Fundamenta Informaticae XX (2003) 1–20
IOS Press

Knowledge-Based Asynchronous Programming

Hendrik Wietze de Haan

Wim H. Hesselink

Gerard R. Renardel de Lavalette

Abstract. A knowledge-based program is a high-level description of the behaviour of agents in terms of knowledge that an agent must have before (s)he may perform an action. The definition of the semantics of knowledge-based programs is problematic, since it involves a vicious circle; the knowledge of an agent is defined in terms of the possible behaviours of the program, while the possible behaviours are determined by the actions which depend on knowledge. We consider a large class of knowledge-based programs, and define their semantics via an approximation approach generalizing the well-known fixpoint construction.

1. Introduction

A knowledge-based program (KBP) is a program with explicit tests for knowledge. A multi-agent setting is assumed: the KBP consists of subprograms associated with each of the agents involved, describing under what conditions the agent should take certain actions. The conditions are epistemic formulae, expressing, e.g., that the agent knows certain facts or regards certain facts as possible. Although KBPs are, in general, not directly implementable, this high-level approach may yield better insight into complex situations like multi-agent protocols.

However, the semantics of KBPs is problematic, due to a vicious circle. On the one hand, the tests in a KBP may contain epistemic operators, and the semantics of these epistemic operators is defined in terms of the traces (possible behaviours) of the KBP. On the other hand, the collection of traces of the KBP depends on the semantics of the tests. How to deal with this cyclic dependency? In this paper, we present and work out a method to do this for a large class of KBPs.

This work is inspired by the seminal book [6] by Fagin and others. They define the semantics of a KBP via an implementation by a *standard program* (i.e. without epistemic formulae in the tests), such

Address for correspondence: Department of Mathematics and Computing Science, University of Groningen,
P.O. Box 800, 9700 AV Groningen, The Netherlands
Email: {hwh, wim, grl}@cs.rug.nl

that the runs (i.e. traces) of the standard program correspond to the runs of the KBP; the runs of the standard program are used to evaluate the tests for knowledge, and the semantics of the KBP is defined via a fixpoint construction. However, not all KBPs admit such implementations: restrictions on KBPs are given that guarantee a well-defined implementation, but these only apply to a synchronous systems where all agents execute actions in lock-step. These restrictions hamper the application of their approach, since it limits formal reasoning to a subset, excluding e.g. asynchronous KBPs.

In the approach presented in this paper, we assign semantics to *all* KBPs under consideration. We are particularly interested in the semantics of KBPs in an asynchronous setting, where the agents do not synchronize with a common clock. To this end, an abstract setup of the theory of asynchronous KBPs is given together with the definition of the semantics to assign to such programs. Our choice of semantics is justified by means of a number of examples. These examples show how the semantics correspond with our intuition.

1.1. Overview

In section 2 we discuss related work and sketch our contribution. The formal framework is developed in section 3. We define a language of KBPs, assign an interpretation to this language and propose how to assign semantics to all knowledge-based programs. Our choice of semantics is justified in 4 via a number of examples. Some conclusions and directions for future research are discussed in section 5.

2. Several approaches to knowledge-based programming

The first papers on knowledge in distributed systems appear in the '80s: Chandy and Misra [3] describe how processes gain and lose knowledge, Katz and Taubenfeld [13] define various notions of knowledge, depending on the state of information a process has. However, the programs in these papers are not KBPs in our sense, since they do not contain explicit tests for knowledge.

In an attempt to reason formally about KBPs and to gain more insight in KB-programming, Sanders defines in [16] knowledge of a process using predicate transformers, and points out that safety and liveness properties of KBPs need not be preserved when the initial conditions are strengthened. Yet, Halpern and Zuck successfully use in [9] the knowledge-based approach to derive and prove the correctness of a family of protocols for the sequence transmission problem. Their motivation for using a knowledge-based approach is that correctness proofs should also offer an understanding to the reader why a protocol is correct. Stulp and Verbrugge in [17] show that real-life protocols can be analysed using a knowledge-based approach by presenting a KBP for the TCP-protocol.

Moses and Kislev introduce in [15] the notion of knowledge-oriented programs, i.e. KBPs with high-level actions that change the epistemic state of the agents. An example: the action $\mathit{notify}(j, \varphi)$ which ensures that agent j will eventually know φ. Formal semantics are not given, however. This kind of programs is also investigated more formally from the perspective of dynamic epistemic logic, e.g. by Baltag [2] and Van Ditmarsch [4]. The implementation of knowledge-oriented programs is not considered.

The book "Reasoning about Knowledge" [6], by Fagin et al., contains an overview of the work done on KB-programming by the authors and others. A general framework for KBPs is given, and the semantics of a KBP is defined via a standard (i.e. knowledge-free) program that implements the KBP. This

knowledge-free program can be decomposed into a protocol for each agent that maps local states to actions. In [6, 7], sufficient conditions for the existence of a well-defined implementation of a KBP are given; there conditions only apply to synchronous systems, however. In [7] the complexity of determining whether a KBP has a well-defined implementation in a given finite state context is characterized. In [18], the complexity of checking whether a protocol implements a KBP in a given finite state context is studied.

Van der Meyden in [14] describes an axiomatization of a logic of knowledge and time for a class of synchronous and asynchronous systems and studies the model checking problem for this setting. However, the gap between writing down a KBP and determining its model remains. Engelhardt, van der Meyden and Moses develop in [5] a refinement calculus for KBPs. Their framework assumes that agents and environment act synchronously.

2.1. Our approach

Our approach is related to the general approach given in [6], but there are some differences. The most importance difference is that all KBPs are assigned semantics in our formalism, whereas [6] only assigns semantics when a KBP has an implementation. In our view, the principal aim of any semantic framework for programming languages is that it should assign meanings to all programs that can be expressed in the language under consideration. Sufficient conditions on a KBP to ensure existence of a well-defined implementation were given in [6], but only for synchronous systems.

In this paper we abandon the synchrony of the agents by a common clock and adopt an interleaving semantics. Partly because this choice, a generally applicable fixpoint definition for the semantics is excluded. The alternative we propose is based on the transfinite sequence of approximations. Cardinality reasons ensure that this sequence turns into itself. We then proceed along the sequence and stop at the approximation where the next step would introduce contradictory knowledge. In a certain sense, we thus define the semantics of the program as the last acceptable approximation.

3. Modelling knowledge-based programs

In this section, we develop our framework in which we can define the knowledge that agents have during execution.

3.1. Knowledge-based programs

Knowledge-based programs are programs with explicit tests for knowledge. We define a language *KBP* of programs in a way similar to *PDL* (propositional dynamic logic, see e.g. [10]) with the exception that we disallow the dynamic operator, i.e. $PDL + K_a - [\cdot]$. As a consequence, no mutual recursion is needed to define the language of formulas and the language of programs (An alternative is to include the dynamic operator, but to forbid its occurrence in tests.)

The language $\mathcal{L}$ of epistemic formulas (knowledge expressions) for a set of agents A, given a set Φ_0 of propositional variables, is defined in BNF-notation as

$$\varphi ::= \bot \mid p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid K_a \varphi$$

where $p \in \Phi_0$, $a \in A$. The connectives $\vee, \rightarrow, \leftrightarrow$, and the constant $\top$ are defined as usual in terms of $\bot$, $\neg$ and $\wedge$. K_a is the knowledge operator for agent a. Possibility, M_a, is the dual of K_a, that is, $M_a = \neg K_a \neg$.

