Knowledge-Based Asynchronous Programming

Haan, Hendrik Wietze de; Hesselink, Wim H.; Renardel de Lavalette, Gerard R.

Published in:
Fundamenta Informaticae

Publication date:
2004

Fundamenta Informaticae 63 (2004) 259–281
IOS Press

Knowledge-Based Asynchronous Programming

Hendrik Wietze de Haan, Wim H. Hesselink and Gerard R. Renardel de Lavalette*
Department of Mathematics and Computing Science
University of Groningen,
P.O. Box 800, 9700 AV Groningen, The Netherlands
{*hwh,wim,grl*}*@cs.rug.nl*

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 define the semantics of knowledge-based programs via an iteration approach generalizing the well-known fixpoint construction. We propose a specific iteration as the semantics of a knowledge-based program, and justify our choice by a number of examples, including the Unexpected Hanging Paradox.

Keywords: knowledge-based programming, semantics of programming languages, concurrent programming, asynchronicity, unexpected hanging paradox.

1. Introduction

A knowledge-based program (KBP) is a program with explicit tests for knowledge. KBPs can be used as high-level descriptions of the behaviour of agents in a multi-agent system. The general idea is that agent i knows φ iff φ holds in all situations that i considers possible. The *de facto* standard framework for KBPs in multi-agent systems is the theory of systems and runs as described in the book *Reasoning About Knowledge* by Fagin, Halpern, Moses and Vardi ([8]). In this framework, the meaning (semantics) of a KBP is formulated in terms of the meaning of joint protocols, which are composed from individual (nondeterministic but sequential) protocols via synchronous parallellism.

The purpose of this paper is twofold: we want to use asynchronous parallellism as the basis for KBPs, and we intend to resolve the inherent circularity in the definition of the meaning of KBPs: on

*Address for correspondence: Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands

the one hand, the meaning of a KBP depends on the meaning of the knowledge operators for the agents involved, on the other hand, the meaning of the knowledge operators depends on the collection of possible executions of the KBP, i.e. on the meaning of the KBP. In [8], this circularity is resolved by giving an implicit definition of the meaning of a KBP (see Section 3 below). As a consequence, several KBPs have a unique meaning, some have no meaning at all, and others have more than one meaning. Our approach is to observe that the implicit definition of the meaning of a KBP can be read as a fixed point of an automorphism on the collection of possible meanings, and to define the *unique* meaning of the KBP in question as either the greatest fixed point of the automorphism if it exists, or a well-chosen iteration of the automorphism.

In our investigation, we bring together three traditions, that of knowledge-based programming, e.g. [8], that of semantics of sequential programming languages, e.g. [3, 14], and the semantics of concurrent programming, e.g. [1, 2].

Our reasons for this investigation are the following. Firstly, in [11] we were inspired by a problem from delay-insensitive circuits, which are inherently asynchronous. Asynchronous parallellism is, in general, more realistic in the context of multi-agent systems than synchronous parallellism. It is natural to assume that agents have different processing speeds. Asynchronicity was therefore our primary design goal. In combination with interleaving semantics, where the atomic actions are serialized in arbitrary order, it is the usual framework for concurrent programming [1, 2]. This has the technical advantage that the designer need not consider joint actions. The stutterings of [1] are introduced in our framework to model that agents cannot know the computation speeds of other agents.

Secondly, we are interested in the semantics of KBPs and proof rules for KBPs. For program design it is desirable to work with high-level programs as specifications. It is then useful that these programs have a unique meaning, so one can reason about the system under design, e.g. using proof rules based on this unique meaning. Since we aim at design by specification, we do not want to define the meanings of KBPs by means of standard protocols as in [8]. Therefore, we adapt the framework of Fagin et al. in such a way that we can eliminate the intermediate protocols.

Our investigation results in a number of semantic dilemmas. We therefore refrain from adding the complications of temporal operators and fairness. Consequently, we can restrict to finite prefixes of runs. So, instead of runs we use finite sequences of states, called traces. We prefer to investigate the best-possible semantics thoroughly, before making simplifications that would enable model checking.

1.1. Overview

In Section 2 we discuss related work. The general approach of Fagin et al. is outlined in Section 3. We adapt this general framework to suit our needs in Section 4. We define a language of KBPs, assign an interpretation to this language and propose how to assign semantics to all KBPs. Our choice of semantics is justified in Section 5 via a number of examples. Some conclusions and directions for future research are discussed in Section 6.

2. Several approaches to knowledge-based programming

The first papers on knowledge in distributed systems appear in the ’80s: Chandy and Misra [5] describe how processes gain and lose knowledge, Katz and Taubenfeld [17] 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 [20] defines 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. Halpern and Zuck [12] successfully use 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 [21] show that real-life protocols can be analyzed using a knowledge-based approach by presenting a KBP for the TCP-protocol.

Moses and Kislev [19] introduce 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 *notify*(j, φ) 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 [4] and Van Ditmarsch [6]. The implementation of knowledge-oriented programs is not considered.

