Epistemic Strategies and Games on Concurrent Processes

Konstantinos Chatzikokolakis[1], Sophia Knight[2], and Prakash Panangaden[2]

[1] Computing Laboratory, Oxford University and LIX, École Polytechnique
[2] School of Computer Science McGill University

Abstract. We develop a game semantics for process algebra with two interacting agents. The purpose of our 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. We 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. We show that different restrictions on strategies represent different amounts of information being available to a scheduler. We 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.

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. The most commonly used equivalence is bisimulation. There is also a modal logic which exactly characterizes bisimulation. This combination of algebraic and logical principles is powerful for reasoning about concurrency.

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 [1, 2]. 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 situations where omniscient schedulers are unreasonably powerful and create a situation 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 holds 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, does one traditionally treat security issues using process algebra? In fact a scrutiny reveals that they do not have a completely demonic scheduler all the time. For example, Schneider and Sidiropoulos [3] 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[3][4]. Anonymity is a problem where these issues manifest themselves particularly sharply.

Even bisimulation, a notion often used in the analysis of security properties, doesn't 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 $t \xrightarrow{a} t'$ ... 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 [5]: 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.

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

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 paper 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 [6–8] 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. 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 these schedulers solve certain security problems, this is the first time that the epistemic qualities of these schedulers have been made explicit.

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. 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. Indeed, previously one only had an intuitive notion of what the schedulers of [5] "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 a beneficial way to model interaction and independence in process algebra.

Related work There are many kinds of games used in mathematics, logic and computer science. 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 [8] and Hyland and Ong [7] on full abstraction for PCF. These games do not have the

notion of winning. Rather the games simply delineate sets of possible plays and *strategies* are used to model programs. This has been a very fruitful paradigm and many researchers - far too many to enumerate - have contributed to this flourishing paradigm. 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 condition 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 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 language. While there have been some connections drawn between concurrent languages like the π-calculus and dialogue games [7] 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.

A very important concept that pervades game semantics is "innocence" [7, 9]. 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.

2 Background

We begin by introducing a process calculus with labelled actions and a protection operator. The labels on actions allow us to control what is visible 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 top-level action in the protected subprocess must be chosen independently from unprotected actions.

We let l 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. 1. There are corresponding right rules for $+$ and $|$; these operators are both associative and commutative. There is an additional condition that no derivation tree for a transition may contain more than one occurrence of the SWITCH rule. The

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

$$\text{PAR1}\ \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \text{COM}\ \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad \text{SWITCH}\ \frac{P \xrightarrow{\tau} P'}{l : \{P\} \xrightarrow{\tau} P'}$$

Fig. 1. Operational semantics

reason for this condition is explained below. 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 these two restrictions 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. Only one occurrence of the SWITCH rule is allowed in a derivation tree because we only require one level of independence; this is sufficient to capture independence of choices.

The set $tl(P)$ of top level labels of P is defined as $tl(l : \alpha.P) = tl(l : \{P\}) = \{l\}$, $tl(P|Q) = tl(P + Q) = tl(P) \cup tl(Q)$, $tl((\nu a)P) = tl(P)$, and $tl(0) = \emptyset$.

Definition 1. *A process P is* deterministically labelled *if for every process P' that can be reached by any series of zero or more transitions from P, each label occurs at most once in the top level labels for P'.*

Roughly, this means that two enabled actions never have the same label. For example, $P = l_1 : a \ + \ l_1 : \{l_2 : \tau\}$ is not deterministically labelled because l_1 occurs twice in the top level labels for P, and no process with this as a subprocess is deterministically labelled. However, $l_1 : a \,.\, l_3 : c \ + \ l_2 : b \,.\, l_3 : c$ is deterministically labelled even though l_3 occurs twice.

3 Games and Strategies

In this section 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. We define strategies for 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*, which will be discussed in the next section.