Next the language Π of epistemic programs is defined, given a set Π_0 of primitive action symbols, as

$$\alpha ::= f \mid \varphi? \mid (\alpha; \alpha) \mid (\alpha \cup \alpha) \mid \alpha^*$$

where $f \in \Pi_0$, $\varphi \in \mathcal{L}$. A program is either a primitive action f, a test φ, the composition of two programs, the choice between two programs or the repetition of a program. Repetition binds first, then test, followed by composition and choice (in that order). **skip** is a fixed constant in Π_0.

In the sequel we restrict ourselves to programs *Pg* of the form

$$Pg_J \;=\; ((\cup_{j \in J} \;\varphi_j? \,;\; f_j) \;\cup\; \mathbf{skip})^*$$

where $\varphi_j \in \mathcal{L}$, $f_j \in \Pi_0$ and J is some finite index set. This is index set is implicit, so the subscript J will be omitted. We call programs *Pg* of the above form knowledge-based programs (KBPs).

A KBP thus is a repeated nondeterministic choice over a family of so-called guarded commands. A guarded command is a primitive action preceded by a test, the guard of that action. We allow guarded commands of the form $\top?\,;\, f_j$ to be replaced by its body f_j. The alternative **skip** introduces asynchrony. Such a step is called a stuttering step since the state does not change.

A multi-agent system consists of some agents and an environment. Each agent executes a KBP and the environment executes a knowledge-free program, i.e. a KBP where the guards do not contain epistemic operators. The KBPs of the agents and the program of the environment can be combined into a larger knowledge-based program of the same form as *Pg*, because of the following rule

$$Pg_J \;||\; Pg_{J'} \;=\; Pg_{J \cup J'}.$$

More precisely, we abstract away from which agent it is that acts and consider programs of the form *Pg* to represent the entire multi-agent system.

Before we can assign semantics to programs we must define how to evaluate a test for knowledge.

3.2. A Kripke model of traces

Knowledge is usually defined w.r.t. a Kripke model $\langle W, R, V \rangle$, where W is a nonempty set of worlds, $\pi : \Phi_0 \rightarrow \mathcal{P}(X)$ is a valuation and $R = (R_a)_{a \in A}$ is an equivalence relation on states for each agent $a \in A$. Agent a knows that φ holds in world $w \in W$ if φ is satisfied in all worlds $v \in W$ for which $(w, v) \in R_a$ holds.

For the semantics of KBP, we shall use an extended Kripke model $M_B = \langle X^+, B, \approx, \pi \rangle$. Here X^+ is the collection of finite nonempty sequences (called *traces*) of states $x \in X$, $B \subseteq X^+$ is the collection of possible traces, $\approx = (\overset{a}{\approx})_{a \in A}$ a collection of equivalence relations, and $\pi : \Phi_0 \rightarrow \mathcal{P}(X)$ is a valuation.

The collection of states X is part of a (simpler) Kripke model $\langle X, \sim, \pi \rangle$ which we present first. The relation $\overset{a}{\sim}$ captures the knowledge that an agent has in a given state. The usual approach here is to consider the state x as a global state that consists of several local states, one for each agent; the local state of an agent is determined by all variables that are accessible for the agent. The relation $\overset{a}{\sim}$ is then defined as: agent a cannot distinguish between two states x and y (notation: $x \overset{a}{\sim} y$) iff a's local state is

the same in both x and y. When defined like this, $\overset{a}{\sim}$ is an equivalence relation. We shall abstract from the underlying notion of local state, and only require that the $\overset{a}{\sim}$ are equivalence relations.

Now we turn to traces $xs \in X^+$. The idea is: agent a knows φ in xs iff φ holds in all traces ys that (i) a cannot distinguish from xs, and (ii) considers possible. To formalize (ii), we refer to the collection $B \subseteq X^+$ of possible traces, given the program under consideration (as we shall see, this parameter B will be the source of cyclic dependency in the definition of the semantics of KBP). The formalization of (i) is based on two principles: perfect recall and asynchronicity. Perfect recall expresses that each agent is assumed to remember her local history, i.e. to have access to all states in the trace, not only to its last element. Asynchronicity implies that agents cannot distinguish between traces that differ only because of possible repetitions of their states. This is made more precise as follows.

When $xs \in X^+$, we write $\ell(xs)$ for the length of the trace, and xs_i for the ith element of the trace (where $0 \leq i < \ell(xs)$). The function *last* returns the last element of a trace, i.e. $last(xs) = xs_{n-1}$ where $\ell(xs) = n$. Concatenation of traces is denoted by $*$.

A list ys is a *stuttering* of xs, notation $xs \preceq ys$, iff ys is obtained from xs by consecutive repetition of certain elements of xs. For example, $aabbccc$ is a stuttering of $abbc$, but $ababc$ is not. Our notation for stuttering comes from [11].

Agent a considers trace xs indistinguishable from trace ys when stutterings can be added to both traces to obtain traces xt and yt of equal length such that agent a cannot distinguish individual states along xt and yt. We formally define the relation $\overset{a}{\approx}$ on traces for agent a by

$$xs \overset{a}{\approx} ys \quad \equiv \quad \exists\, xt, yt \in X^+.(xs \preceq xt \,\wedge\, ys \preceq yt \,\wedge\, xt \overset{a}{\sim} yt)$$

where $\overset{a}{\sim}$ has been lifted to lists over X by means of

$$xt \overset{a}{\sim} yt \quad \equiv \quad \ell(xt) = \ell(yt) \,\wedge\, \forall\, i.(0 \leq i < \ell(xt) \Rightarrow xt_i \overset{a}{\sim} yt_i) \ .$$

This ends the definition of our model $M_B = \langle X^+, B, \approx, \pi \rangle$. Before we go on, we show that $\overset{a}{\approx}$ is an equivalence relation. Indeed, it is easy to see that $\overset{a}{\approx}$ is reflexive and symmetrical, since $\overset{a}{\sim}$ is reflexive and symmetrical. It is a bit harder to verify that $\overset{a}{\approx}$ is also transitive. This follows from transitivity of $\overset{a}{\sim}$, confluence of the stuttering relation $\preceq$ and the observation that

$$xs \overset{a}{\sim} ys \,\wedge\, ys \preceq yt \quad \Rightarrow \quad \exists\, xt \in X^+.(xs \preceq xt \,\wedge\, xt \overset{a}{\sim} yt),$$

where xs, xt, ys and $yt \in X^+$.

3.3. Evaluating epistemic formulae

Now we define the interpretation of formulae and programs of KBP in the model M_B. B, the collection of traces 'under consideration', is used to resolve the circularity of mutual dependency between the definition of the semantics of the knowledge operator and the definition of the collection of possible traces, given a program and a collection of initial states. For any subset B of X^+, the interpretation

$[\![\varphi]\!]_B$ of an epistemic formula φ is defined as a set of traces, in the following way

$$\begin{array}{lcl}
[\![\bot]\!]_B & = & \emptyset, \\
[\![p]\!]_B & = & \{xs \in X^+ \mid \mathit{last}(xs) \in \pi(p)\}, \\
[\![\varphi_1 \wedge \varphi_2]\!]_B & = & [\![\varphi_1]\!]_B \cap [\![\varphi_2]\!]_B, \\
[\![\neg\varphi]\!]_B & = & X^+ \setminus [\![\varphi]\!]_B, \\
[\![K_a\,\varphi]\!]_B & = & \{xs \in X^+ \mid \forall\ ys \in B.(xs \overset{a}{\approx} ys \ \Rightarrow\ ys \in [\![\varphi]\!]_B)\}.
\end{array}$$

A trace xs satisfies proposition p iff p holds in the last state of xs under valuation π. An agent a knows that φ holds in trace xs iff φ holds in all traces ys under consideration that a cannot distinguish from xs.