The book *Reasoning about Knowledge* [8], 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. We briefly review this general framework in the next section. In [8, 9], sufficient conditions for the existence of a well-defined implementation of a KBP are given; these conditions only apply to synchronous systems, however. In [9] the complexity of determining whether a KBP has a well-defined implementation in a given finite state context is characterized. In [22], Vardi investigates the complexity of checking whether a protocol implements a KBP in a given finite state context.

Van der Meyden describes in [18] 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 [7] a refinement calculus for KBPs. Their framework assumes that agents and environment act synchronously.

3. The general framework of Fagin, Halpern, Moses and Vardi

In this section we give a concise presentation of the relevant material from chapters 4, 5 and 7 of the book *Reasoning about Knowledge* [8], by Fagin et al.

3.1. Global states, local states, runs and interpreted systems

A multi-agent system consists of an number of agents $A = \{1, \ldots, n\}$ in an environment. At each moment in time the system is in a certain *global state* $\in \mathcal{G}$:

$$\mathcal{G} = L_e \times L_1 \times \cdots \times L_n$$

where for $i \in A$, L_i is the set of *local states* of agent i, and L_e is the set of local states of the environment. If $s \in \mathcal{G}$ and $i \in A$ then s_i is the $i+1$-th component of s, the local state of agent i.

A *run* captures how the global state changes over time. So, a run r is a function from $\mathbb{N}$ to $\mathcal{G}$ (time is taken to range over the natural numbers). A *point* (r, m) is a run r together with a time m, and the corresponding global state is $r(m)$. A *system* $\mathcal{R}$ is a nonempty set of runs.

An *interpreted system* $\mathcal{I}$ is a system $\mathcal{R}$ together with an *interpretation function* $\pi : \mathcal{G} \to (\Phi \to \{\mathit{true}, \mathit{false}\})$, that assigns a valuation to a set Φ of primitive propositions in each global state. The set Φ may be partitioned into sets of local primitive propositions, i.e. $\Phi = \cup_{i \in A} \Phi_i$. An interpretation π is *compatible* if π is generated from local interpretations $\pi_i : L_i \to \Phi_i \to \{\mathit{true}, \mathit{false}\}$. So, for compatible π, we have that

$$\pi(s)(p) = \pi_i(s_i)(p) \text{ if } p \in \Phi_i. \tag{1}$$

Definition. Given a set Φ of primitive propositions, the language $\mathcal{L}(\Phi)$ of *epistemic formulas* is defined in BNF-notation as

$$\varphi ::= \bot \mid p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid K_i\varphi \tag{2}$$

where $p \in \Phi$, $i \in A$. The connectives $\vee, \rightarrow, \leftrightarrow$, and the constant $\top$ are defined as usual in terms of $\bot$, $\neg$ and $\wedge$. K_i is the knowledge operator for agent i. The possibility operator M_i is the dual of K_i: $M_i\varphi = \neg K_i \neg\varphi$.

An interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ induces a *Kripke structure* $M_{\mathcal{I}} = \langle S, \pi, \sim_1, \ldots, \sim_n \rangle$ in the following way. S is the set of all points occurring in the runs of $\mathcal{R}$. The *indistinguishability relation* $\sim_i$ for agent i is defined on points as

$$(r, m) \sim_i (r', m') \Leftrightarrow (r(m))_i = (r'(m'))_i$$

That is, agent i cannot distinguish between two global states s and s' if she has the same local state in both s and s'.

The *modeling relation* $\models$ for an interpreted system $\mathcal{I}$ is now taken to be the modeling relation of the associated Kripke model $M_{\mathcal{I}}$. More formally, if $\mathcal{I}$ is a interpreted system and (r, m) is a point of $\mathcal{I}$, then

$$(\mathcal{I}, r, m) \models \varphi \Leftrightarrow (M_{\mathcal{I}}, r(m)) \models \varphi$$

for φ in $\mathcal{L}(\Phi)$. That is

$$\begin{aligned}
(\mathcal{I}, r, m) \models p \quad &\Leftrightarrow \quad \pi(r(m))(p) = \mathit{true}, \text{ for } p \in \Phi \\
(\mathcal{I}, r, m) \models K_i\varphi \quad &\Leftrightarrow \quad \forall\, r' \in \mathcal{R}\ \forall\, m'\ (r'(m') \sim_i r(m) \Rightarrow (\mathcal{I}, r', m') \models \varphi)
\end{aligned}$$

For the temporal operators $\Box$ (always) and $\bigcirc$ (next), we have

$$\begin{aligned}
(\mathcal{I}, r, m) \models \Box\varphi \quad &\Leftrightarrow \quad \forall\, m' \geq m\ (\,(\mathcal{I}, r, m') \models \varphi\,) \\
(\mathcal{I}, r, m) \models \bigcirc\varphi \quad &\Leftrightarrow \quad (\mathcal{I}, r, m+1) \models \varphi
\end{aligned}$$

but we shall not use $\Box$ and $\bigcirc$ in the rest of this paper.