3.1 Valid Positions

Valid positions are defined on a process and represent valid plays for that process, with player X moving first. Every valid position is a string of moves (labels on 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 2. *A valid position for a labelled process P is defined inductively:*

1. *ε is a valid position for any process P.*
2. *If $P \xrightarrow{\alpha} P'$ and there is no occurrence of the SWITCH rule or the COM rule in the derivation tree for this transition, and if s is a valid position for P' and l is the label for the action α, then $l.s$ is a valid position for P and this occurrence of l is an X move.*
3. *If $P \xrightarrow{\tau} P'$ and the COM rule occurs in the derivation tree for this transition but the SWITCH rule does not occur anywhere, and if the synchronizing actions a and $\bar{a}$ are labelled l_1 and l_2 respectively, and s is a valid position for P', then $(l_1, l_2).s$ is a valid position for P and (l_1, l_2) is an X move.*
4. *If $P \xrightarrow{\tau} P'$ and the SWITCH rule occurs in the derivation tree for this transition but the COM rule does not, and the bracketed subprocess chosen in the SWITCH rule is labelled l_1 and the action chosen within the bracketed subprocess is labelled l_2, and s is a valid position for P', then l_1 and $l_1.l_2.s$ are both valid positions for P, and l_1 is an X move and l_2 is a Y move.*
5. *If $P \xrightarrow{\tau} P'$ and the SWITCH rule and COM rule both occur in the derivation tree for this transition, and if the bracketed subprocess in the SWITCH rule is labelled l_1 and the actions synchronized within the bracketed subprocess are labelled l_2 and l_3, and s is a valid position for P', then l_1 and $l_1.(l_2, l_3).s$ are both valid positions for P and l_1 is an X move and (l_2, l_3) is a Y move.*

Note that the set of valid positions is prefix closed. The fourth clause of the definition is necessary to ensure this, since the fifth and sixth clauses both add two moves sequentially to the string.

Example 1. 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 two of the valid positions for P, with the Y moves in bold: $l_1.\mathbf{k_1}.l_2.(l_3, l_4).l_5$ and $l_1.\mathbf{k_2}.l_2.(l_3, l_4).l_6$.

3.2 Strategies

A strategy for a player is a subset of the valid positions, each valid position ending with moves 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.

Definition 3. *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 and every prefix of s ending with a Z move is in S.*

Example 2. For

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

one strategy for X is: $\{\varepsilon,\ l_1,\ l_1.k_2.l_2,\ l_1.k_2.l_2.(l_3, l_4),\ l_1.k_2.l_2.(l_3, l_4).l_6\}$. Another strategy for X is: $\{\varepsilon,\ l_1,\ l_1.k_1.l_2,\ l_1.k_2.l_2\}$. One strategy for Y is: $\{\varepsilon,\ l_1.k_1,\ l_1.k_2\}$.

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

Definition 4. *A strategy S is* deterministic *if: $s.m_1 \in S, s.m_2 \in S$ implies $m_1 = m_2$.*

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 5. *Let V denote the set of valid positions for P. For s a valid position for P, define $enabled(s) = \{m | s.m \in V\}$. Define $enabled_X(s) = \{m | s.m \in V$ and m is an X move$\}$ and define*
$enabled_Y(s) = \{m | s.m \in V$ and m is a Y move$\}$.

Note that a position can have X moves enabled or Y moves enabled, but not both.

Definition 6. *If s is a valid position for P, then $X(s)$ is the string of the X moves in s, and $Y(s)$ is the string of the Y moves in s.*

Definition 7. *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 $Z(s) = \varepsilon$ ensures that a strategy can take into account all the moves made by the opponent but, of course, we do not want to quantify over moves made by the strategy's own player.