The above definitions do not restrict the interpretation of a formula to be a subset of the set of traces under consideration. That is, $[\![\varphi]\!]_B$ need not be a subset of B. This is a deliberate choice to be justified later by means of the example programs in section 4.

3.4. Interpretation of epistemic programs

Since epistemic programs may contain epistemic tests, the interpretation of an epistemic action is also defined w.r.t. a set B of traces that are under consideration. The interpretation of an epistemic program α, given a set B of traces, is a binary relation on traces, that is $[\![\alpha]\!]_B \subseteq X^+ \times X^+$. Similar to the interpretation of epistemic formulas, the interpretation of an epistemic program is not restricted to be a relation on the set of traces under consideration.

We assume that the valuation π is extended to basic action symbols, such that $\pi(f)$ is a binary relation on X for an action symbol $f \in \Pi_0$. In particular, we assume that $\pi(\mathbf{skip}) = 1_X = \{(x, x) \mid x \in X\}$.

The interpretation of epistemic programs with respect to B is inductively defined by

$$\begin{array}{lcl}
[\![f]\!]_B & = & \{(xs, xs{*}z) \mid (\mathit{last}(xs), z) \in \pi(f)\} \\
[\![\varphi?]\!]_B & = & \{(xs, xs) \mid xs \in [\![\varphi]\!]_B\}, \\
[\![\alpha_1\ ;\ \alpha_2]\!]_B & = & [\![\alpha_1]\!]_B \circ [\![\alpha_2]\!]_B, \\
[\![\alpha_1 \cup \alpha_2]\!]_B & = & [\![\alpha_1]\!]_B \cup [\![\alpha_2]\!]_B, \\
[\![\alpha^*]\!]_B & = & \bigcup_{n\geq 0} ([\![\alpha]\!]_B)^n
\end{array}$$

The interpretation of primitive action symbols is lifted to traces. Two traces xs and ys are related via action f iff ys is equal to xs appended with an outcome of f applied to the last element in xs. The interpretation of an epistemic program can be seen as a (nondeterministic) recipe for transforming traces. Now **skip** has the effect that the last (i.e. current) state is repeated once.

3.5. Interpretation of knowledge-based programs

Recall that a KBP is of the form

$$Pg \ =\ ((\cup_{j\in J}\quad \varphi_j?\ ;\ f_j)\ \cup\ \mathbf{skip})^*$$

So the interpretation $[\![Pg]\!]_B$ can be rewritten as

$$\{(xs*x_0,\ xs*x_0*x_1*\cdots*x_n) \mid xs \in X^* \ \wedge\ n \geq 0 \wedge \\ \forall\, i < n.(x_i = x_{i+1} \ \vee\ \exists j.(xs*x_0*x_1*\cdots*x_i \in [\![\varphi_j]\!]_B \ \wedge\ (x_i, x_{i+1}) \in \pi(f_j)))\}$$

That is, $[\![Pg]\!]_B$ consists of pairs (ys, zs) where zs is an extension of trace ys, such that the new subsequent states are either the result of stuttering, or of some basic action f_j in Pg whose guard φ_j holds in the trace upto that state.

The set of traces generated by Pg, given a set B of traces under consideration, is defined as the set of all traces that are generated by the program Pg when starting in an initial state $y \in Y$; here $Y \subseteq X$ is a nonempty set of initial states, which is identified with a subset of X^+ in a straightforward way. We define

$$Y[\![Pg]\!]_B = \{xs \mid \exists y.(y \in Y \ \wedge\ (y, xs) \in [\![Pg]\!]_B)\}$$

We want to define the semantics of a program Pg to be some set of traces B such that Pg generates the set B when B is the set of traces under consideration, i.e. B should satisfy

$$Y[\![Pg]\!]_B = B$$

This is a fixpoint characterization of the semantics of program Pg. The set of all traces of program Pg could be a fixpoint of the function

$$F = \lambda B.Y[\![Pg]\!]_B$$

where F implicitly depends on X, Y, Pg, π.

If function F is monotonic in the sense that $F(B) \subseteq F(B')$ whenever $B \subseteq B'$, the theorem of Knaster-Tarski implies that F has a unique least and a unique greatest fixpoint. In that case, we could prefer to define the semantics of Pg as the greatest fixpoint, since that would be the most liberal interpretation of Pg. In general, however, function F is not monotonic and may have no fixpoints at all, or it may have distinct fixpoints without having a least or greatest one.

We will study a sequence of approximations to the set of traces that a program generates. These approximation sets (B_λ) are defined by transfinite induction in the following way. The first approximation B_0 consists of all nonempty finite sequences of states. For any ordinal λ, the approximation $B_{\lambda+1}$ consists of traces generated by the program when B_λ is used to evaluate epistemic formulas. When λ is a limit ordinal, the approximation B_λ is the intersection of unions of approximations that are sufficiently close to the limit.

The latter choice can be motivated as follows. If the approximation sequence forms a descending chain by the subset-relation, then for the limit an intersection is needed. On the other hand, if the sequence forms an ascending chain, then for the limit a union is needed. However, approximations may also grow and shrink. Therefore, a union of intersections or an intersection of unions are considered. Both result in an intersection for a descending sequence, and in a union for an ascending sequence. We choose the intersection of unions, as it is more liberal than a union of intersections, i.e. it includes more traces. More traces implies less knowledge. We want the agents to knows facts only when there are good reasons for them.

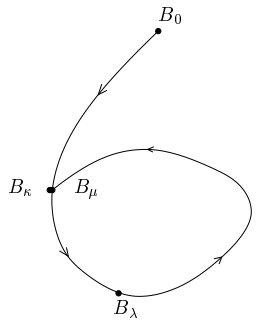

Figure 1. The approximation sequence. κ is the least ordinal such that $\exists \mu > \kappa.(B_\kappa = B_\mu)$

Formally we define

$$\begin{array}{lcll} B_0 & = & X^+, & \\ B_{\lambda+1} & = & F(B_\lambda), & \text{for any ordinal } \lambda, \\ B_\lambda & = & \cap_{\mu<\lambda} \cup_{\mu\leq\nu<\lambda} B_\nu, & \text{for any limit ordinal } \lambda. \end{array}$$

A cardinality argument implies that the transfinite sequence of approximations turns into itself: there exist ordinals $\kappa < \mu$ with $B_\kappa = B_\mu$. By well-foundedness, there is a least such κ. We now choose κ minimal with the property that $B_\kappa = B_\mu$ for some $\mu > \kappa$. If $F(B_\lambda) \subseteq B_\lambda$ for all $\lambda \geq \kappa$, then $F(B_\kappa) = B_\kappa$. That is, B_κ is a fixpoint and we choose B_κ as the semantics for the program. Otherwise we take the smallest $\lambda \geq \kappa$ such that $F(B_\lambda) \not\subseteq B_\lambda$ as the semantics for the program. This latter choice is justified by the argument that it allows as much knowledge as possible without introducing contradictory knowledge. Formally we define,

$$\text{sem}(Pg) = B_\lambda$$

where

$$\begin{array}{l} \lambda = \inf\{\lambda \,|\, \kappa \leq \lambda \wedge (B_\lambda = B_{\lambda+1} \vee\ B_{\lambda+1} \not\subseteq B_\lambda)\} \\ \kappa = \inf\{\kappa \,|\, \exists\, \mu.(\kappa < \mu\ \wedge\ B_\kappa = B_\mu)\} \end{array}$$

Since the approximation sequence turns into itself, it has the shape of a figure-six, see figure 1. The figure is an idealization; the vertical dimension is associated with inclusion. All approximations between B_κ and B_λ are subsets of previous approximations. B_λ is the last one for which this holds.

