combine the epistemic and probabilistic notions using ideas from information theory [Sha48]. These ideas have been used for an for an analysis of anonymity [CPP06], indeed it was that investigation that sparked the research reported in [CP07] and which ultimately led to the present work.

REFERENCES

[AG97] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[AJ94] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *J. Symbolic Logic*, 59(2):543–574, 1994.

[AJM00] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.

[BP05] Mohit Bhargava and Catuscia Palamidessi. Probabilistic anonymity. In *Proc. of CONCUR*, volume 3653 of *LNCS*, pages 171–185. Springer, 2005.

[CKP09] Konstantinos Chatzikokolakis, Sophia Knight, and Prakash Panangaden. Epistemic strategies and games on concurrent processes. In *SOFSEM*, pages 153–166, 2009.

[CNP09] Konstantinos Chatzikokolakis, Gethin Norman, and David Parker. Bisimulation for demonic schedulers. In *FOSSACS '09: Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, pages 318–332, Berlin, Heidelberg, 2009. Springer-Verlag.

[CP07] Konstantinos Chatzikokolakis and Catuscia Palamidessi. Making random choices invisible to the scheduler. In *CONCUR*, LNCS 4703, pages 42–58, 2007.

[CPP06] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. In *Proc. of TGC*, number 4661 in LNCS, pages 281–300, 2006.

[DH01] Vincent Danos and Russell Harmer. The anatomy of innocence. *Lecture Notes in Computer Science*, 2142:188–202, 2001.

[FHMV95] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[HM84] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. In *Proc. of Principles of Distributed Computing*, pages 50–61, 1984.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–162, 1985.

[HO00] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF. *Information and Computation*, 163:285–408, 2000.

[KNP90] Paul Krasucki, Gilbert Ndjatou, and Rohit Parikh. Probabilistic knowledge and probabilistic common knowledge. In *ISMIS 90*, pages 1–8. North Holland, 1990.

[Kri63] S. Kripke. Semantical analysis of modal logic. *Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Mil80] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI Conference on Theoretical Computer Science*, number 104 in Lecture Notes In Computer Science, pages 167–183. Springer-Verlag, 1981.

[Sha48] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423,623–656, July and October 1948.

[SS96] Steve Schneider and Abraham Sidiropoulos. Csp and anonymity. In *In European Symposium on Research in Computer Security*, pages 198–218. Springer-Verlag, 1996.