Example 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 strategy for X given above, $S = \{\varepsilon,\ l_1,\ l_1.k_2.l_2,\ l_1.k_2.l_2.(l_3, l_4),\ l_1.k_2.l_2.(l_3, l_4).l_6\}$ is not complete, because $l_1 \in S$, $l_1.k_1 \in V$, $enabled_X(l_1.k_1) \neq \emptyset$ and $X(k_1) = \varepsilon$, but $l_1.l_2.m \notin S$ for any move m. Here is a complete strategy for X:

$$\{\varepsilon,\ l_1,\ l_1.k_1.l_2,\ l_1.k_2.l_2,\ l_1.k_1.l_2.(l_3, l_4),\ l_1.k_2.l_2.(l_3, l_4),\ l_1.k_1.l_2.(l_3, l_4).l_5,\ l_1.k_2.l_2.(l_3, l_4).l_6\}.$$

Proposition 1. *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.

Now we define the execution of a process according to a valid position. If s is a valid position for P, then we will say $P \stackrel{s}{\Longrightarrow} P'$ if starting from process P, the labels in s can be chosen in order and the end result will be process P'.

Definition 8. *Let P be a deterministically labelled process.*

1. $P \stackrel{\varepsilon}{\Longrightarrow} P$.
2. $P \stackrel{l}{\Longrightarrow} P'$ *if* $P \stackrel{\alpha}{\longrightarrow} P'$, *there is no occurrence of the SWITCH rule or the COM rule in the derivation tree for this transition, and the action α is labelled l.*
3. $P \stackrel{(l_1, l_2)}{\Longrightarrow} P'$ *if* $P \stackrel{\tau}{\longrightarrow} P'$ *and the COM rule occurs in the derivation tree for this transition but the SWITCH rule does not, and the synchronizing actions are labelled l_1 and l_2.*

4. $P \stackrel{l_1.l_2}{\Longrightarrow} P'$ *if* $P \stackrel{\tau}{\longrightarrow} P'$ *and the SWITCH rule occurs in the derivation tree for this transition but the COM rule does not, and the bracketed subprocess in the SWITCH rule is labelled* l_1 *and the action chosen within the bracketed subprocess is labelled* l_2.

5. $P \stackrel{l_1.(l_2,l_3)}{\Longrightarrow} P'$ *if* $P \stackrel{\tau}{\longrightarrow} P'$ *and the SWITCH rule and COM rule both occur in the derivation tree for this transition, and the bracketed subprocess in the switch rule is labelled* l_1 *while the synchronizing actions within the bracketed subprocess are labelled* l_1 *and* l_2.

6. $P \stackrel{l.s}{\Longrightarrow} P'$ *if* $P \stackrel{l}{\Longrightarrow} P''$ *and* $P'' \stackrel{s}{\Longrightarrow} P'$.

7. $P \stackrel{l_1.l_2.s}{\Longrightarrow} P'$ *if* $P \stackrel{l_1.l_2}{\Longrightarrow} P''$ *and* $P'' \stackrel{s}{\Longrightarrow} P'$.

It is easy to see that $P \stackrel{s}{\Longrightarrow} P'$ for some P' if and only if s is a valid position for P. Note that if s is a valid position for P, then $P \stackrel{s}{\Longrightarrow} P'$ for exactly one process P'. The determinacy of P's labelling forces the process P' to be unique.

Define the execution of a deterministically labelled process P with deterministic, complete X and Y strategies S_1 and S_2 as follows: Let s be the maximal element in $S = \{s \in S_1 \cup S_2 \mid \text{every prefix of } s \text{ is in } S_1 \cup S_2\}$. The execution of P according to S_1 and S_2 is the sequence of processes $P, P_1 ... P_n$ such that

$$P \stackrel{s_1}{\Longrightarrow} P_1 \stackrel{s_2}{\Longrightarrow} P_2 \stackrel{s_3}{\Longrightarrow} ... \stackrel{s_{n-1}}{\Longrightarrow} P_{n-1} \stackrel{s_n}{\Longrightarrow} P_n$$

and $s = s_1 s_2 ... s_n$ and each s_i is either a single X move or an X move followed by a Y move. 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.