When determining the traces in an approximation it sometimes useful to restrict attention to traces generated by the knowledge-free, flat program obtained from *Pg* by omitting all guards:

$$Pg_\flat \ = \ ((\cup_{j\in J}\ f_j) \ \cup\ \mathbf{skip})^*$$

The set of all traces generated by program $Pg_\flat$ is denoted by $B_\flat$. Its definition is straightforward, since no test needs to be evaluated and so no set of traces under consideration is needed.

4. Examples

In this section we investigate the consequences of our formalism by means of a number of examples. The first three examples are quite similar and serve to show that function F is not monotonic, may have unordered fixpoints, or no fixpoints at all. The fourth example shows that the approximation sequence may turn infinite. The fifth example demonstrates how message passing can be modelled. The sixth example is an example of a program with nested knowledge operators, viz. a test for knowledge of the knowledge that another agent has. This example also illustrates that stability of knowledge predicates does not guarantee fixpoint semantics. The final example shows what semantics the unexpected hanging paradox is given in our formalism. For every example the approximation sequence is also given as a figure.

4.1. Failure of monotonicity

Suppose there are two agents a and b and two boolean variables p, q, initially true. We represent truth values as the integers 0 and 1, so the state space is $X = \{0,1\} \times \{0,1\}$ with initial state $(1,1)$. The first component of the state is the value of variable p, the second one is the value of q. This induces a valuation on states.

Variable p is private to agent a, and q is private to agent b. That is, agent a can only see and write p, agent b can only see and write q. We take the indistinguishability relations on states to be defined by

$$(p,q) \overset{a}{\sim} (p',q') \Leftrightarrow p = p', \quad (p,q) \overset{b}{\sim} (p',q') \Leftrightarrow q = q'.$$

The agents execute KBPs. Agent a can falsify p by setting p to zero when she knows that q holds. Agent b can falsify q when she knows that p holds. We take assignments to variables as primitive action symbols, with corresponding interpretation. The resulting KBP is thus

$$Pg = (\ K_a q\,?\ ;\ p := 0\ \cup\ K_b p\,?\ ;\ q := 0\ \cup\ \textbf{skip}\ )^*.$$

By definition $B_0 = X^+$. For the next approximation, the traces in B_0 are used to evaluate epistemic tests. When X^+ is considered, agents have only knowledge of the values of their own private variables. That is, if $B = X^+$ then $[\![K_a q]\!]_B = \emptyset$ and $[\![K_b p]\!]_B = \emptyset$. Both agents are unable to act. Therefore, $B_1 = F(B_0) = (1,1)^+$. So, traces in B_1 are repetitions of the initial state.

All traces in B_1 satisfy both p and q. Therefore, if $B = B_1$ then $[\![K_a q]\!]_B = [\![K_b p]\!]_B = X^+$. This implies that the agents can independently set their private variable to zero. Therefore, $B_2 = F(B_1)$ consists of traces generated by the knowledge-free program

$$(\ p := 0\ \cup\ q := 0\ \cup\ \textbf{skip}\ )^*.$$

Alternatively, we can say that B_2 consists of traces that match the regular expression

$$(1,1)^+\ \mid\ (1,1)^+(0,1)^+(0,0)^*\ \mid\ (1,1)^+(1,0)^+(0,0)^*.$$

When the agents can independently set their private variable to zero, they have no knowledge on the value of the other agent's variable. That is, if $B = B_2$ then $[\![K_a q]\!]_B = [\![K_b p]\!]_B = \emptyset$. Therefore $B_3 = F(B_2) = (1,1)^+$, and so $B_3 = B_1$.

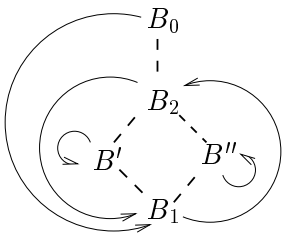

Figure 2. Example 4.1: approximation sequence and the fixpoints B' and B''

The approximation sequence repeats itself. It seems reasonable to take B_1 as the correct meaning of the program. This is in accordance with the definition in section 3.5.

The above approximations also show that F is not monotonic. For example, B_1 is a subset of B_2, however, $F(B_1)$ is not a subset of $F(B_2)$. Furthermore, in this example F has the following two unordered fixpoints:

$$\begin{aligned} B' &= (1,1)^+(0,1)^*, \\ B'' &= (1,1)^+(1,0)^*. \end{aligned}$$

To show that B' is a fixpoint of F, observe that all traces in B' satisfy q, so $[\![K_a q]\!]_{B'} = X^+$. Therefore, agent a can set p to zero at any time. Traces matching $(1,1)^+$ or $(1,1)^+(0,1)^+$ are both indistinguishable for agent b from traces of the form $(1,1)^+(0,1)^+$, and therefore do not satisfy $K_b p$ under B'. Agent b can not set q to zero in the initial state nor after agent a has reset p, so $F(B') = B'$. Analogously, $F(B'') = B''$.

The approximation sequence and the fixpoints B' and B'' are sketched in figure 2. The dashed line indicates a subset relation, the lower set is a subset of the upper set. Lines with an arrow give the direction of approximation, i.e. application of F.

4.2. A case with fixpoint semantics

This example has the same setting as the previous example. There are two agents a and b, with private boolean variables p respectively q. The state space is as before, but now the initial state is $(0,0)$. An agent may set her private variable to 1 if she considers it possible that the other agent's private variable has already been set. This corresponds to the following program

$$Pg = (\ M_a q\,?\ ;\ p := 1\ \cup\ M_b p\,?\ ;\ q := 1\ \cup\ \textbf{skip}\ )^*.$$

Again we determine a sequence of approximations. $B_0 = X^+$. When X^+ is considered, agents have only knowledge of their private variables. If agent a does not know whether q holds then she considers both q and $\neg q$ possible. So, if $B = X^+$ then $[\![M_a q]\!]_B = [\![M_b p]\!]_B = X^+$. The agents can independently of each other set their private variables, so $B_1 = F(B_0)$ consists of the traces generated by

$$(\ p := 1\ \cup\ q := 1\ \cup\ \textbf{skip}\ )^*.$$

In B_1 agents always consider it possible that the other has set her variable. Therefore, if $B = B_1$ then $[\![M_a q]\!]_B = [\![M_b p]\!]_B = X^+$, so $B_2 = F(B_1) = B_1$, and we have reached a fixpoint. Note that $(0,0)^+$ is also a fixpoint of F. The approximation is sketched in figure 3.

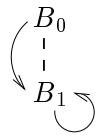

Figure 3. Example 4.2: approximation sequence that ends in a fixpoint

The semantics of section 3.5 chooses B_1 as the semantics of the program. Anthropomorphically speaking, we see that either agent reckons with the possibility that the other agent has set her variable, even though she can argue that the other agent should not be able to do so as the first one. This is an unexpected side effect of the asynchrony.

4.3. Absence of fixpoints

Again there are two agents a and b with private boolean variables p and q. Initially p is false and q is true. Now, agent a can set p when she knows that q holds, and agent b can reset q when she considers it possible that p has been set. The program is given as

$$Pg = (\ K_a q\,?\ ;\ p := 1\ \cup\ M_b p\,?\ ;\ q := 0\ \cup\ \textbf{skip}\ )^*.$$

As seen in the previous two examples, if $B = X^+$ then $[\![K_a q]\!]_B = \emptyset$ and $[\![M_b p]\!]_B = X^+$. Only agent b can act, so $B_1 = F(B_0)$ consists of traces of program

$$(\ q := 0\ \cup\ \textbf{skip}\ )^*.$$

That is, B_1 contains traces that match $(0,1)^+(0,0)^*$.

