Protocols and Programs

Motivation

Starting in some initial global state, what causes the system to change state ?

Intuitively, it is clear that changes occur as a result of *actions* performed by the agents and the environment.

The agents typically perform their actions deliberately, according to some *protocol*.

Protocols are often represented by *programs.*

Programs are designed to satisfy some *specifications*.

We shall describe

- Actions
- Protocols and Contexts
- Programs
- Specifications

We shall illustrate these notions on examples of

- Bit-transmission problem
- Games
- Message-passing systems
- Reliable message-passing systems
- Asynchronous message-passing systems
- Distributed systems

Actions

We already have shown several examples of actions taken by agents in multi-agent systems. For example,

in message-passing systems, the actions include sending and receiving messages and possibly some internal actions performed by agents. So far, we have not considered actions taken by the environment. We shall consider environment as an agent as well,

in games G_1 and G_2, the actions were moves $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{b}_1$ and $\mathbf{b}_2$,

in a distributed system, an action $\mathsf{send}(x, j, i)$ - intuitively corresponding to i sending j the value of variable x. It might be in the set ACT_i of actions of agent i if x is a local variable of i. On the other hand, if x is not a local variable of i, then it would not be appropriate to include $\mathsf{send}(x, j, i)$ in ACT_i.

We take environment as an agent e and we allow it to perform actions from a set ACT_e.

In message-passing systems, it is appropriate to view *message delivery* as an action of environment.

For both the agents and the environment, we allow for the possibility of a special *null* action Λ, which corresponds to the agents or environment performing no action.

Actions performed simultaneously by different agents in a system may interact. To deal with potential interactions between actions, we consider *joint actions*.

A joint action is a tuple $(\mathbf{a}_e, \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$, where $\mathbf{a}_e$ is an action performed by the environment and $\mathbf{a}_i$, is an action performed by agent i.

Recall

L_e set of all possible states of environment

L_i set of all possible states of agent i

$G = L_e \times L_1 \times \ldots \times L_n$ set of all possible global states

Run over G is a sequence r of global states

$r(m) = (s_e, s_1, \ldots, s_n)$ in the run r in the (time) point m.

If $r(m) = (s_e, s_1, \ldots, s_n)$ is a global state in the point $(r, m) = r(m)$, we define projections

$r_e(m) = s_e$ a $r_i(m) = s_i$ for $i = 1, \ldots, n$.

Actions

Example 1. (The bit-transmission problem)

Sender $ACT_S = \{\mathbf{sendbit}, \Lambda\}$

Receiver $ACT_R = \{\Lambda, \mathbf{sendack}\}$

Environment $ACT_e = \{(\mathbf{a},\mathbf{b}) \mid (\mathbf{a}$ is $\mathbf{deliver}_S$ (current) or Λ_S, $\mathbf{b}$ is $\mathbf{deliver}_R$ (current) or $\Lambda_R\}$

For example, if e performs $(\Lambda_S, \mathbf{deliver}_R(\text{current}))$ then R receives whatever message S sends in that round (if there is one) but S does not receive any message, and if R did send a message in that round, then that message is lost.

Example 2. (Asynchronous message-passing systems)

In the previous example, the environment could either deliver the message currently being sent by either S or R , or it could lose it altogether.

In the , asynchronous message-passing systems (a.m.p. systems) to be defined later on, the environment has more possible actions e.g. it can decide to deliver a message an arbitrary number of rounds after it has been sent.

It is also useful to think of the environment in an a.m.p. system as doing more than just deciding when messages will be delivered.

Recall that in a.m.p. systems we make no assumption on relative speed of processes. This means that there may be arbitrary long intervals between actions taken by processes. One way to describe this possibility is *to let the environment* to decide when the process is allowed to take an action.

More formally, in an asynchronous message-passing system

we assume that

ACT_e consists of actions $\mathbf{a}_e$ of the form $\mathbf{a}_e = (\mathbf{a}_{e1}, \mathbf{a}_{e2}, \ldots, \mathbf{a}_{en})$, where

$\mathbf{a}_{ei} \cdots \mathbf{deliver}_i(current,\ j)$ ($\approx$ $\mathbf{deliver}_R(current)$)

$\mathbf{deliver}_i(\mu,\ j)$ ($\approx$ i receives μ from j)

$\mathbf{go}_i$ ($\approx$ i is allowed to perform an action)

$\mathbf{nogo}_i$ ($\approx$ i is not allowed to perform an action)

The set ACT_i of possible actions for process i consists of send actions $\mathbf{send}(\mu,\ j)$ and all the internal actions INT_i , where $\mu \in MSG$, and $j \in \{1, 2, \ldots, n\}$.

Here MSG is the set of all possible messages common to all processes.

Recall that we took the state of each process in an a.m.p. system to be its history, and said that the environment's state records the evnts that have taken place, but we did not described the en vironment's state in detail.

We consider *joint actions* $(\mathbf{a}_e, \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$ to deal with possible interactions between actions of different agents. Now, we can take the environment's state to be the sequence of joint actions performed thus far. Hence,

s_e is a sequence of joint actions performed thus far, and

that a history is a sequence starting with an initial state and whose later elements consist of non empty sets of events

$send(\mu,\ j, i)$ corresponds to $\mathbf{send}_i(\mu,\ j)$ by i

$receive(\mu,\ j, i)$ corresponds to $\mathbf{deliver}_i(\mu,\ j)$

$int(\mathrm{a}, i)$ corresponds to $\mathbf{int}(\mathbf{a}, i)$ by i

The transition function

τ simply updates the processes' and the environments states to reflect the actions performed.

Suppose

$$\tau(\mathbf{a}_e, \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)(s_e, s_1, s_2, \ldots, s_n) = (s`_e, s`_1, s`_2, \ldots, s`_n)$$

where $\mathbf{a}_e = (\mathbf{a}_{e1}, \mathbf{a}_{e2}, \ldots, \mathbf{a}_{en})$

then $(s`_e, s`_1, s`_2, \ldots, s`_n)$

must satisfy the following constraints, for $i = 1, 2, \ldots, n$.