If φ is an atemporal, epistemic formula (i.e. contains no temporal operators) then $(\mathcal{I}, r, m) \models \varphi$ only depends on $\mathcal{I}$ and the global state $s = r(m)$, and we may write $(\mathcal{I}, s) \models \varphi$ in that case. Moreover, if φ is a propositional formula (i.e. contains neither epistemic nor temporal operators) then $(\mathcal{I}, s) \models \varphi$ only depends on the interpretation π instead of the entire interpreted system $\mathcal{I}$, and we may write $(\pi, s) \models \varphi$ in that case. Finally, if φ is a propositional formula over Φ_i and π is a compatible interpretation, then $(\pi, s) \models \varphi$ only depends on agent i's local state and so we may write $(\pi, s_i) \models \varphi$ in that case.

3.2. Protocols and standard programs

For each agent i, a set ACT_i of *actions* available for that agent is given; the environment can execute actions from the set ACT_e. It is assumed that each set of actions contains a null-action, Λ, which has no effect. A *joint action* is tuple $(a_e, a_1, \ldots, a_n)$ of actions of the environment and the agents. ACT is the set of joint actions. There is a *transition function* $\tau : ACT \to \mathcal{G} \to \mathcal{G}$ that indicates how joint actions may change the global state.

A *protocol* is a function from local states to nonempty sets of actions, i.e. a protocol for agent i is a function $P_i : L_i \to (\mathcal{P}(ACT_i) - \{\emptyset\})$. A *joint protocol* is a tuple $(P_1, \ldots, P_n)$ of protocols for each of the agents.

A *context* γ is a tuple $(P_e, \tau, \mathcal{G}_0, \Psi)$, consisting of the protocol P_e of the environment, the transition function τ, a set of initial global states $\mathcal{G}_0 \subseteq \mathcal{G}$, and an admissibility restriction $\Psi \subseteq (\mathbb{N} \to \mathcal{G})$ on runs.

A run r is *consistent* with joint protocol $P = (P_1, \ldots, P_n)$ in context $\gamma = (P_e, \tau, \mathcal{G}_0, \Psi)$ if

- $r(0) \in \mathcal{G}_0$,
- $\forall\, m \in \mathbb{N}\, \exists \vec{a} \in \vec{P}(r(m))\, \exists a_e \in P_e(r_e(m))\ (r(m+1) = \tau(a_e, \vec{a})(r(m)))$,
- $r \in \Psi$.

where $\vec{P}(s)$ is the result of applying the joint protocol to a global state:

$$\vec{P}(s_e, s_1, \ldots, s_n) = P_1(s_1) \times \cdots \times P_n(s_n).$$

Observe that the second formula in the definition of consistency implies that the joint protocol is executed not once, but infinitely often.

The *representing system* $\mathcal{R}^{\mathrm{rep}}(P, \gamma)$ of joint protocol P in context γ is defined by

$$\mathcal{R}^{\mathrm{rep}}(P, \gamma) = \{r \in \mathbb{N} \to \mathcal{G} \mid r \text{ is consistent with } P \text{ in } \gamma\}$$

Protocols are represented by programs. A *standard program* Pg_i for agent i has the following form

case of
  if t_1 **do** a_1
  $\vdots$
  if t_m **do** a_m
end case

where the t_j are propositional formulas over Φ_i and the a_j are in ACT_i.

Given a program Pg_i and a compatible interpretation π, the *associated protocol* Pg_i^π in local state $l \in L_i$ is defined as

$$Pg_i^\pi(l) = \begin{cases} \{a_j \mid (\pi, l) \models t_j\} & \text{if } \{j \mid (\pi, l) \models t_j\} \neq \emptyset \\ \{\Lambda\} & \text{otherwise} \end{cases}$$

The compatibility of π (see (1)) ensures that the outcome of tests t_j in program Pg_i under π only depends on the local state of agent i, so we may write $(\pi, l) \models t_j$ for $(\mathcal{I}, r, m) \models t_j$. So the associated protocol

chooses nondeterministically one of the actions a_j for which the test t_j holds; if there is no such a_j then the empty action Λ is chosen.

A *joint program* is a tuple $Pg = (Pg_1, \ldots, Pg_n)$. Given an interpretation π, compatible with every program of the joint program, the joint protocol Pg^π denoted by *Pg* is defined straightforwardly.

An *interpreted context* is a context γ with an interpretation π, that is a tuple (γ, π). The interpreted system of all runs that are consistent with P in interpreted context (γ, π) is denoted by

$$\mathcal{I}^{\mathrm{rep}}(P, \gamma, \pi) = (\mathcal{R}^{\mathrm{rep}}(P, \gamma), \pi).$$

This is the interpreted system that represents P in interpreted context (γ, π).

Observe that the infinite repetition of joint program *Pg*, which is a consequence of the definition of their interpretation in terms of consistent runs, is left implicit in the notation. The same holds for the knowledge-based programs defined in [8]: see the next section. We shall make this repetition explicit in our adapted definition of knowledge-based programs in Section 4.4.

3.3. Knowledge-based programs

A *knowledge-based program* (KBP) for agent i is of the form

case of
  if $t_1 \wedge k_1$ **do** a_1
  $\vdots$
  if $t_m \wedge k_m$ **do** a_m
end case

where the t_j are propositional formulas over Φ_i, the a_j are in ACT_i and the $k_j \in \mathcal{L}(\Phi)$ are epistemic formulas as defined in (2). A *joint KBP* is a tuple of KBPs.