For $B = B_1$, the interpretation $[\![K_a q]\!]_B$ consists of traces that cannot be generated by the program. For example, a trace xs matching $(1,0)^+$ satisfies $K_a q$ under B_1, since no trace in B_1 is indistinguishable for a from xs. Therefore, we determine the set $B_\flat$ of all traces that can possibly be generated by program Pg. The program only allows p to change from 0 to 1, and q from 1 to 0. So $B_\flat$ is the set of all traces of the program in which the guards are removed, that is the program

$$(\ p := 1\ \cup\ q := 0\ \cup\ \textbf{skip}\ )^*.$$

No trace in $B_\flat$ satisfies $K_a q$ under B_1. In other words, if $B = B_1$ then $B_\flat \cap [\![K_a q]\!]_B = \emptyset$. No trace in B_1 satisfies p. Therefore, if $B = B_1$ then $[\![M_b p]\!]_B = \emptyset$. Both agents can not act, so $B_2 = F(B_1) = (0,1)^+$.

All traces in B_2 satisfy $\neg p$ and q, so for $B = B_2$ we have $[\![K_a q]\!]_B = X^+$ and $[\![M_b p]\!]_B = \emptyset$. Only agent a can act. Therefore, $B_3 = F(B_2)$ consists of traces generated by program

$$(\ p := 1\ \cup\ \textbf{skip}\ )^*.$$

Since q is not reset, all traces in B_3 satisfy q, so if $B = B_3$ then $[\![K_a q]\!]_B = X^+$. Observe that there are traces that satisfy $M_b p$ under B_3, but that are not in $B_\flat$. However, if $B = B_3$ then $B_\flat \cap [\![M_b p]\!]_B = B_\flat$. Therefore, agent b can set q to zero under B_3, so $B_4 = F(B_3) = B_\flat$.

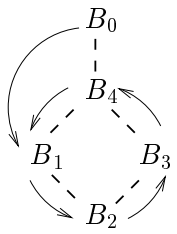

Figure 4. Example 4.3: a concrete six-shaped diagram as given in figure 1

If $B = B_4$ then $[\![K_a q]\!]_B = \emptyset$ and $B_\flat \cap [\![M_b p]\!]_B = B_\flat$. Only agent b can act, therefore $B_5 = F(B_4)$ is equal to B_1. At this point the approximation sequence repeats. The sequence is drawn in figure 4.

In this example the semantics of section 3.5 chooses B_2 as the semantics of the program, since $B_1 \supseteq B_2 \not\supseteq B_3$. Therefore, the semantics is that the systems stays in its initial state.

4.4. An infinite approximation sequence

In this example we show that the approximation sequence may become infinite. There are again two agents a and b. Agent a has private integer variables p and m, and agents b has private integer variables q and n. Initially $p = q = 0$ and $m = n = 1$. Agent a may always increment m. If agent a knows that q is zero she can set p to 1. If agent a considers it possible that q between 1 and m, she can advance p to $m + 1$. Agent b can do similar actions. If we allow equalities and inequalities to appear in the epistemic formulas, then KBP is

$$
\begin{array}{ll}
(& m := m + 1 \\
\cup & n := n + 1 \\
\cup & K_a(q = 0)\,?\ ;\ p := 1 \\
\cup & K_b(p = 0)\,?\ ;\ q := 1 \\
\cup & M_a(1 \le q \le m)\,?\ ;\ p := m + 1 \\
\cup & M_b(1 \le p \le n)\,?\ ;\ q := n + 1 \\
\cup & \textbf{skip}\ )^*.
\end{array}
$$

Again $B_0 = X^+$. When X^+ is considered, agent a can increment p, but always to a value larger than m. Since agent a cannot inspect q she can always consider a trace in X^+ possible such that $1 \le q \le m$. Agent a can not set p to 1, because for $B = X^+$ we have $[\![K_a(q = 0)]\!]_B = \emptyset$. Agent a can always increment m. Similar arguments for agent b show that b cannot set q to 1. Therefore, $B_1 = F(B_0)$ consists of the traces of program

$$(m := m + 1 \ \cup\ n := n + 1 \ \cup\ p := m + 1 \ \cup\ q := n + 1 \ \cup\ \textbf{skip}\ )^*.$$

All traces in B_1 satisfy $p \neq 1 \neq q$. So in B_1 agent a only considers $1 \le q \le m$ possible if $m \ge 2$. Again a never knows that $q = 0$. Similar arguments hold for b. Therefore, $B_2 = F(B_1)$ consists of the traces of program

$$(m := m + 1 \ \cup\ n := n + 1 \ \cup\ (m \ge 2)\,?\ ;\ p := m + 1 \ \cup\ (n \ge 2)\,?\ ;\ q := n + 1 \ \cup\ \textbf{skip}\ )^*.$$

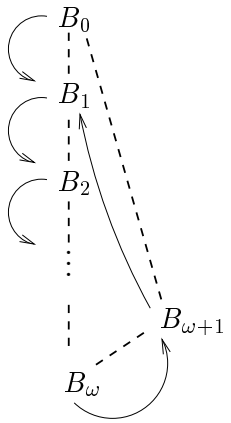

Figure 5. Example 4.4: an infinite sequence of approximations

All traces in B_2 satisfy $p \notin \{1,2\}$ and $q \notin \{1,2\}$. In B_2 agent a only considers $1 \leq q \leq m$ possible if $m \geq 3$, but she never knows that $q = 0$. Continuing the same reasoning as above, we observe that for all integers $i \geq 1$, all traces in $B_{i+1} = F(B_i)$ satisfy $p \notin \{1,..,i\}$ and $q \notin \{1,..,i\}$. Furthermore, it is not hard to observe that $B_{i+1} \subseteq B_i$. Therefore, the transfinite approximation is $B_\omega = \cap_{i<\omega} \cup_{i\leq j<\omega} B_j = \cap_{i<\omega} B_i$. In B_ω values of p and q are never increased, that is B_ω consists of traces of program

$$(\ m := m+1 \ \cup \ n := n+1 \ \cup \ \textbf{skip}\ )^*.$$

All traces in B_ω satisfy $p = 0 = q$, so in B_ω agent a knows that $q = 0$ and can thus set p to 1. However, p can not be incremented further as there are no traces in B_ω for which $1 \leq q \leq m$ holds for any m. Similarly, q can only be set to 1. So $B_{\omega+1}$ consists of traces of program

$$(\ m := m+1 \ \cup \ n := n+1 \ \cup \ p := 1 \ \cup \ q := 1 \ \cup \ \textbf{skip}\ )^*.$$

In $B_{\omega+1}$, agent a never knows that $q = 0$. However, she may consider it possible that b has incremented q such that $1 \leq q \leq m$, so she can set p to $m+1$. Therefore, $B_{\omega+2} = F(B_{\omega+1})$ is equal to B_1, so the approximation sequence repeats itself. The sequence is drawn in figure 5.

In this example the semantics of section 3.5 chooses B_ω as the semantics of the program. In this program agents can not make any assumption on the values of private variables of the other agent. The values of p and q should remain unchanged, while m and n may increase.