3.4 Epistemic Restrictions on Strategies

Since certain strategies determine the execution of a process, we can use epistemic aspects of strategies to represent interacting agents' restricted knowledge. 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." 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 $s_2.m \in S$. 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 9. *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. *Z has at least one move available at s_1 and at least one move available at s_2.*
3. *For all prefixes s_1' and s_2' of s_1 and s_2 respectively, 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, we describe an indistinguishability relation on positions where player Z has a move available. We define two positions as 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 10. *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 things are all the same at two positions, the player cannot distinguish them, so he makes the same move at both positions.

Example 4. 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, $S = \{\varepsilon,\ l_1,\ l_1.k_1.l_2,\ l_1.k_2.l_2,\ l_1.k_1.l_2.(l_3, l_4),\ l_1.k_2.l_2.(l_3, l_4),\ l_1.k_1.l_2.(l_3, l_4).l_5,\ l_1.k_2.l_2.(l_3, l_4).l_6\}$ is not introspective. This is because in order to satisfy the introspection condition, $l_1.k_1.l_2.(l_3, l_4)$ and $l_1.k_2.l_2.(l_3, l_4)$ should have the same moves appended to them in S, since they are X indistinguishable. However, $l_1.k_1.l_2.(l_3, l_4).l_5 \in S$ and $l_1.k_2.l_2.(l_3, l_4).l_5 \notin S$, and similarly, $l_1.k_2.l_2.(l_3, l_4).l_6 \in S$ and $l_1.k_2.l_2.(l_3, l_4).l_5 \notin S$.

An example of an introspective strategy for X is:

$\{\varepsilon,\ l_1,\ l_1.k_1.l_2,\ l_1.k_2.l_2,\ l_1.k_1.l_2.(l_3, l_4),\ l_1.k_2.l_2.(l_3, l_4),\ l_1.k_1.l_2.(l_3, l_4).l_5,\ l_1.k_2.l_2.(l_3, l_4).l_5\}$.

Here is an example showing why the prefixes of the valid positions are discussed in the definition of introspective.

Example 5. Consider $P = l_0 : \{k_1 : \tau.(l_1 : c.(l_4 : f + l_5 : g) + l_2 : d) + k_2 : \tau.(l_1 : c.(l_4 : f + l_5 : g) + l_3 : e)\}$. Let X's strategy be $S =$

$$\{\varepsilon,\ l_0,\ l_0.k_1.l_1,\ l_0.k_2.l_1,\ l_0.k_1.l_1.l_4,\ l_0.k_2.l_1.l_5\}.$$

This strategy is introspective. Even though $X(l_0.k_1.l_1) = X(l_0.k_2.l_1)$ and $enabled_X(l_0.k_1.l_1) = enabled_X(l_0.k_2.l_1)$, it is acceptable that the two strings have different moves appended to them, because $enabled_X(l_0.k_1) = \{l_1, l_2\}$ and $enabled_X(l_0.k_2) = \{l_1, l_3\}$. This can be thought of as X being able to distinguish between the two positions $l_0.k_1.l_1$ and $l_0.k_2.l_1$ 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.

4 Correspondence between Strategies and Schedulers

In this section, we first review the syntactic schedulers defined in [5] and then prove that deterministic complete introspective strategies correspond exactly to these schedulers. This result is important because these schedulers were 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.

4.1 Background on Schedulers

The new ingredient in the process calculus is explicit syntax for a pair of schedulers. A *complete process* is an ordinary process augmented with a pair of schedulers. The notations $\sigma(l)$ and $\sigma(l_1, l_2)$ are used to designate choices made by the schedulers. The latter is used to indicate that the scheduler has chosen to synchronize two processes.

$$\rho, \eta ::= \sigma(l).\rho \mid \sigma(l,k).\rho \mid \textbf{if } l \textbf{ then } \rho \textbf{ else } \eta \mid 0$$
$$CP ::= P \parallel \rho, \eta$$