In contrast to the propositional formulas t_j, the outcome of epistemic formulas k_j does not only depend on the interpretation π and the local state of the agent, but on the entire interpreted system. So the joint protocol $Pg^{\mathcal{I}} = (Pg_1^{\mathcal{I}}, \ldots, Pg_n^{\mathcal{I}})$ has $\mathcal{I}$ instead of π as a parameter. As before, the interpretation π in $\mathcal{I}$ should be compatible with *Pg* w.r.t. the propositional formulas t_j. There is no restriction on the propositions in the epistemic formulas k_j.

The modeling relation is extended to evaluate epistemic formulas for agent i in a local state $l \in L_i$ for a compatible interpretation π. If φ is a propositional formula, then $(\mathcal{I}, l) \models \varphi$ iff $(\pi, l) \models \varphi$. For epistemic formulas,

$$(\mathcal{I}, l) \models K_i\psi \quad \Leftrightarrow \quad \forall\, r, m\ (\,(r, m) \in \mathcal{I} \wedge (r(m))_i = l \ \Rightarrow\ (\mathcal{I}, r, m) \models \psi\,)$$

A protocol $Pg_i^{\mathcal{I}}$ is associated with KBP Pg_i for $l \in L_i$ as

$$Pg_i^{\mathcal{I}}(l) = \begin{cases} \{a_j \mid (\mathcal{I}, l) \models t_j \wedge k_j\} & \text{if } \{j \mid (\mathcal{I}, l) \models t_j \wedge k_j\} \neq \emptyset \\ \{\Lambda\} & \text{otherwise } \emptyset \end{cases}$$

An interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ represents a KBP *Pg* in context (γ, π) iff π is compatible with *Pg* and $\mathcal{I}$ represents the associated protocol $Pg^{\mathcal{I}}$. That is, $\mathcal{I}$ represents *Pg* in (γ, π) iff

$$\mathcal{I} = \mathcal{I}^{\mathrm{rep}}(Pg^{\mathcal{I}}, \gamma, \pi) \tag{3}$$

Observe that we have a circularity here: to determine whether an interpreted system $\mathcal{I}$ represents *Pg* we must verify whether the runs of $\mathcal{I}$ are consistent with $Pg^{\mathcal{I}}$. So this is an implicit definition, and there need not be a unique interpreted system that represents a KBP in a given context.

This ends our resumé of the theory of systems and runs as given in [8].

4. Asynchronous KBPs

For our treatment of asynchronous KBPs, we shall adapt the framework of systems and runs given in the previous section. The main changes are the following.

- We combine actions via interleaving with possible delay (and not via joint actions).
- We work with a more general programming language, inspired by dynamic logic (see [13]) which is interpreted directly (i.e. not via protocols) in a Kripke model.
- The indistinguishability relations $\approx_i$ in our Kripke model are relations on runs (not on states), so we assume that the agents can recall their own history when comparing points in the model.
- In the definition of the interpretation of programs in our Kripke model, we introduce an explicit parameter B to restrict the collection of runs under consideration when interpreting the knowledge operators K_i; via abstraction from B, we obtain an automorphism on our Kripke model which is the starting point for the definition of the (unique and parameter-free) meaning of our programs.

Some minor changes are: we consider the environment as an agent like the others, we shall shift from infinite runs to finite traces (both approaches are equivalent when restricted to atemporal formulas), and we work with a slightly different notion of context.

4.1. Asynchronicity via stutterings

To model asynchronicity, we work with an interleaving semantics: the single actions of the individual programs of the agents are performed consecutively in an unspecified order, possibly with delay. As a consequence, an agent cannot distinguish two runs $r, r' \in (\mathbb{N} \to \mathcal{G})$ if r' is obtained from r by consecutive repetition of certain global states $s \in G$.

We illustrate this with an example. Assume that agent a can see the contents of numerical variable x, and that agents b and c can modify x: b can increase it with 2, c can decrease it with 1. In our setting, $02133333\ldots$ is a possible run, where every step corresponds with either an action of b, an action of c or no action; $013555555\ldots$ is not possible, for the first step (going from 0 to 1) requires combining the actions of b and c in one joint action, which is not allowed in interleaving semantics. Moreover, the runs $r = 021354333\ldots$ and $r' = 0002133333555443333\ldots$ cannot be distinguished by agent a, for both correspond with actions from b, c, b, b, c, c, respectively, and we abstract away from the delay between some of the actions that occurred in r'.

Definition. A run r_2 is a *stuttering* of r_1, notation $r_1 \preceq r_2$, iff r_2 is obtained from r_1 by consecutive, finite repetition of certain elements of r_1. In formula:

$$r_1 \preceq r_2 \quad \equiv \quad \exists f : \mathbb{N} \to \mathbb{N}\ (f \text{ monotonic and surjective, and } r_2 = r_1 \circ f) \tag{4}$$

For example, r' of the example above is a stuttering of r, but $0211111\ldots$ is not (infinite repetition of 1), nor $012354333\ldots$ (the order of the elements is not preserved). The concept of stuttering comes from [1], and the relation $\preceq$ was introduced in [15]. We claim that $\preceq$ is a partial order: reflexivity and transitivity follow directly from the definition, antisymmetry is a nice nontrivial exercise.