4.5. Message passing

Message passing can be modelled as a variable that can be written by one agent and read by another. In this example, the agents a and b communicate via the integer variable q. Agent a has a private integer variable p and agent b has a private integer variable r. All variables are initially 0. The state space consists of triples (p,q,r), and we have

$$(p,q,r) \overset{a}{\sim} (p',q',r') \Leftrightarrow p = p' \ \wedge \ q = q',$$
$$(p,q,r) \overset{b}{\sim} (p',q',r') \Leftrightarrow q = q' \ \wedge \ r = r'.$$

Agent a may increment p when she considers it possible that $p \leq r$. She may increment q when she knows that $q < p$. Agent b may increment r when she knows that $r < p$. The program is

$$\begin{array}{ll} Pg = & (\quad M_a\,(p \leq r)\,?\,;\; p := p+1 \\ & \cup \qquad (q < p)\,?\,;\; q := q+1 \\ & \cup \;\; K_b\,(r < p)\,?\,;\; r := r+1 \\ & \cup \;\mathbf{skip}\;)^* \,. \end{array}$$

Because agent a can read both p and q, the test $q < p$ is equivalent to $K_a(q < p)$.

Agent b can infer $r < p$ from $r < q$, since she can read q and agent a preserves the invariant $q \leq p$. Operationally speaking, agent a uses variable q as a message to agent b to let b know that a has incremented p.

Intuitively, one could expect the following behaviours from this program. Agent a may increment p, followed by q. Then b may increment r. Since a knows this, she may increment p after the incrementation of q; she need not wait for the incrementation of r, which is hidden to her anyhow. We will investigate whether our formalism justifies this intuition.

In $B_0 = X^+$, agent a considers it possible that $p \leq r$, but b has no knowledge whether $r < p$. Therefore the traces in $B_1 = F(B_0)$ are the traces generated by the knowledge-free program $P1$

$$\begin{array}{ll} \text{P1} = & (\quad (p \leq r)\,?\,;\; p := p+1 \\ & \cup\; (q < p)\,?\,;\; q := q+1 \\ & \cup\; \mathbf{skip}\;)^* \,. \end{array}$$

For the next approximations it is useful to only consider traces that can possibly be generated by program *Pg*. Therefore, while determining the set of traces that satisfy $K_b(r < p)$ given a set B, attention can be be restricted to traces in the set $B_\flat$, which consist of traces generated by the flat, guardless program

$$(\; p := p+1 \;\cup\; q := q+1 \;\cup\; r := r+1 \;\cup\; \mathbf{skip}\;)^*.$$

For $B = B_1$, we have that $[\![M_a(p \leq r]\!]_B$ consists of traces $p \leq 0$, since r remains 0 in B_1. On the other hand, $B_\flat \cap [\![K_b(r < p)]\!]_B$ consists of traces of $B_\flat$ that end with $r < q$, since $q \leq p$ is invariant in B_1. Therefore, $B_2 = F(B_1)$ consists of traces generated by

$$\begin{array}{ll} \text{P2} = & (\quad (p \leq 0)\,?\,;\; p := p+1 \\ & \cup\; (q < p)\,?\,;\; q := q+1 \\ & \cup\; (r < q)\,?\,;\; r := r+1 \\ & \cup\; \mathbf{skip}\;)^* \,. \end{array}$$

Note that the traces in B_2 satisfy the invariant $0 \leq r \leq q \leq p \leq 1$.

For $B = B_2$, we have that $[\![M_a(p \leq r)]\!]_B$ consists of traces with $p \leq q \;\wedge\; p \leq 1$, since $r \leq 1$ holds in B_1. On the other hand, $B_\flat \cap [\![K_b(r < p)]\!]_B$ consists of the traces of $B_\flat$ that end with $r < q \vee\; 2 \leq q$. In fact, no trace *xs* with $2 \leq q$ is indistinguishable for b from any trace in B_2, since all traces in B_2 satisfy $q \leq 1$. It follows that $B_3 = F(B_2)$ consists of traces generated by

$$\begin{array}{ll} \text{P3} = & (\quad (p \leq q \;\wedge\; p \leq 1)\,?\,;\; p := p+1 \\ & \cup\; (q < p)\,?\,;\; q := q+1 \\ & \cup\; (r < q \vee\; 2 \leq q)\,?\,;\; r := r+1 \\ & \cup\; \mathbf{skip}\;)^* \,. \end{array}$$

Note that, traces in B_3 satisfy the invariants $0 \leq r$ and $0 \leq q \leq p \leq 2$ and $q < 2 \Rightarrow r \leq q$.

For $B = B_3$, we have that $[\![M_a(p \leq r)]\!]_B$ consists of traces with $p \leq q \vee\ 2 \leq q$, since in B_3, r is only bounded by q while $q < 2$. On the other hand, $B_b \cap [\![K_b(r < p)]\!]_B$ consists of traces of B_b that end with $r < q \vee\ 3 \leq q$. No trace satisfying $3 \leq q$ is indistinguishable for b from any trace in B_3, since traces in B_3 satisfy $q \leq 2$. It follows that $B_4 = F(B_3)$ consists of the traces generated by

$$\begin{array}{ll} \text{P4} = & (\ \ (p \leq q \vee\ 2 \leq q)\,?\ ;\ p := p + 1 \\ & \cup\ (q < p)\,?\ ;\ q := q + 1 \\ & \cup\ (r < q \vee\ 3 \leq q)\,?\ ;\ r := r + 1 \\ & \cup\ \textbf{skip}\ \ )^* \ . \end{array}$$

Observe that, traces in B_4 satisfy the invariants $0 \leq r$ and $0 \leq q \leq p$ and $q < 3 \Rightarrow r \leq q$.

For $B = B_4$, we have that $[\![M_a(p \leq r)]\!]_B$ consists of traces with $p \leq q \vee\ 3 \leq q$, since in B_4, r is only bounded by q while $q < 3$. On the other hand, $B_b \cap [\![K_b(r < p)]\!]_B$ consists of those traces of B_b that end with $r < q$ because of the invariant $q \leq p$ in B_4. It follows that $B_5 = F(B_4)$ consists of the traces generated by

$$\begin{array}{ll} \text{P5} = & (\ \ (p \leq q \vee\ 3 \leq q)\,?\ ;\ p := p + 1 \\ & \cup\ (q < p)\,?\ ;\ q := q + 1 \\ & \cup\ (r < q)\,?\ ;\ r := r + 1 \\ & \cup\ \textbf{skip}\ \ )^* \ . \end{array}$$

Traces in B_5 satisfy the invariant $0 \leq r \leq q \leq p$.

For $B = B_5$, we have that $[\![M_a(p \leq r)]\!]_B$ consists of the traces with $p \leq q$ since r is only bounded by q in B. On the other hand, $B_b \cap [\![K_b(r < p)]\!]_B$ consists of those traces of B_b that end with $r < q$ because all traces in B_5 satisfy the invariant $q \leq p$. It follows that $B_6 = F(B_5)$ consists of traces generated by

$$\begin{array}{ll} \text{P6} = & (\ \ (p \leq q)\,?\ ;\ p := p + 1 \\ & \cup\ (q < p)\,?\ ;\ q := q + 1 \\ & \cup\ (r < q)\,?\ ;\ r := r + 1 \\ & \cup\ \textbf{skip}\ \ )^* \ . \end{array}$$

Note that, traces in B_6 satisfy the invariant $0 \leq r \leq q \leq p \leq q + 1$.