- $s`_e$ is the result of appending $(\mathbf{a}_e, \mathbf{a}_1, ..., \mathbf{a}_n)$ to s_e
- if $\mathbf{a}_{ei} = \mathbf{go}_i$ and $\mathbf{a}_i$ is an internal action or send action **send**$_i(\mu, j)$, then $s`_i$ is the result of appendindg the event corresponding to $\mathbf{a}_i$ to the history s_i.
- if $\mathbf{a}_{ei} = \mathbf{deliver}_i(\text{current}, j)$, $\mathbf{a}_{ej} = \mathbf{go}_j$ and $\mathbf{a}_j$ is $\mathbf{send}(\mu, i)$, then $s`_i$ is the result of appending $receive(\mu, j, i)$ to s_i.
- if $\mathbf{a}_{ei} = \mathbf{deliver}_i(\text{current}, j)$, then $s`_i$ is the result of appending $receive(\mu, j, i)$ to s_i.
- in all other cases, $s`_i = s_i$.

Notice how the joint actions in a joint tuple interact. For example,

- unless $\mathbf{a}_{ei} = \mathbf{go}_i$, the effect of $\mathbf{a}_i$ is nullified, and
- in order for a message sent by j to i to be received by i in the current round, we must have both $\mathbf{a}_{ej} = \mathbf{go}_j$ and $\mathbf{a}_{ej} = \mathbf{deliver}_i(current, j)$.

We choose message delivery to be completely under control of environment. We could instead assume that when the environment chooses to deliver a message from i to j, it puts it into a buffer (which is a component of its local state). In theis case, i would receive a message only if it actually performed a **receive** action. We have chosen the simpler way of modelling message delivery, since it suffices for our examples.

These examples should make it clear how much freedom we have in choosing how to model a system.

The effect of a joint action will be very dependent on our choice.

Example 1 (continued).

In the bit-transmission problem, we choose to record in the local state of S only wheter or not S has received an *ack* message and not how many *ack* messages S receives. The delivery of an *ack* message may have no effect on S' state. If we had chosen instead to keep track of the number of messages S received, then every message delivery would have caused a change in S's state.

Ideally, we choose a model that is rich enough to capture all the relevant details, but one that make s it easy to represent state transitions.

Example 2. (continued)

This example shows, that if we represent a process's state in an a.m.p. system by its history, modelling the effect of joint action becomes quite straightforward.

Protocols and Contexts

Agents usually perform actions according to some *protocol*, which is a rule for selecting actions. For example in the bit-transmition problem , the receiver's protocol involves sending an *ack* message after it has received a bit from the sender.

Intuitively, a *protocol* for agent i is a description what actions agent i may take, as a function of her local state. We formally define a protocol P_i for agent i as follows:

Definition. (i) a *protocol* P_i for agent i is a function from the set L_i of agent´s local states to non-empty sets of actions in ACT_i.

The fact that we consider a set of possible actions allows us to capture the possible nondeterminism of the protocol. Of course, at a given step of the protocol, only one of these actions is actually performed; the choice of action is nondeterministic.

(ii) A *deterministic protocol* P_i maps states to actions (not to subsets of ACT_i) . We write $P_i(s_i) = \{\mathbf{a}\}$ for each local state s_i in L_i . If P_i is deterministic, then we write simply $P_i(s_i) = \mathbf{a}$.

(iii) It is also useful to view the environment e as running a protocol.

We define a protocol for the environment P_e to be a function from L_e to nonempty subsets of ACT_e.

For example, in a message-passing system (see Example 1), we can use the environment's protocol to capture the possibility that messages are lost or that messages may be delivered out of order.

In most of our examples, the agents follow deterministic protocols, but the environment does not.

Remark. While our notion of protocol is quite general, there is a crucial restriction: a *protocol* is a function on *local* states, rather than a function on *global* states. This captures our intuition that all the information that the agent has is encoded in his local state, and not on the whole global state.

Thus what an agent does can depend only on his local state, and not on the whole global state.

In the definition, we allow protocols that are arbitrary (set-theoretical) functions. In practice, we are interested in *computable* protocols.

Joint Protocols

Processes do not run their protocols in isolation.

We define a *joint protocol* P to be a tuple $(P_1, P_2, \ldots, P_n)$ consisting of protocols P_i for each agent.

Comments. Note that, in contrast to joint actions joint protocol does not include the protocol P_e of the environment. This is because of the environment's special role: we usually design the agent's protocols, taking the environment's protocol as given.

In fact, when designing multi-agent systems, the environment is often seen as an *adversary* who may be trying to force the system to behave in some undesirable way.

In other words the joint protocol P and the environment's protocol P_e can be viewed as the strategies of opposing players.

The joint protocol P and the environment's protocol P_e prescribe the behaviour of all „participants" in the system and therefore, intuitively, should determine the complete behaviour of the system.

On closer inspection, the protocols describe only the actions taken by the agents and the environment. To determine the behaviour of the system, we also need to know the "context" in which the joint protocol (of the agents) is executed.

What does such a context consist of ?

• Clearly, the environment's protocol P_e should be part of the context, since it determines the environ-ment's contribution to the joint actions.

• In addition, the context should include the tra nsition function τ, because it is τ that describes the results of the joint actions.

• Furthermore, the context should contain the set G_0 of *initial* global states, because this describes the state of the system when execution of the protocol begins.

These components of the context provide us with a way of describing the environment's behaviour at any single step of an execution.

There are times when we wish to consider more global constraints on the environment's behaviour, ones that are not easily expressible by P_e, τ, and G_0.

To illustrate this point, recall from Example 2. that in an a.m.p. system, we allow the environment to take actions of the form $(\mathbf{a}_{e1}, \ldots, \mathbf{a}_{en})$, where $\mathbf{a}_{ei}$ is one of $\mathbf{nogo}_i$, $\mathbf{go}_i$, $\mathbf{deliver}_i(current, j)$, or $\mathbf{deliver}_i(\mu, j)$.

In an a.m.p. system, we can think of the environment's protocol as prescribing a nondeterministic choice among these actions at every step, subject to the requirement that a message is delivered only if it has been sent earlier but not yet delivered.

Now suppose we consider an a.r.m.p. system, where all message delivery is taken to be reliable. Note that this does not restrict the environment's actions in any given round.

The most straightforward way to model an a.r.m.p. system is to leave the environment's protocol unchanged, and place an additional restriction on the acceptable behaviour of the environment. Namely, we require that all messages sent must be delivered by the environment.

There are a number of ways that we could capture such a restriction on the environment's behavour. Perhaps the simplest is to specify a condition Ψ on runs, one that tells us which ones are "acceptable" .

Definition. (The set of acceptable runs)

Let R be a system and Ψ be a condition defining a subset of R. We say that Ψ is a *condition* defining the set of acceptable runs. Namely, $r \,\varepsilon\, \Psi$ if r satisfies the condition Ψ.

Notice that while the environment's protocol can be thought of as describing a restriction on the environment's behaviour at any given point in time, the reliable delivery of messages is a restriction on the environment's "global" behaviour, namely, on the acceptable (possibly infinite) behaviours of the environment.