With this notion of stuttering, we can define the appropriate indistinguishability relation for agents in the context of asynchronous KBPs.

Definition. Agent i considers run r_1 indistinguishable from run r_2 (notation: $r_1 \approx_i r_2$) when both runs can be stuttered to runs r_1' and r_2' such that agent i cannot distinguish individual states along r_1' and r_2'. So we have

$$r_1 \approx_i r_2 \quad \equiv \quad \exists r_1', r_2' \,(\, r_1 \preceq r_1' \,\wedge\, r_2 \preceq r_2' \,\wedge\, r_1' \sim_i r_2' \,). \tag{5}$$

In short: $(\approx_i) = (\preceq \circ \sim_i \circ \succeq)$, where $\circ$ denotes relational composition. Here we use the straightforward lifting of $\sim_i$ to runs, defined by

$$r_1 \sim_i r_2 \quad \equiv \quad \forall\, n(\, r_1(n) \sim_i r_2(n)\,)\ .$$

We show that $\approx_i$ is an equivalence relation. It is easy to see that $\approx_i$ is reflexive and symmetrical, since $\sim_i$ is reflexive and symmetrical and $\preceq$ is reflexive. For transitivity, i.e. $(\approx_i \circ \approx_i) \ \subseteq\ (\approx_i)$, we use the confluence of $\preceq$ and the fact that $(\sim_i \circ \preceq) \ \subseteq\ (\preceq \circ \sim_i)$:

$$\begin{array}{ll}
 & \approx_i \circ \approx_i \\
= & \quad \{\text{definition}\} \\
 & \preceq \circ \sim_i \circ \succeq \circ \preceq \circ \sim_i \circ \succeq \\
\subseteq & \quad \{\text{confluence of } \preceq\text{: } (\succeq \circ \preceq) \ \subseteq\ (\preceq \circ \succeq)\} \\
 & \preceq \circ \sim_i \circ \preceq \circ \succeq \circ \sim_i \circ \succeq \\
\subseteq & \quad \{(\sim_i \circ \preceq) \ \subseteq\ (\preceq \circ \sim_i) \text{ and } (\succeq \circ \sim_i) \ \subseteq\ (\sim_i \circ \succeq)\} \\
 & \preceq \circ \preceq \circ \sim_i \circ \sim_i \circ \succeq \circ \succeq \\
= & \quad \{\text{transitivity of } \sim_i \text{ and } \preceq\} \\
 & \preceq \circ \sim_i \circ \succeq \\
= & \quad \{\text{definition}\} \\
 & \approx_i
\end{array}$$

4.2. A Kripke model of traces

In this paper, we study only KBPs with purely epistemic tests (the extension to tests with temporal operators is an interesting subject for further research). In the last paragraph of Subsection 3.1, we observed that in that case we may restrict ourselves, without loss of generality, to finite nonempty prefixes of runs, i.e. *traces* $xs \in \mathcal{G}^+$. We shall do so from now on, and introduce some notation. We write $\ell(xs)$ for the length of the trace, and xs_j for its jth element (where $0 \leq j < \ell(xs)$). 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 $*$. The definition of stuttering and indistinguishability for traces is analogous to that of runs.

We intend to define the semantics of KBPs, given a context $\gamma = (\mathcal{G}, \pi, \tau, \mathcal{G}_0)$. As before, $\mathcal{G}$ is the collection of global states, $\pi : \mathcal{G} \to (\Phi \to \{true, false\})$ is a valuation of atomic propositions, $\tau : ACT \to \mathcal{G} \to \mathcal{G}$ is a transition function for the interpretation of atomic actions and $\mathcal{G}_0 \subseteq \mathcal{G}$ is a collection of initial states.

We now present the Kripke model $M = \langle \mathcal{G}^+, \pi, (\approx_i)_{i \in A} \rangle$ for the interpretation of knowledge formulas and knowledge-based programs. The set of worlds is the collection $\mathcal{G}^+$ of all traces over $\mathcal{G}$. π is a valuation given by the context, and $(\approx_i)_{i \in A}$ is the collection of equivalence relations on traces defined above.

The interpretation of epistemic formulas in M is defined as follows. We use a parameter B, denoting the collection of traces *under consideration*, to resolve the mutual dependency between the definition of the semantics of the knowledge operator and the definition of the collection of possible traces. For $B \subseteq \mathcal{G}^+$, the interpretation $[\![\varphi]\!]_B \subseteq \mathcal{G}^+$ is defined inductively by

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

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

Observe that the role played by B differs from the more global role of $\mathcal{R}$ in the definition of the interpretation of formulas in interpreted system $\mathcal{I} = (\mathcal{R}, \pi)$ used in [8] (see Section 3). Here, we work in the Kripke model of *all* possible traces, and B is only used as a restriction in the definition of the interpretation of K_i; as a consequence, we do not have $[\![\varphi]\!]_B \subseteq B$ in general. This is a deliberate choice to be justified later by means of the example programs in Section 5.