Similarly as before, for $B = B_6$, we have that $[\![M_a(p \leq r)]\!]_B$ consists of traces with $p \leq q$ since r is only bounded by q in B. On the other hand, $B_b \cap [\![K_b(r < p)]\!]_B$ consists of those traces of B_b that end with $r < q$ because of the invariant $q \leq p$ in B_6. It follows, that $F(B_6) = B_6$, a fixpoint. Moreover, this fixpoint corresponds with the intuition sketched above, and is chosen according to section 3.5. The approximation sequence is shown in figure 6.

4.6. Stable predicates

Informally, a predicate is stable iff once the predicate holds it continues to hold. In this example, the guards in the program are stable. However, as we will show, this is no guarantee for fixpoint semantics. Furthermore, stability of epistemic formulas depends on the set of traces that are considered. At the moment it is unclear to us how to formally define stability with respect to KBPs.

In this example, there are three agents, a, b and c. The state space consists of six boolean variables pa, qa, pb, qb, r, s, with initially $pa = qa = pb = qb = 0$ and r, s indeterminate. Agent a can read pa

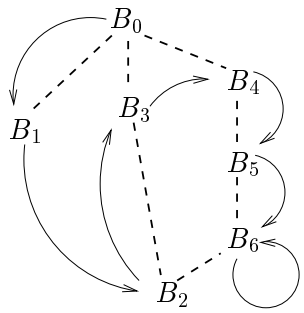

Figure 6. Example 4.5: nonmonotonic approximation that leads to a fixpoint.

and read and write qa, agent b can read pb and read and write qb. Agent c can see all variables and write pa and pb. The indistinguishability relations on states for agents a and b are as follows.

$$(pa, qa, pb, qb, r, s) \overset{a}{\sim} (pa', qa', pb', qb', r', s') \Leftrightarrow\ pa = pa' \wedge qa = qa'$$
$$(pa, qa, pb, qb, r, s) \overset{b}{\sim} (pa', qa', pb', qb', r', s') \Leftrightarrow\ pb = pb' \wedge qb = qb'$$

The relation $\overset{c}{\sim}$ is just equality on states.

Variables pa, qa, pb and qb will be used to send messages. Agent c can send a message to a by altering variable pa, and agent c can send a message to b by altering variable pb. Agent a sends c a message by altering qa. Likewise, b can send c a message via qb.

Informally, the agents act as follows. Agent a may send c a message when she knows that s holds. Agent b may send c a message when she knows that r holds. When agent c knows that either r holds or that a knows that s holds, she may send b a message. When agent c knows that either s holds or that b knows that r holds, she may send a a message. The knowledge-based program is

$$\begin{array}{ll} \text{Pg} = & (\ \ K_a s\,?\ ;\ qa := 1 \\ & \cup\ K_b r\,?\ ;\ qb := 1 \\ & \cup\ K_c(r \vee K_a s)\,?\ ;\ pb := 1 \\ & \cup\ K_c(s \vee K_b r)\,?\ ;\ pa := 1 \\ & \cup\ \textbf{skip}\ \ )^* \ . \end{array}$$

Note that the values of variables r and s do not change, so $K_a s$ and $K_b s$ are stable. Also, $K_c(r \vee K_a s)$ and $K_c(s \vee K_b r)$ are stable.

For $B = B_0$, we have that $[\![K_a s]\!]_B = \emptyset$ and $[\![K_b r]\!]_B = \emptyset$. On the other hand, since c can read all variables, $[\![K_c(r \vee K_a s)]\!]_B$ consists of all traces where r holds at the end. Likewise, $[\![K_c(s \vee K_b r)]\!]_B$ consists of all traces where s holds at the end. It follows that $B_1 = F(B_0)$ consists of the traces generated by

$$\text{P1} = \ \ (\ r\,?\ ;\ pb := 1\ \cup\ s\,?\ ;\ pa := 1\ \cup\ \textbf{skip}\ )^*.$$

Note that all traces in B_1 satisfy the invariants $pb \Rightarrow r$, and $pa \Rightarrow s$.

For B_1 we restrict attention to traces that can be generated by the program, that is we only consider traces from $B_\flat$, generated by the flat program

$$(\ pa := 1\ \cup\ qa := 1\ \cup\ pb := 1\ \cup\ qb := 1\ \cup\ \textbf{skip}\ )^*.$$

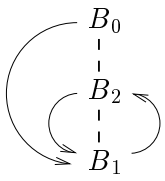

Figure 7. Example 4.6: Stable guards, not leading to a fixpoint.

For $B = B_1$, we have that $B_\flat \cap [\![K_a s]\!]_B$ consists of traces that end with pa, since the invariant $pa \Rightarrow s$ holds in B_1. Likewise, $B_\flat \cap [\![K_b r]\!]_B$ consists of traces that end with pb, since the invariant $pb \Rightarrow r$ holds in B_1. Because the guard $K_a s$ can be replaced by pa under B_1, we have that $B_\flat \cap [\![K_c(r \vee K_a s)]\!]_B$ consists of traces in $B_\flat$ that end with $r \vee pa$. Similarly, $B_\flat \cap [\![K_c(s \vee K_b r)]\!]_B$ consists of traces $B_\flat$ that end with $s \vee pb$. Therefore, $B_2 = F(B_1)$ consists of traces generated by

$$
\begin{array}{lll}
\text{P2} = & (& pa\,?\,;\ qa := 1 \\
& \cup & pb\,?\,;\ qb := 1 \\
& \cup & (r \vee pa)\,?\,;\ pb := 1 \\
& \cup & (s \vee pb)\,?\,;\ pa := 1 \\
& \cup & \textbf{skip}\ \ )^* .
\end{array}
$$

Note that the invariants $pb \Rightarrow r$ and $pa \Rightarrow s$ do not hold for traces in B_2.

For $B = B_2$, we have that $B_\flat \cap [\![K_a s]\!]_B = \emptyset$ and $B_\flat \cap [\![K_b r]\!]_B = \emptyset$, since the invariants that held in B_1 do not hold for the traces in B_2. On the other hand, $B_\flat \cap [\![K_c(r \vee K_a s)]\!]_B$ consists of the traces in $B_\flat$ that end with r. Similarly, $B_\flat \cap [\![K_c(s \vee K_b r)]\!]_B$ consists of the traces of $B_\flat$ that end with s. Therefore $B_3 = F(B_2)$ consists of the traces generated by the knowledge-free program P1, so $B_3 = B_1$. The approximation sequence is sketched in figure 7. According to section 3.5, B_1 is chosen as the semantics of *Pg*.

4.7. The Unexpected Hanging Paradox

This paradox was first drawn attention to in [12] as the case of the "Class A blackout". Presently, it is commonly known as "The Surprise Examination", or "The Unexpected Hanging".

In the unexpected hanging paradox, a convicted prisoner is to be executed within seven days, but the judge tells him that he will not know the day of his execution, even on that day itself. The prisoner might then reason that he cannot be executed on day six, because when he would still be alive on day six, he would know that he would be executed that day. By backward induction he might reason that he cannot be executed without knowing that he will be executed. Yet, on day 2 the prisoner is surprised to meet the executioner.

Let us formulate this situation as a KBP. The state space consists of three variables: an integer *day*, the day of execution *exec* $\in \{0, .., 6\}$ and a boolean *dead*. Initially *day* $= 0$, *dead* $= 0$ and the precise day *exec* of execution is unknown to the agent. The agent only knows that *exec* lies in the range $\{0, .., 6\}$.