Indeed, often the condition Ψ can be characterized by one (or a collection of) formulas of temporal logic, and the runs in Ψ are those that satisfy these formulas.

For example, to specify reliable message-passing systems, we could use the condition

$Rel = \{ r \mid$ all messages sent in r are eventually received$\}$

Recall that $send(\mu, j, i)$ is the event (we may take it as a propositional formula) that is interpreted to mean "message μ is sent to j by i "

and let $receive(\mu, i, j)$ be a proposition that is interpreted to mean "message μ is received from i by j ". Then a run r is in Rel precisely if

$$\Box(send(\mu, j, i) \;\text{->}\; \Diamond\, receive(\mu, i, j))$$

holds at $(r, 0)$ (and thus at every point in r) for each message μ and processes i, j.

Another condition of interest is $True$, the condition accepting all runs; this is the appropriate condition to use if we view all runs as "good".

Definition. (Context)

Formally, we define a context γ to be a tuple (P_e, G_0, τ, Ψ), where P_e is a protocol mapping the set L_e of the environment's local states to nonempty subsets of ACT_e, G_0 is a nonempty subset of G, τ is a transition function, and Ψ is a condition on runs.

There are a number of ways that we could capture such a restriction on the environment´s behaviour. One of them is to specify a condition ψ on runs that tells us which ones are „acceptable".

Set of acceptable runs

ψ is a set of runs usually defined by a condition: r belongs to ψ if r satisfies the condition ψ**.** Often the condition ψ can be characterized by one or more temporal formulas.

Example. (Reliable message-passing systems)

For ψ we could use the condition *Rel*, where

$Rel = \{ r \mid$ all messages sent in the round m are eventually received$\}$

A run r is in Rel iff $\Box$(send(μ, j, i) => $\Diamond$*receive*(μ, i, j))

holds at $(r, 0)$ (and thus at every point in r) for each message μ and processes i, j.

Comments. Notice that by including τ in the context we implicitely include the domain of τ i.e. $L_e \times L_1 \times ... \times L_n$ as well as the range of τ consisting of the set ACT which is the product of the set ACT_e of the environment's actions and the sets $ACT_1, ..., ACT_n$ of actions of the processes, since the domain of τ is the set ACT and the set of global states is the domain of the transition functions yielded by τ.

To minimize notation, we do not explicitely mention the sets of states and the sets of actions in the context. (We shall, however, to refer to these sets and to the set $G = L_e \times L_1 \times ... \times L_n$ of global states as if they were part of the context.)

As we shall see later on, the combination of a context γ and a joint protocol P for the agents uniquely determines a set of runs, which we shall think of as the system representing the execution of the joint protocol P in the context γ.

As we shall see later on, the combination of a context γ and a joint protocol P for the agents uniquely determines a set of runs, which we shall think of as the system representing the execution of the joint protocol P in the context γ.

Contexts provide us with a formal way to capture our assumptions about the systmes under consideration. We give two examples of how it can be done here; many others appear in the next parts of this presentation.

Examples 1. and 2. (continued)
In the bit-transmition problem and asynchronous m.p.s. systems, we assumed that the environment keeps track of the sequence of joint actions that were performed. We can formalize this in terms of recording contexts.

Definition. (Recording context)

We say that $(P_e, \boldsymbol{G}_0, \tau, \psi)$, is a *recording context* if the following holds:

- the environment\`s state is of the form { ... h ... }, where h is a sequence of joint actions
- in all global states in $\boldsymbol{G}_0$ the sequence h is empty (so that no actions have been performed initially)
- if $\tau(\mathbf{a}_{ei}, \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)(s_e, s_1, s_2, \ldots, s_n) = (s\grave{}_e, s\grave{}_1, s\grave{}_2, \ldots, s\grave{}_n)$

then we require that the sequence h' that occurs in s'_e is obtained from s_e by appending $(\mathbf{a}_e, \mathbf{a}_1, \ldots, \mathbf{a}_n)$ to the corresponding sequence h.

Example 3. (message-passing systems)

In a message-passing system, fix a set Σ_i of initial states for process *i,* a set INT_i of internal actions for i, and a set MSG of messages.

Definition. A context $(P_e, \boldsymbol{G}_0, \tau, \psi)$ is a *message-passing context* if

- process $i's$ actions are sets consisting of elements of the form **send**(μ, j) or **a** for **a** in INT_i, μ in MSG and for $i = 1, 2, \ldots, n$.
- process $i's$ local states are histories.
- for every global state $(s_e, s_1, s_2, \ldots, s_n)$ in $\boldsymbol{G}_0$ we have s_i in Σ_i for all processes.
- if $\tau(a_{ei}, a_1, a_2, \ldots, a_n)(s_e, s_1, s_2, \ldots, s_n) = (s\grave{}_e, s\grave{}_1, s\grave{}_2, \ldots, s\grave{}_n)$, then we require that either $s'_i = s_i$ or s'_i is obtained from s_i by appending the set consisting of events corresponding to the above actions and events that corresponding to messages sent earlier to i by some process j.

Comment.

- Intuitively, the state s'_i is the result of appending to s_i the additional events that occured from process $i's$ point of view in the most recent round.
- These consist of the actions performed by i, together with the messages received by i.
- We allow $s'_i = s_i$ to accomodate the possibility that the environment performs a $\mathbf{nogo}_i$ action.
- Note that we have placed no restrictions on P_e, ψ, or the form of the environments states here, although in practice we often take message-passing contexts to be recording contexts as well.
- In practice, as we shall see later on, this is the context that capture a.m.p. systems.

In many cases we have a particular collection Φ of primitive propositions and a particular thterpretation π for Φ over G in mind when we define a context. Just as we went from systems to interpreted systems, we can go from contextes to *interpreted contexts.*

Definition. (Interpreted contexts)

An *interpreted context* is a pair (γ, π) where γ is a context and π is an interpretation.

(we do not explicitely include Φ here, just as we did not in the case of interpreted systems and Kripke structures.)

Comments. Recall that an interpretation π is a mapping that gives a boolean value to each of the basic formulas in the set Φ in every point of run.

Similarly, as we had some flexibility in describing the global states, we often some have flexibility in desribing the other components of a context.

We typically thin k of G_0 as describing the initial conditions, while τ and P_e describe the system's local behaviour, and ψ describes all other aspects of the environment's behaviour.

To describe the behaviour of the system we have to decide what actions performed by the environments are (this is a part of P_e) and how these actions interact with the actions of the agents (this is described by τ .