4.3. Epistemic programs

Definition. Given a set *ACT* of atomic actions a, the language of *epistemic programs* is defined as

$$\alpha ::= \Lambda \mid a \mid \varphi? \mid (\alpha; \alpha) \mid (\alpha \cup \alpha) \mid \alpha^* \tag{6}$$

where $a \in \mathit{ACT}$, φ an epistemic formula. So a program is either a null action Λ, an atomic action a, a test on φ, the composition of two programs, the nondeterministic choice between two programs, or the repetition (zero or more times) of a program.

The interpretation of an epistemic program is a binary relation on traces. As for formulas, we have a parameter $B \subseteq \mathcal{G}^+$ that represents the traces under consideration. Now the definition of $[\![\alpha]\!]_B \subseteq \mathcal{G}^+ \times \mathcal{G}^+$ reads

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

We give some explanation. Λ (doing the empty action) corresponds with a stuttering step: the last (i.e. current) state is repeated once. Observe that Λ is different from $\top?$ (not acting), since $[\![\top?]\!]_B =$

$\{(xs, xs) \mid xs \in \mathcal{G}^+\}$. The interpretation of atomic actions is lifted to traces: two traces *xs* and *ys* are related via action a iff *ys* is equal to *xs* appended with an outcome of a applied to the last element in *xs*. The interpretation of the program constructs (composition, choice and repetition) follows the usual treatment in dynamic logic.

4.4. Knowledge-based programs

Recall the definition of knowledge-based program Pg_i of agent $i \in A = \{1, \ldots, n\}$, given in Section 3.3. In our formalism, it can be rendered as the repetition of a nondeterministic choice of guarded atomic actions:

$$Pg_i = (\bigcup_{j \in J_i} (\varphi_j?; a_j))^*$$

where J_i is some finite index set for $i \leq n$. The implicit repetition of the KBPs of Section 3.3 is made explicit here by Kleene's star. In interleaving semantics, the parallel composition *Pg* of $Pg_1, \ldots, Pg_n$ is a nondeterministic repetition of all the alternatives $\varphi_j?; a_j$ that occur in the different programs, with the alternative Λ added to model possible delay. So

$$Pg \;=\; (\,\Lambda \;\cup \bigcup_{j \in J} (\varphi_j?\,;\; a_j)\,)^* \tag{7}$$

where $J = J_1 \cup \cdots \cup J_n$. In the rest of this paper, we restrict ourselves to programs of this form, which we call *asynchronous* KBPs. The interpretation $[\![Pg]\!]_B$ can be rewritten as

$$\begin{aligned}[\![Pg]\!]_B = \{(xs{*}x_0,\, xs{*}x_0{*}x_1{*}\cdots{*}x_n) \mid xs \in \mathcal{G}^* \wedge n \geq 0 \wedge \\ \forall\, k < n\,(x_k = x_{k+1} \vee \exists j\,(xs{*}x_0{*}x_1{*}\cdots{*}x_k \in [\![\varphi_j]\!]_B \wedge x_{k+1} = \tau(a_j)(x_k)))\}\end{aligned} \tag{8}$$

That is, $[\![Pg]\!]_B$ consists of pairs of traces (ys, zs) where *zs* is an extension of *ys*, such that the new subsequent states are either the result of stuttering (i.e. doing the empty action Λ), or of some atomic action a_j in *Pg* with a guard φ_j that holds in the trace up to that state.

Now the *set of traces generated by Pg*, given a context $\gamma = (\mathcal{G}, \pi, \tau, \mathcal{G}_0)$ and a set B of traces under consideration, is defined as the set of all traces that are generated by *Pg* when starting in an initial state $y \in \mathcal{G}_0 \subseteq \mathcal{G}$:

$$\mathcal{G}_0[\![Pg]\!]_B = \{xs \mid \exists y\,(y \in \mathcal{G}_0 \wedge (y, xs) \in [\![Pg]\!]_B)\} \tag{9}$$

We would like 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 $\mathcal{G}_0[\![Pg]\!]_B = B$. This is a fixpoint characterization of the semantics of program *Pg*. It is our analogue to equation (3) from Section 3.3.

4.5. Choosing a semantics for KBPs

In this section we set out to find a unique, parameter-free semantics for asynchronous KBPs of the form given in (7). For that purpose, we define an automorphism $F : \mathcal{P}(\mathcal{G}^+) \to \mathcal{P}(\mathcal{G}^+)$, based on the definition of the trace set of *Pg* given in (9):

$$F(B) = \mathcal{G}_0[\![Pg]\!]_B \tag{10}$$

If F is monotonic (in the sense that $F(B) \subseteq F(B')$ whenever $B \subseteq B'$), then the theorem of Knaster-Tarski tells us that F has a unique least fixpoint and a unique greatest fixpoint. In that case, we prefer to define the semantics of *Pg* as the *greatest* fixpoint, since that would be the most liberal interpretation of *Pg*. In general, however, 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 show this in the examples of Section 5.