The rules for the operational semantics of the process calculus with schedulers

$$\text{ACT}\ \frac{}{l : \alpha.P \parallel \sigma(l).\rho, \eta \xrightarrow{\alpha} P \parallel \rho, \eta} \qquad \text{RES}\ \frac{P \parallel \rho, \eta \xrightarrow{\alpha} P' \parallel \rho', \eta' \quad \alpha \neq a, \bar{a}}{(\nu a)P \parallel \rho, \eta \xrightarrow{\alpha} (\nu a)P' \parallel \rho', \eta'}$$

$$\text{SUM1}\ \frac{P \parallel \rho, \eta \xrightarrow{\alpha} P' \parallel \rho', \eta'}{P + Q \parallel \rho, \eta \xrightarrow{\alpha} P' \parallel \rho', \eta'} \qquad \text{PAR1}\ \frac{P \parallel \rho, \eta \xrightarrow{\alpha} P' \parallel \rho', \eta'}{P|Q \parallel \rho, \eta \xrightarrow{\alpha} P'|Q \parallel \rho', \eta'}$$

$$\text{IF1}\ \frac{l \in tl(P) \qquad P \parallel \rho_1, \eta \xrightarrow{\alpha} P' \parallel \rho_1', \eta'}{P \parallel \textbf{if } l \textbf{ then } \rho_1 \textbf{ else } \rho_2, \eta \xrightarrow{\alpha} P' \parallel \rho_1', \eta'} \qquad \text{IF2}\ \frac{l \notin tl(P) \qquad P \parallel \rho_2, \eta \xrightarrow{\alpha} P' \parallel \rho_2', \eta'}{P \parallel \textbf{if } l \textbf{ then } \rho_1 \textbf{ else } \rho_2, \eta \xrightarrow{\alpha} P' \parallel \rho_2', \eta'}$$

$$\text{SWITCH}\ \frac{P \parallel \eta, 0 \xrightarrow{\tau} P' || \eta', 0}{l : \{P\} \parallel \sigma(l).\rho, \eta \xrightarrow{\tau} P' \parallel \rho, \eta'} \qquad \text{COM}\ \frac{P \parallel \sigma(l_1).0, 0 \xrightarrow{a} P' \parallel 0, 0 \qquad Q \parallel \sigma(l_2).0, 0 \xrightarrow{\bar{a}} Q' \parallel 0, 0}{P|Q \parallel \sigma(l_1, l_2).\rho, \eta \xrightarrow{\tau} P'|Q' \parallel \rho, \eta}$$

Fig. 2. Operational semantics for processes with schedulers

are in Fig. 2. 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 which one is to be regarded as primary.

A process is *blocked* if no transition is possible with any schedulers. Roughly speaking, a single primary or secondary scheduler for a process is nonblocking if it can be paired with any nonblocking secondary or primary scheduler (respectively) and not cause the process to be blocked[4].

4.2 Correspondence Theorem

The main correspondence theorem can now be stated.

Theorem 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.*

[4] Obviously, this would be a circular definition, so in the full paper we define nonblocking first inductively for a secondary scheduler, and then for a primary scheduler, with reference to nonblocking secondary schedulers.

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.

The proof is in the appendix. We begin by stating a procedure to construct a strategy from a scheduler, then we prove that the strategy constructed by this procedure always leads to the same choice of action in the process as the corresponding scheduler. Next we prove that these strategies are always deterministic, complete, and introspective. To prove the other part of the theorem, we give a procedure to translate a deterministic, complete, introspective strategy into a syntactic scheduler.

5 Conclusions

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 which 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.

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 [10]. We have used these information theoretic ideas for an analysis of anonymity [11], indeed it was that investigation that sparked the research reported in [5] and which ultimately led to the present work. As far as we know, the only paper looking at epistemic logic and probability is by Krasucki, Ndjatou and Parikh [12] where they quantify the amount of information shared when agents possess common knowledge.

Acknowledgments We would like to thank Samson Abramsky, Yannick Delbecque and especially Catuscia Palamidessi for many helpful discussions.