There is often more than one way in which this can be done. For example we choose earlier to model message delivery by an explicit **deliver** action by the environment rather as the direct result of a **send** action by the agents.

Although we motivated the condition ψ by the need to be able to capture global aspects of the enviroment's behaviour, we have put no constraints on the condition ψ. As a result it is possible (although not advisable) to place much of the burden of determining the initial codnitions and local behaviour of the environment on the condition ψ.

Thus, for example, we could do away with G_0 altogether, have ψ consist only of runs r whose initial state would ha been in G_0. In well-chosen context we would expect the components to be orthogonal.

In particular, we expect that P_e would specify local aspects of the environment's protocol, while ψ would capture the more global properties of the environment's behaviour over time (such as "all messages are eventually delivered").

However, there are times, when we need to made further restrictions on the condition ψ to ensure that it does not interact with the other coponents of the context in undesired ways. We shall see an exaple of it later.

We can now talk about the runs of protocol in a given context.

Definition. (Consistent runs)

We say that a *run* r *is consistent with a joint protocol* $P = (P_1, \ldots, P_n)$ *in the context* $\gamma = (P_e, G_0, \tau, \psi)$ if

- $r(0)$ is in G_0 (so $r(0)$ is a legal initial state)
- for all $m \geq 0$, if $r(m) = (s_e, s_1, \ldots, s_n)$, then there is a joint action $(\mathbf{a}_e, \mathbf{a}_1, \ldots, \mathbf{a}_n)$ in $P_e(s_e) \times P_1(s_1) \times \ldots \times P_n(s_n)$ such that $r(m+1) = \tau(\mathbf{a}_e, \mathbf{a}_1, \ldots, \mathbf{a}_n)(r(m))$ so $r(m+1)$ is the result of transforming $r(m)$ by a joint action that could have been performed from $r(m)$ according to P and P_e), and
- r is in ψ (so that, intuitively, r conforms to the restrictions made by ψ).

Comment. The first condition says that $r(0)$ is a legal initial state. The second one states that $r(m+1)$ is obtained from $r(m)$ by a joint action that could have been performed according to P and P_e. The third condition requires that r conforms with the restriction of ψ .

Definition. (Weakly consistent runs)

We say that r is *weakly consistent* with P in context γ, if it satisfies only the first two conditions of the three conditions of consistency, but is not necessarily in ψ .

Comments

- Intuitively this means that r is consistent only with the step-by-step behaviour of P.
- Note that there are always runs weakly consistent with P (in the context γ), it is also possible that there is no run that is consistent with P in context γ.
- This happens iff there is no run in ψ weakly consistent with P.

Definition. (Consistent systems)

We say that a system R (respectively an interpreted system $I = (R, \pi)$ *is consistent with a protocol* P *in context* γ (resp. interpreted context (γ, π)) if every run r in R is consistent with P in γ.

Comment.

Because systems are nonempty sets of runs, this requires that P be consistent with γ. Typically, there will be many systems consistent with a protocol in a given context. However, when we think of running a protocol, we usually have in mind the system where all possible behaviours of the protocol are represented.

Definition. (Representing systems)

(i) We define $\mathbf{R}^{rep}(P, \gamma)$ to be the system consisting of all runs consistent with P in context γ. We call it the *system representing protocol* P *in context* γ.

(ii) Similarly, we say that $\mathbf{I}^{rep}(P, \gamma, \pi) = (\mathbf{R}^{rep}(P, \gamma), \pi)$ is the *interpreted system representing* P *in the interpreted context* (γ, π).

Comments.

• Note that R is consistent with P in γ iff R is a subset of $\mathbf{R}^{rep}(P, \gamma)$. so $\mathbf{R}^{rep}(P, \gamma)$ is the maximal system consistent with P and γ.

• While we are mainly interested in the (interpreted) system representing P in a given (interpreted) context, there is a good reason to look at some of the other systems consistent with P in that context as well.

We may start out considering one context γ, and then be interested in what happens if we restrict attention to a particular set of initial states, or may wish to see what happens if we limit the behaviour of the environment.

The following definitions make precise the idea of "restricting" a context γ.

Definition. (Restricting contexts)

(i) We say that the environment protocol P_e' is a restriction of P_e written $P_e' \lhd P_e$, if $P_e'(s_e) \subseteq P_e(s_e)$ holds for every local state $s_e \in L_e$.

(ii) We say that a context $\gamma' = (P_e', G_0', \tau, \psi')$ is a subcontext of a context $\gamma = (P_e, G_0, \tau, \psi)$ and write $\gamma' \lhd \gamma$, if $P_e' \lhd P_e$ and G_0' is a subset of G and ψ' is a subset of ψ.

(iii) Similarly, we say that an interpreted context (γ', π) is a subcontext of (γ, π) if γ' is a subcontext of γ.

Lemma.

R is consistent with P in context γ iff R represents P in some subcontext γ'.

Example 1. (continued)

What are the sender's and receiver's protocols in our bit-transmission problem?

Recall that the sender S is in one of four states 0, 1, (0, *ack*), (1, *ack*), and its possible actions are **sendbit** and $\mathbf{\Lambda}$. Its protocol P_S^{bt} is quite straghtforward to describe

- $P_S^{bt}(\lambda) = P_S^{bt}(1) =$ **sendbit**
- $P_S^{bt}(0, ack) = P_S^{bt}(1, ack) = \mathbf{\Lambda}$

Recall that the receiver is in one of three states: λ, 0, or 1 and its possible actions are **sendack** and $\mathbf{\Lambda}$. The receiver's protocol is

- $P_R^{bt}(\lambda) = \mathbf{\Lambda}$
- $P_R^{bt}(0) = P_R^{bt}(1) =$ **sendack**

We now need to describe a context for the joint protocol $P^{bt} = (P_S^{bt}, P_R^{bt})$. Recall that

- the environment's state is a sequence recording the events taking place in the system , and
- the enviroment's four actions are of the form (**a , b**), where **a** is either **deliver**$_S$(*current*) or Λ_S , while **b** is either **deliver**$_R$(*current*) or Λ_R.

We view the environment as running the nondeterministic protocol P_e^{bt}, acording to which, at every state, it nondeterministically chooses to perform one of these four actions.

The set G_0 of initial states is the product {<>}x {0, 1}x {λ} i.e. initially the environment´s and receiver´s state record nothing, and the sender´s state records the input bit.