Fixpoint equations are common in the study of the semantics of sequential programs with loops or recursive procedures (see e.g. [3, 14]), and the fixpoint is usually approximated by iterations of the fixpoint operator. We generalize that idea in a more general setting here, viz. when there may be no fixpoint to approximate.

Definition. The iterations (B_λ) are defined by transfinite induction over the ordinals:

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

where F is as defined in (10).

The first iteration B_0 consists of all nonempty finite sequences of states. For any ordinal λ, the iteration $B_{\lambda+1}$ consists of traces generated by the program when B_λ is used to evaluate epistemic formulas. When λ is a limit ordinal, the iteration B_λ is the intersection of unions of iterations that are sufficiently close to the limit. We motivate the definition of B_λ for a limit ordinal λ as follows. If the iteration sequence forms a descending chain, 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, iterations may also grow and shrink, and therefore a union of intersections or an intersection of unions is indicated. 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: it includes more traces, more traces implies less knowledge and we want the agents to know facts only when there are good reasons for them.

A cardinality argument implies that the transfinite sequence of contains multiple elements: since all sets B_λ are subsets of $\mathcal{G}^+$ and there exist more ordinals than $\mathcal{G}^+$ has subsets, the sets B_λ cannot all be different. This implies the existence of ordinals κ and μ with $\kappa < \mu$ and $B_\kappa = B_\mu$. By well-foundedness, there is a least such κ. Now let κ be minimal with the property that $B_\kappa = B_\mu$ for some $\mu > \kappa$. If $F(B_\kappa) = B_\kappa$, i.e. B_κ is a fixpoint, then 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 (such a λ always exist when B_κ is not a fixpoint). This latter choice is justified by the argument that it allows as much well-justified knowledge as possible without introducing contradictory knowledge.

Definition. We define

$$\text{sem}(Pg) = B_\lambda \tag{11}$$

where

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

Since the iteration sequence has multiple elements, it has the shape of a 6, see figure 1. All iterations between B_κ and B_λ are subsets of previous iterations, and B_λ is the last one for which this holds.

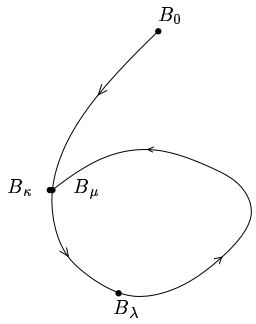

Figure 1. The iteration sequence. κ is the least ordinal such that $\exists \mu > \kappa.(B_\kappa = B_\mu)$

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

$$Pg_\flat \;=\; (\,\Lambda \,\cup\, \bigcup_{j \in J} a_j\,)^*$$

We put $B_\flat = [\![Pg_\flat]\!]$; observe that $[\![Pg_\flat]\!]_B = [\![Pg_\flat]\!]_{B'}$ for all $B, B' \subseteq \mathcal{G}^+$, so we may drop the parameter B here.

5. Examples

In this section we investigate the consequences of our formalism by means of a number of examples. The first three examples serve to show that function F can be monotonic or not monotonic, and may have incomparable fixpoints or no fixpoints at all. The fourth example shows that the iteration sequence may become infinite. The fifth example demonstrates how message passing can be modelled. The final example is a formulation of the Unexpected Hanging Paradox as an asynchronous KBP. For every example the iteration sequence is also given as a figure.

5.1. Failure of monotonicity

Suppose there are two agents 1 and 2 and two boolean variables p, q, initially true. We represent truth values as the integers 0 and 1, so the state space is $\mathcal{G} = \{0,1\} \times \{0,1\}$ 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 1, and q is private to agent 2. That is, agent 1 can only see and write p, agent 2 can only see and write q. We take the indistinguishability relations on states to be defined by

$$(p,q) {\sim_1} (p',q') \Leftrightarrow p = p', \quad (p,q) {\sim_2} (p',q') \Leftrightarrow q = q'.$$

Here and henceforth we use p, p', q, q' to denote the values of the program variables p and q in the states considered.

The agents execute KBPs. Agent 1 can falsify p by setting p to zero when she knows that q holds. Agent 2 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

$$Pg = (\ \Lambda\ \cup\ K_1 q\,?\ ;\ p := 0\ \cup\ K_2 p\,?\ ;\ q := 0\ )^*.$$

By definition $B_0 = \mathcal{G}^+$. For the next iteration, the traces in B_0 are used to evaluate epistemic tests. When $\mathcal{G}^+$ is considered, agents have only knowledge of the values of their own private variables. That is, if $B = \mathcal{G}^+$ then $[\![K_1 q]\!]_B = \emptyset$ and $[\![K_2 p]\!]_B = \emptyset$. 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 $[\![K_1 q]\!]_B = [\![K_2 p]\!]_B = \mathcal{G}^+$. 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

$$(\ \Lambda\ \cup\ p := 0\ \cup\ q := 0\ )^*.$$

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 $[\![K_1 q]\!]_B = [\![K_2 p]\!]_B = \emptyset$. Therefore $B_3 = F(B_2) = (1,1)^+$, and so $B_3 = B_1$ and the iteration sequence starts to repeat itself. It seems reasonable to take B_1 as the correct meaning of the program. Since B_2 is strictly larger than B_1, this is in accordance with the definition of the semantics given in (11).