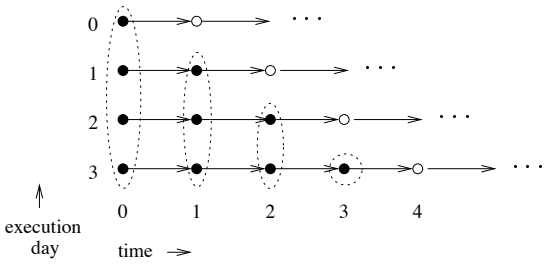

Figure 8. Traces of $B_1 = F(B_0)$ for *exec* $\in \{0,..,3\}$

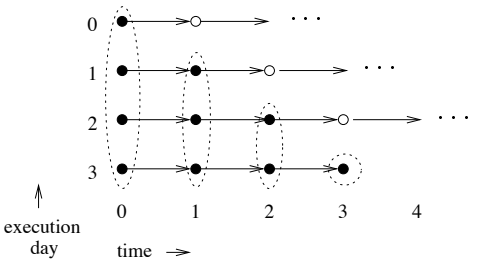

Figure 9. Traces of $B_2 = F(B_1)$ for *exec* $\in \{0,..,3\}$

The convicted agent a can observe *day* and *dead*, but not *exec*. The program is

$$\begin{array}{ll} (& (\mathit{day} = \mathit{exec} \ \wedge \ \neg K_a(\mathit{day} = \mathit{exec}) \ \wedge \ \neg \mathit{dead})\,? \ ; \ (\mathit{dead} := 1 \ , \ \mathit{day} := \mathit{day} + 1) \\ \cup & (\mathit{day} \neq \mathit{exec})\,? \ ; \ \mathit{day} := \mathit{day} + 1 \\ \cup & \textbf{skip} \ )^* \end{array}$$

where $(\mathit{dead} := 1 \ , \ \mathit{day} := \mathit{day} + 1)$ is a single action that sets the flag *dead* and advances to the next day.

In our model the program has an execution with *dead* being set when $\mathit{exec} < 6$. If initially $\mathit{exec} = 6$, the agent will know this at $\mathit{day} = 6$. So at that time, the program is stuck at day six. This seem reasonable with intuition.

We illustrate the approximations B_1 and B_2 in a smaller version of this paradox, where the execution day lies in the range $\{0,..,3\}$. The traces of the approximations B_1 and B_2 are given in figures 8 and 9. The arrows corresponding to **skip** are not shown. The dashed ellipses give the uncertainty of the agent, and are induced by $\overset{a}{\sim}$. The filled bullets are the states where the agent is alive, and the open bullets are the states where the agent has been executed. In $B_1 = F(B_0)$, the agent has no knowledge on the execution day, so he can be executed on day 3. In $B_2 = F(B_1)$, the agent cannot be executed on day 3. The approximation B_2 is a fixpoint of F, and is chosen as the semantics of the program.

Technically, our formalism does not allow deadlock, since the alternative **skip** can always be executed. Moreover, no fairness assumptions are given, e.g. there is no guarantee that time progresses. Therefore, the agent cannot use backward induction to exclude execution sequences that lead to dead-

lock, and the paradox is resolved. This is in accordance with the philosophical analysis based on dynamic epistemic logic given in [8].

5. Conclusions and directions for future research

We have presented a formal framework in which knowledge-based programs can be modelled. In our framework, the synchrony assumption has been dropped in favor of interleaving semantics. The fixpoint equation that the semantics of KBP should satisfy, does not always have a solution. Instead of restricting the semantics definition to the programs that lead to a fixpoint equation with a unique solution, we worked out a method to assign semantics to all KBPs, by calculating a sequence approximations. If this sequence ends in a fixpoint, that fixpoint is taken to be the semantics of a KBP. If not, we choose a specific approximation as the semantics. To justify this choice, a number of examples have been studied.

We fear that our current semantics does not admit very useful proof-rules. There are other choices for the semantics, such as the re-entry approximation B_κ or the first (local) minimum, i.e. the first approximation B_i for which $B_{i+1} \not\subseteq B_i$. It remains to be seen how useful these alternatives are. Moreover, the relation with the predicate transformer approach of [16] needs to be explored in more detail.

We hope to be able to define a refinement relation between KBPs in our framework. A KBP can then gradually be refined to a knowledge-free program. This would eliminate the need to explicitely calculate an approximation sequence.

It may be useful to express the function F from section 3.5 in terms of automatons. The set of traces under consideration can be defined as finite state automaton that recognizes traces from that set. An automaton transformer is then associated with our function F. This automaton transformer might be studied in a different setting.

In the current framework only finite traces are considered, and no fairness assumptions or progress properties can be expressed. We consider the possibility to extend the framework to include infinite traces and to handle deadlock and other progress properties.

References

[1] M. Abadi and L. Lamport, *The Existence of Refinement Mappings*, Theoretical Computer Science 82, 1991, pp. 253–284.

[2] A. Baltag, *A Logic for Suspicious Players: Epistemic Actions and Belief Updates in Games*, Bulletin of Economic Research, volume 54 (1), 2002, pp. 1–45.

[3] K.M. Chandy and J. Misra, *How processes learn*, Distributed Computing 1 (1), 1986, pp. 40–52.

[4] H.P. van Ditmarsch, *Descriptions of game actions*, Journal of Logic, Language and Information (JoLLI), volume 11, 2002, pp. 349–365.

[5] K. Engelhardt, R. van der Meyden and Y. Moses, *A Refinement Theory that Supports Reasoning about Knowledge and Time for Synchronous Agents*, 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2001), vol. 2250 of LNAI, Springer-Verlag, Dec 2002, pp. 125–141.

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

[7] R. Fagin, J.Y. Halpern, Y. Moses, and M.Y. Vardi, *Knowledge-based Programs*, Distributed Computing 10 (4), 1997, pp. 199–225.

[8] J. Gerbrandy, *Bisimulations on Planet Kripke*, ILLC Dissertation Series, Amsterdam, 1999.

[9] J.Y. Halpern and L.D. Zuck, *A little knowledge goes a long way: knowledge-based derivations and correctness proofs for a family of protocols*, Journal of the ACM 39 (3), 1992, pp. 449–478.

[10] D. Harel, D. Kozen and J. Tiuryn, *Dynamic Logic*, MIT Press, 2000.

[11] W.H. Hesselink, *Eternity Variables to Simulate Specifications*, In: E.A. Boiten, B. Möller (eds.): Proceedings Mathematics of Program Construction, Dagstuhl, 2002 (LNCS 2386), pp. 117–130.

[12] D.J. O'Connor, *Pragmatic Paradoxes*, Mind 57, 1948, pp. 358–359.

[13] S. Katz and G. Taubenfeld, *What Processes Know: definitions and proof methods*, Proceedings of the 5th ACM Symposium on Principles of Distributed Computing, 1986, pp. 249–262.

[14] R. van der Meyden, *Common Knowledge and Update in Finite Environments*, Information and Computation Vol 140, No. 2, Feb 1998, pp. 115–157.

[15] Y. Moses and O. Kislev, *Knowledge-oriented programming (extended abstract)*, Proceedings of the 10th ACM Symposium on Principles of Distributed Computing, 1993, pp. 261–270.

[16] B. Sanders, *A predicate transformer approach to knowledge and knowledge-based protocols*, Proceedings of the 10th ACM Symposium on Principles of Distributed Computing, 1991, pp. 217–230.

[17] F. Stulp and R. Verbrugge, *A knowledge-based algorithm for the Internet protocol TCP*, Proceedings of the 4th Conference on Logics and the Foundations Game and Decisions Theory (LOFT 4), Torino, Italy, June 2000.

[18] M.Y. Vardi, *Implementing Knowledge-Based Programs*, Proceedings of the 6th Conference on Theoretical Aspects of Rationality and Knowledge (TARK), 1996, pp. 15–30.