The context capturing the situation described in Example 1. is

$$\gamma^{bt} = (P_e^{bt}, G_0, \tau, True)$$

Moreover, the system R^{bt} described in Example 1 , is exactly

$$R^{bt} = \mathbf{R}^{rep}(P^{bt}, \gamma^{bt})$$

We may want to restrict the environment and the context such that the system´s communication channel is *fair* in the sense that every message sent infitely often is eventually delivered. Thus, a run is in *Fair* if it satisfies the formula

$$(\Box\text{<>}\ sendbit \Rightarrow \text{<>}recbit)\ \&\ ((\Box\text{<>}\ sendack \Rightarrow \text{<>}\ recack)$$

Let γ^{bt}_{fair} be the context we get by replacing *True* in γ^{bt} by *Fair*. The system $\mathbf{R}^{fair}$ that represents P_e^{bt} in a fair setting is then

$$\mathbf{R}^{rep}(P^{bt}, \gamma^{bt}_{fair})$$

Example 4. (asynchronous m.p.s.)

Consider a.m.p. system over $\Sigma_1, \ldots, \Sigma_n$, $INT_1, \ldots, INT_n$ and *MSG*. As we now show, these can be characterized by the context (P_e^{amp}, $\mathbf{G}_0$, τ, *True*). This context is both a recording context and a message-passing context.

All we need to do to complete its description is to describe the environment`s actions and local states:

- since it is a recording context, the environment`s states must include the sequence of joint actions performed thus far. In fact, here we take the environment's state to be precisely this sequence.
- $\mathbf{G}_0$ is {<>} x Σ_1 x … x Σ_n ,
- we discussed earlier how the trasition function τ is defined ,
- finally, P_e^{amp} simply nondeterministically chooses one of the environment`s actions , except that if **deliver**$_i$(μ , j) is performed, then the message μ must have been sent earlier by i to j , and not yet received.

(Note that the environment *e* can determine this just by looking at its state, since its state records the sequences of actions performed thus far.)

- Call this context γ^{amp}.

In what sense does γ^{amp} characterize a.m.p. systems?

Suppose that we are given a prefix-closed set V_i of histories for process i. We show that V_i determines a protocol P_i for process i.

- If h is in V_i and **a** is in ACT_i, let $h \bullet a$ denote the history that results from appending to h the event a corresponding to the action **a**.
- We then define $P_i(h)$ = {**a** | **a** belogs to ACT_i and $h \bullet a$ to V_i }. Intuitively, $P_i(h)$ consists of all allowable actions according to the set V_i.
- Let $P^{amp}(V_1, \ldots, V_n)$ be the the joint protocol that corresponds to the sets $V_1, \ldots, V_n$ of histories.

It is not hard to show that

$$R(V_1, \ldots, V_n) \text{ is essentially } \mathbf{R}^{rep}(P^{amp}(V_1, \ldots, V_n), \gamma^{amp}) \quad (1)$$

Comments.

(i) it follows from (1) that the right way to think about the a.m.p. system

$$R(V_1, \ldots, V_n)$$

is as the system that results when the processes run the joint protocol

$$P^{amp}(V_1, \ldots, V_n)$$

(ii) Notice, that if we wanted to consider *asynchronous reliable message passing systems (a.r.m.p. systems)* rather than a.m.p. systems, we would simply replace the condition *True* in the context γ^{amp} by the condition *Rel.*

We may also want to require that the environment follows a *fair schedule,* in the sense that no process is blocked from moving from some point on.

Definition. (Faire Schedule)

Formally, we can capture that the environment runs a *faire schedule* by the condition *FS* that holds of a run if there are infinitely many $\mathbf{go}_i$ actions for each process i.

Thus if we add a proposition go_i that is true at a state exactly if a go_i was performed by the environment in the preceding round, then the condition *FS* can be characterized by the formula

$$\Box <> go_1 \ \& \ \Box <> go_2 \ \& \ \ldots \& \ \Box <> go_n$$

Example 5. Games.

Let us reconsider the game-theoretic framework. There, we described systems that model all possible plays of a game by including a run for each path of the game.

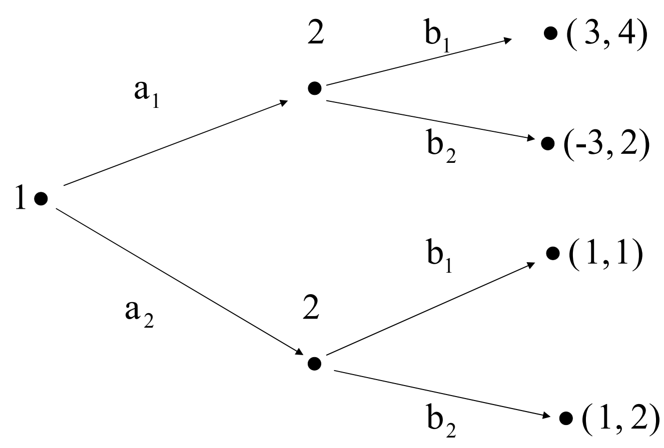
We did not attempt to model the *strategies* of the players, which are the major focus in the game theory.

A *strategy* is a function that tells a player which move to choose based on the player's "*current information*" about the game.

In our model, a player's current information is completely captured by his local state; thus a strategy for player i is simply a deterministic protocol for player i, i.e. a function from his local state to actions.

Let us consider

Game 1



What are the possible strategies for the player 1 in the game G_1 ?

Because player 1 takes an action only at the first step, he has only two possible strategies: "choose $\mathbf{a_1}$" and "choose $\mathbf{a_2}$".

We call these strategies σ_1 and σ_2.

Player 2 has four strategies in G_1, since her choice of actions can depend on what player 1 did.

These strategies can be described by the pairs

. $(\mathbf{b_1\, b_1}), (\mathbf{b_1\, b_2}), (\mathbf{b_2\, b_1}), (\mathbf{b_2\, b_2})$

The first strategy correspodns to "choose $\mathbf{b_1}$ no matter what". The second one corresponds to "choose $\mathbf{b_1}$ if the player 1 choose $\mathbf{a_1}$ and choose $\mathbf{b_2}$ if player 1 choose $\mathbf{a_2}$" etc.

Call these strategies $\sigma_{11}\, \sigma_{12}\, \sigma_{21}\, \sigma_{22}$. Note that while there are eight pairs of strategies (for the two players), there are only four different plays.

For example the pair (σ_1, σ_{11}) and the pair (σ_1, σ_{12}) result in the same play.

Recall that the system R_1 cooresponding to G_1, contains four runs, one run for each path in the game e.g. the local state of both players at the start is the empty sequence $< >$ and their local state after player 1 chooses $\mathbf{a_1}$ is the sequence $< \mathbf{a_1} >$.