References

1. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. In: Proc. of Principles of Distributed Computing. (1984) 50–61
2. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press (1995)
3. Schneider, S., Sidiropoulos, A.: CSP and anonymity. In: Proc. of ESORICS. Volume 1146 of LNCS., Springer (1996) 198–218
4. Bhargava, M., Palamidessi, C.: Probabilistic anonymity. In: Proc. of CONCUR. Volume 3653 of LNCS., Springer (2005) 171–185
5. Chatzikokolakis, K., Palamidessi, C.: Making random choices invisible to the scheduler. In: CONCUR. LNCS 4703 (2007) 42–58
6. Abramsky, S., Jagadeesan, R.: Games and full completeness for multiplicative linear logic. J. Symbolic Logic **59** (1994) 543–574
7. Hyland, J.M.E., Ong, C.H.L.: On full abstraction for PCF. Information and Computation **163** (2000) 285–408
8. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Information and Computation **163** (2000) 409–470
9. Danos, V., Harmer, R.: The anatomy of innocence. Lecture Notes in Computer Science **2142** (2001) 188–202
10. Shannon, C.: A mathematical theory of communication. Bell System Technical Journal **27** (1948) 379–423,623–656
11. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity protocols as noisy channels. In: Proc. of TGC. Number 4661 in LNCS (2006) 281–300

12. Krasucki, P., Ndjatou, G., Parikh, R.: Probabilistic knowledge and probabilistic common knowledge. In: ISMIS 90, North Holland (1990) 1–8

A Proofs

Definition 11. *Given a process P, a valid position s and a strategy S, define $S/s = \{s' | s.s' \in S\}$.*

Definition 12. *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.*

The following is Prop. 1.

Proposition 2. *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, we prove that S is prefix closed. If $s \in S$, then every prefix of s is in $S_1 \cup S_2$ by definition of S. Consider s' an arbitrary prefix of s. Every prefix of s' is also a prefix of s, so every prefix of s' is in $S_1 \cup S_2$. Therefore $s' \in S$ and S is prefix closed.

Now, we prove by induction that S has at most one element of any length, and that S is nonempty.

Base Case: length=0. Since every strategy contains ε, and the only prefix of ε is ε, $\varepsilon \in S$ and ε is the only position of length 0, so S has exactly one position of length 0.

Induction Hypothesis: If there is a position in S of length n, then there is at most one position of each length up to and including n in S.

Induction Step: Case 1: there is no position in S of length $n+1$. Then the IH is vacuously true.

Case 2: there is some position $s.m$ of length $n+1$ in S. Since S is prefix closed, $s \in S$. Since the length of s is n, the IH applies, so s is the only element of S of length n.

Now assume there is some element $t.m'$ of length $n+1$ in S. Since S is prefix closed, $t \in S$, but s is the only element of S of length n, therefore $s = t$. Since a position cannot have both X and Y moves enabled, either $s.m, t.m' \in S_1$, or $s.m, t.m' \in S_2$. In either case, since $s = t$, $m = m'$ by the determinacy of that strategy. Therefore, $s.m$ is the only element of S with length $n+1$, and every prefix of $s.m$ is in S, and these are the only elements of S of length less than $n+1$.

Thus, the induction step is verified in both cases.

The following is the proof of the main theorem Thm. 1 of Section 4.2.

Theorem 2. *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 $P \parallel \rho, \eta$.

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 $l \in enabled(s_1)$, 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 $l \in enabled(s_1)$, the labels will be chosen according subscheduler ρ_1, which corresponds to the strategy $Strat(\rho_1, V/s_1)$ by the induction hypothesis. On the other hand, if $Z(s_1) = \varepsilon$ and $l \notin enabled(s_1)$, 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 $l \in enabled(s_1)$ and s_2 comes from a

recursive call to *Strat* or $s_1.s_2$ where $l \notin enabled(s_1)$, $enabled_Z(s_1) \neq \emptyset$ 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**
        $\vdots$
            if l_k **then** ρ_1
                else ρ_2
        $\vdots$
        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.