We would like to define the protocols for the players that capture the strategies that they follow. However, there is a difficulty. After player 1 chooses $\mathbf{a_1}$ player's 2 local state is $< \mathbf{a_1} >$. Thus, a deterministic protocol would tell player 2 to choose either $\mathbf{b_1}$ or $\mathbf{b_2}$. But in R_1, player 2 chooses $\mathbf{b_1}$ in one run and $\mathbf{b_2}$ in another.

Does it mean that player 2 does not follow a deterministic protocol ?
No. Rather it means that our description of his local state is incomplete.

We now present a system R_1' that enriches the player's local states so that they include not only the history of the game, but also a representation of the strategy of the player.

Thus, the set of local states of player 1 includes the states such as $(\sigma_1, < >), (\sigma_1, < \mathbf{a_1}, \mathbf{b_1}, >), (\sigma_2, < \mathbf{a_2} >)$ etc.

Similarly, the set of local states of player 2 includes states such as $(\sigma_{11}, < >), (\sigma_{12}, < \mathbf{a_2} >), (\sigma_{21}, < \mathbf{a_1}, \mathbf{b_1} >)$ etc.

Again all the relevant informaton in the system is described by the player's local states, we can take the environment's state to be constantly $\boldsymbol{\lambda}$.

There are eight initial states to all pairs of strategies, so G_0 consists of these eight states.

The actions of the players are $\mathbf{a_1}$, $\mathbf{a_2}$, $\mathbf{b_1}$, $\mathbf{b_2}$, and $\mathbf{\Lambda}$. The environment plays no role here. Its only action is $\mathbf{\Lambda}$, that is $P_e(\lambda) = \mathbf{\Lambda}$. We take $\boldsymbol{\tau}$ as an excercise.

The context $\gamma = (P_e , G_0, \tau, True)$ describes the setting in which the game is played.

We can now define the protocols for the player's according to their strategy. These protocols essenitially say "choose an action according to your strategy.

The protocol P_1 for player 1

- $P_1 (\sigma_i , < >) = \mathbf{a_i}$ for $i = 1, 2,$
- $P_1 (\sigma_i , h) = \mathbf{\Lambda}$ if h is not < >, for $i = 1,2$.

The protocol P_2 for player 2

- $P_2 (\sigma_{ij}, < \mathbf{a_1} >) = \mathbf{b_i}$ for $i, j = 1, 2,$
- $P_2 (\sigma_{ij}, < \mathbf{a_2} >) = \mathbf{b_j}$ for $i, j = 1, 2,$
- $P_2 (\sigma_{ij}, < h >) = \mathbf{\Lambda}$ if h is neither $< \mathbf{a_1} >$ nor $< \mathbf{a_2} >$ for $i, j = 1, 2,$

The system R_1' consists of all runs that start from initial state and are consistent with the joint protocol

$$P = (P_1 , P_2) \quad \text{i.e.} \quad R_1' = \mathbf{R}^{rep}(P , \gamma)$$

So far we described player's local states only in terms of their history. We left out one important point that player's may have their strategies.

In *Game theory* the player's strategies are a focus.

The approach to modeling game trees just dicussed, where player's local states contain information about what strategy the player is using is somewhat more complicated.

It does, however, offer some advantages.

Because it captures the strategies used by the player's, it enables us to reason about what players know about each other's strategies, an issue of critical importance in game theory.

For example, a standard assumption made in the game theory literature is that *players are rational.* To make this precise, we give the following definition.

Definition. (Dominating strategy)

(i) We say that *a strategy* σ *for player* i *dominates a strategy* σ' if, no matter what strategy the other players are using, player i gets at least as high a payoff using strategy σ as using strategy σ' ,

(ii) we say that a strategy σ for player i *strictly dominates a strategy* σ' if it dominates the strategy σ' and there is some strategy that the other players could use whereby i gets a *strictly* higher payoff by using σ'.

Definition. (a rational player)

(i) According to one notion of rationality *a rational player* never uses a strategy if there is another strategy that dominate it.

(ii) We introduce two propositions $rational_i$ for $i = 1, 2$, where $rational_i$ holds at a point if player's i 's strategy at that point is not dominated by another strategy.

Comment. For player 1 to know that player 2 is rational means that $K_1(rational_2)$ holds.

The players can use their knowledge of rationality to eliminate certain strategies.

Example 6. Game G_1

In game G_1, strategy σ_{12} dominates all other strategies for player 2 , so if player 2 were rational, than she would use σ_{12} .

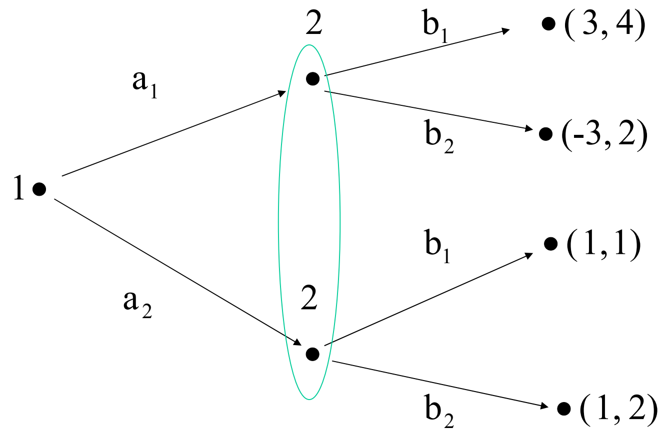
If player 1 knows that player 2 is rational, then he knows that she would use the strategy σ_{12}. With this kowledge, σ_1 dominates σ_2 for player 1.

Thus if player 1 is rational, he would then use σ_1.

It follows that if both players are rational, and player 1 knows that player 2 is rational, than their joint strategy must be (σ_1 , σ_{12}) and the payoff is (3, 4).

Note that if player 1 thinks that player 2 is not rational, it may make sense for 1 to use σ_2 instead , since it guarantees a better payoff in the worst case.

Game 2



How does the game G_2 get modeled in this more refined approach?

- Again, player 1 has two possibile strategies, σ_1, and σ_2.
- But now player 2 also has two strategies, which we call σ_1' and σ_2'.
- Running σ_1', player chooses action b_1, and running σ_2', she chooses b_2. There is no strategy corresponding to σ_{12}, since player 2 does not know what action player 1 performed at the first step, and thus her strategy cannot depend on this action.
- We can define a system R_2' that models this game and captures player's strategies.

By way of contrast, even if we assume that rationality is common knowledge in the game G_2, (assumption that is frequently made by game theorists), it is easy to see that neither players 1 nor 2 has a dominated strategy, and so no strategy for either player is eliminated because of rationality assumption.

Comment.

The above examples show how we can view a context as a description of a class of systems of interest.

The context describes the setting in which a protocol can be run, and running distinct protocols in the same context we generate different systems, all of which share the characteristics of the underlying context.

Programs

We now describe a simple programming language, which is still rich enough to describe protocols, and whose syntax emphesizes tha fact that an agent performs actions based on the result of a test that is applied to her local state.

A (*standard*) *program* Pg_i for agent i is a statement of the form:

case of
  if t_1 **do** a_1
  if t_2 **do** a_2
  …
end case

where t_i are *standard tests* for agent i and a_j are actions of agent i i.e. a_j belongs to ACT_i.

We omit the **case** statement if there is only one clause.

We call such programs *"standard"* to distinguish them from the *"knowledge based"* programs to be introduced later on.

A standard test for agent i is simply a propositional formula over a set Φ_i of primitive propositions.

Intuitively, once we know how to evaluate tests in the program at the local states L_i, we can convert this program to a protocol over L_i.

At a local state ℓ, agent i nondeterministically chooses one of the (possibly infinitely many) clauses in the **case** statement whose test is true at ℓ, and executes the corresponding action.

Compatible interpretations.

We want to use an interpretation π to tell us how to evaluate tests.

We intend the tests in a program for agent i to be *local*, i.e. to depend only on agent i's local state.

It would be inappropriate for agent's i's action to depend on the truth value of a test that i could not determine from her local state.

Definition. (Compatible interpretations)

(i) We say that an interpretation π on the global states in $\boldsymbol{G}$ is *compatible* with a program Pg_i for agent i if every proposition that appears in Pg_i is local to i which means that, if q appears in Pg_i, for any two states s and s' in $\boldsymbol{G}$, such that $s \sim_i s'$, we have

$$\pi(s)(q) = \pi(s')(q)$$

(ii) If A is a propositional formula all of whose primitive propositions are local to agent i, and ℓ is a local state of agent i, then we write

$$(\pi, \ell) \models A$$

if A is satisfied by the truth assignment $\pi(s)$, where $s = (s_e, s_1, \dots, s_n)$ is the global state $s = \ell$.

Comment. Because all the primitive proposition in Φ are local to i, it does not matter which global state s we choose, as long as i's local state in s is ℓ.

Definition. (Interpreted Protocols)

Given a program Pg_i for agent i and an interpretation π compatible with Pg_i, we define a protocol that we denote $Pg_{i\pi}$ by setting

$$Pg_{i\pi}(\ell) = \begin{cases} \{ a_j \mid (\pi, \ell) \models t_j\} & \text{if } \{ j \mid (\pi, \ell) \models t_j\} \text{ is nonempty} \\ \{\Lambda\} & \text{otherwise} \end{cases}$$

Comment. Intuitively, $Pg_{i\pi}$ selects all actions from the clauses that satisfy the test, and selects the null action if no test is satisfied.

In general, we get a nondterministic protocol, since more than one test may be satisfied at a given state.

Many of the definitions for protocols have natural analogues for programs.

Definition (Joint Programs)

We define a *joint program* to be a tuple

$$Pg = (Pg_1, \dots, Pg_n)$$

where Pg_i is a program for agent i.

We say that an interpretation π is *compatible* with Pg if π is compatible with each Pg_i, $i = 1, 2, \dots, n$.

From Pg and π we get a joint protocol

$$Pg_\pi = (Pg_{1\pi}, \dots, Pg_{n\pi})$$

Definition. (Representing Interpreted Systems)

We say that an interpreted system $\mathbf{I} = (R, \pi)$ *represents* (resp., is *consistent with*) a joint program Pg in the interpreted context (γ, π) iff π is compatible with Pg and $\mathbf{I}$ represents (resp., is consistent with) the corresponding protocol Pg_π.

We denote the interpreted system representing Pg in (γ, π) by $\mathbf{I}^{rep}(Pg, \gamma, \pi)$.

Comment. Of course, this definition only make sense if π is compatible with Pg. From now on we always assume that this is the case.

Notice that *the syntactic form* of our standard programs is in many ways more restricted than that of programs e.g. in C or FORTRAN.

In such languages, one typically sees constructs as **for**, **while,** or **if**…**then**…**else,** which do not have syntactic analogues in our formalism.

The semantics of programs containing such constructs depends on the local state containig *instruction counter*, specifying the command that is about to be executed at the local state (of computation).

Since we model the local state of a process explicitely, it is possible simulate these constructs in our framework by having an explicite variable in the local state accounting for the instruction counter.

The local tests t_j used in a program can then reference this variable explicitely, and the actions a_j can include explicite assignments to the variable.

Given that such simulation can be carried out in our framework, there is no loss of generality in our definition of *standard programs.*

It is easy to see that every protocol is induced by a standard program if we have a rich enough set of primitive propositions.

As a result, our programming language is actually more general than many other languages; a program may induce a non-computable protocol.

However, we are interested in programs that induce computable protocols.

In fact, standard programs *usually* satisfy a stronger requirement; they have finite descriptions, and they induce deterministic protocols.

Let us return to the bit-tranmission problem . We saw earlier the sender's protocol.

The sender S can be viewed as running the following program BT_S,

if $\neg$ *recack* **do sendbit**

(Note that if *recack* or $\neg$ recbit holds, then, according to our definitions, the action Λ is selected .)

Similarly, the receiver R can be viewed as running protocol BT_R

if *recbit* **do sendack**

Let $BT = (BT_S, BT_R)$. Recall that we gave an interpretation π^{bt} describing how the propositions in BT_S and BT_R are to be interpreted.

It is easy to see that π^{bt} is compatible with BT, and that $BT^{\pi bt}$ is the joint protocol P^{bt} described in the paragraph on consistent contexts.

Specifications.

Motivation. When designing or analyzing a multi-agent system, we typically have in mind some property that we want the system to satisfy. Very often we start with a desired property and then design a protocol to satisfy this property.

For example, in the bit-transmition problem the desired property is that the sender communicates the bit to the receiver.

We call this desired property the *specification* of the system or protocol under consideration. A specification is typically given as a description of the "good " systems.

Thus, a specification can be identified with a class of interpreted systems, the ones that are "good ".

Definition. (Interpreted system satisfying a specification)

An interpreted system **I** *satisfies* a specification σ if it is in the class σ i.e. **I** is in σ.

Comment. Many specifications that arise in practice are of special type that we call *run-based* i.e. a specification given as a property of runs.

Quite often run-based specifications can be desribed by formulas in temporal logic (with no modal operators for knowledge).

Definition. (Run-based Systems)

We say that a system *satisfies a run-based specification* if all its runs do.

Example1. continued (the bit-transmission problem again)

A possible specification for the bit-transmission problem is: "the receiver eventually receives the bit from the sender and the sender eventually stops sending the bit ". This can be expressed as

<> *recbit* & <>□ ¬ *sendbit*

Similarly, the run-based property: "in every round every message sent is delivered or lost " can be expressed as

□((*sendbit* -> (*recbit* v ¬ recbit)) & (*sendack* -> (*recack* v ¬ *recack*))

The truth of this specification can be decided for each run with no consideration of the system in which the run appears.

Knowledge-based Specifications

Motivation. Although run-based specifications arise often in practice, there are reasonable specifications that are not run-based.

Example. (Muddy children puzzle)

The natural specification of the children's behaviour is: "a child says 'Yes' if he knows whetther he is muddy, and says 'No' otherwise".

This specification is given in terms of the children's knowledge, which depends on the whole system and cannot be determined by considering individual runs in isolation.

We view such a specification as a *knowledge-based* specification.

More generally, we call a specification that is expressible in terms of epistemic (and possibly other) modal operators *a knowledge-based* specification.

Unlike run-based specifications, knowledge-based specifications specify properties of interpreted systems.

Definition. (Satisfaction of Knowledge-based Properties)

We say that P *satisfies* σ in the interpreted context (γ, π) (or is *correct with respect to* σ *in* (γ, π)) , if the interpreted system repre-senting P in (γ, π) satisfies σ i.e. , if $\mathbf{I}^{rep}(P, \gamma, \pi)$ is in σ (i.e. in the set of "good" systems).

Often we are interested in the correctness of a protocol with respect not only one but with respect to some collection Γ of contexts. This collection of contexts corresponds to the various settings in which we want to run the protocol.

Typically, the contexts in Γ are subcontexts of a single context γ. We shall consider a stronger concept of correctness.

Definition. (Stronger Correctness)

We say that a protocol P *strongly satisfies* σ in (γ, π) , or that P is *strongly correct with respect to* σ in (γ, π) , if every interpreted system that represents P in a subcontext γ' of γ satisfies σ.

We know that every system is consistent with P in context γ iff it represents P in some subcontext γ' of γ .

Thus, P is strongly correct with respect to σ in (γ , π) iff P is correct with respect to σ in (γ’ , π) for every subcontext γ’ of γ .

There is one important case where correctness and strong correctness coincide: when σ is a run-based specification. This follows from the fact that a system is consistent with a protocol iff it is a subset of the unique system representing the protocol.

In general, correctness and strong correctness do not coincide. If it is the case, one can argue that strong correctness may be too strong a notion.

After all, even if we are interested in proving correctness with respect to certain subcontexts of γ , we are not interested in *all* subcontexts of γ .

In practice, it is often just as easy to prove strong correctness with respect to γ as it is to prove correctness for a restricted set of subcontexts of γ .

As before, all our definitions for protocols have natural analogues for programs.

Definition. (Programs satisfying a specification)

We say that a program Pg (strongly) satisfies σ in an interpreted context (γ , π) if the protocol Pg_{π} (strongly) satisfies σ in the interpreted context (γ , π) .

Example 1. continued (the bit-transmition problem)

Let σ’ be the run-based specification for the bit-transmition problem i.e.

<> *recbit* & <>□ ¬ *sendbit*

Above, we described a standard program BT = (BT_S, BT_R) for this problem. We also described an interpreted context (γ^{bt}, π^{bt}) for BT.

It is easy to see that BT does not satisfy σ’ in (γ^{bt}, π^{bt}) , for there are runs consistent with BT^{π} in γ^{bt} in which the messages sent by S are never received by R.

However, we are often interested in assuming that the communication channel is *fair*. Recall that γ^{bt}_{fair} is obtained by replacing the condition *True* in γ^{bt} by *Fair*.

Thus, γ^{bt}_{fair} differs from γ^{bt} in that it ensures that communication delivery satisfies the fairness condition.

It is not hard to verify that BT does indeed satisfy σ’ in (γ^{bt}_{fair} , π^{bt}).

Since σ’ is a run-based specification, this implies that BT strongly satisfies σ’ as well.

Hence as long as the communication channel is fair, BT works fine.

We can also give a knowledge-based specification for the bit-transmission problem.

Let σ’’ be the knowledge-based specification: „eventually S knows that R knows the value of the bit, and S stops seending messages when it knows that“.

We can express σ'' as

$$<> K_S K_R(bit) \ \& \ \Box(K_S K_R(bit) \ \text{->} \ \neg \ sendbit)$$

Comment. This specification is more abstract than σ', because it does not refer to the manner in which the agents gain their knowledge.

It is easy to see that BT satisfies σ" in $(\gamma^{bt}_{fair}, \pi^{bt})$.

BT, however, does *not* strongly satisfy σ" in this context. There exists a subcontext γ^{bt}_{ck} of γ^{bt}_{fair} such that BT does not satisfies σ" in $(\gamma^{bt}_{ck}, \pi^{bt})$.

To prove this, assume that γ^{bt}_{ck} is the context where it is common knowledge that *S*'s initial value is 1, and the communication channel is fair i.e. γ^{bt}_{ck} is like γ^{bt}_{fair}, except that the only initial state is (λ, 1, λ).

Clearly γ^{bt}_{ck} is a subcontext of γ^{bt}_{fair}. In this context, the sender knows from the outset that the reveiver knows the bit.

Nevertheless, following BT, the sender would send the bit to the recei-ver in the first round, and would keep sending messages until it receives an acknowledgment.

This does not conform to the requirement made in σ" that if *S* knows that *R* knows the bit, then *S* does not send a message. It follows that BT does not satisfy σ" in $(\gamma^{bt}_{ck}, \pi^{bt})$.

An advantage of σ" is that it can be satisfied without the sender having to send any message in contexts such as $(\gamma^{bt}_{ck}, \pi^{bt})$ in which the value of the initial bit is a common k nowledge.

Note that the specification σ" is not run-based. To verify that the condition $<> K_S K_R(bit)$ holds, we need to consider the whole system, not just a run in isolation.

Knowledge-based specifications such as σ" are quite important in pract-ice. If a system satisfies σ", we know that in a sense no unnecessary mes-sages are sent.

This is an information we do not have if we know only that the system satisfies σ'.