The above iterations 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 $[\![K_1 q]\!]_{B'} = \mathcal{G}^+$. Therefore, agent 1 can set p to zero at any time. Traces matching $(1,1)^+$ or $(1,1)^+(0,1)^+$ are both indistinguishable for agent 2 from traces of the form $(1,1)^+(0,1)^+$, and therefore do not satisfy $K_2 p$ under B'. Agent 2 cannot set q to zero in the initial state nor after agent 1 has reset p, so $F(B') = B'$. Analogously, $F(B'') = B''$.

The iteration sequence and the fixpoints B' and B'' are sketched in figure 2. The dashed line indicates the subset relation: the lower set is a subset of the upper set. Arrows give the direction of iteration, i.e. application of F.

5.2. A case with fixpoint semantics

This example has the same setting as the previous example. There are two agents 1 and 2, with private boolean variables p respectively q. The state space is as before, but now the initial state is $(0,0)$. An

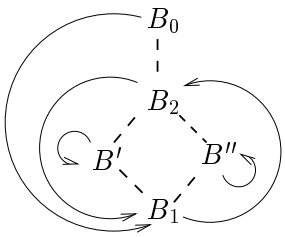

Figure 2. Example 5.1: iteration sequence and the fixpoints B' and B''

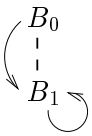

Figure 3. Example 5.2: iteration sequence that ends in a fixpoint

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

$$Pg = (\ \Lambda\ \cup\ M_1 q\,?\ ;\ p := 1\ \cup\ M_2 p\,?\ ;\ q := 1\ )^*.$$

Again we determine a sequence of iterations. When $B_0 = \mathcal{G}^+$ is considered, agents have only knowledge of their private variables. If agent 1 does not know whether q holds then she considers both q and $\neg q$ possible. So, if $B = \mathcal{G}^+$ then $[\![M_1 q]\!]_B = [\![M_2 p]\!]_B = \mathcal{G}^+$. The agents can independently set their private variables, so $B_1 = F(B_0)$ consists of the traces generated by

$$(\ \Lambda\ \cup\ p := 1\ \cup\ q := 1\ )^*.$$

In B_1, agents always consider it possible that the other has set her variable. Therefore, if $B = B_1$ then $[\![M_1 q]\!]_B = [\![M_2 p]\!]_B = \mathcal{G}^+$, so $B_2 = F(B_1) = B_1$, and we have reached a fixpoint; as such, it is the semantics of the program according to the definition of the semantics given in (11). (Note that $(0,0)^+$ is another fixpoint of F.) The iteration is sketched in figure 3.

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.

5.3. Absence of fixpoints

Again there are two agents 1 and 2 with private boolean variables p and q. Initially p is false and q is true. Now, agent 1 can set p when she knows that q holds, and agent 2 can reset q when she considers it possible that p has been set. The program is given as

$$Pg = (\ \Lambda\ \cup\ K_1 q\,?\ ;\ p := 1\ \cup\ M_2 p\,?\ ;\ q := 0\ )^*.$$

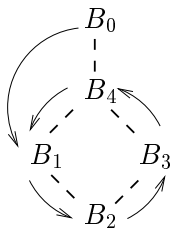

Figure 4. Example 5.3: a concrete six-shaped diagram as given in figure 1

As seen in the previous two examples, if $B = \mathcal{G}^+$ then $[\![K_1 q]\!]_B = \emptyset$ and $[\![M_2 p]\!]_B = \mathcal{G}^+$. Only agent 2 can act, so $B_1 = F(B_0)$ consists of traces of program

$$(\ \Lambda \ \cup \ q := 0 \ )^*.$$

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

For $B = B_1$, the interpretation $[\![K_1 q]\!]_B$ contains traces that cannot be generated by the program. For example, a trace *xs* matching $(1,0)^+$ satisfies $K_1 q$ under B_1, since no trace in B_1 is indistinguishable for agent 1 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 have been removed, that is the program

$$(\ \Lambda \ \cup \ p := 1 \ \cup \ q := 0 \ )^*.$$

No trace in $B_\flat$ satisfies $K_1 q$ under B_1. In other words, if $B = B_1$ then $B_\flat \cap [\![K_1 q]\!]_B = \emptyset$. No trace in B_1 satisfies p. Therefore, if $B = B_1$ then $[\![M_2 p]\!]_B = \emptyset$. Both agents can never 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 $[\![K_1 q]\!]_B = \mathcal{G}^+$ and $[\![M_2 p]\!]_B = \emptyset$. Only agent 1 can act. Therefore, $B_3 = F(B_2)$ consists of traces generated by program

$$(\ \Lambda \ \cup \ p := 1 \ )^*.$$

Since q is not reset, all traces in B_3 satisfy q, so if $B = B_3$ then $[\![K_1 q]\!]_B = \mathcal{G}^+$. Observe that there are traces that satisfy $M_2 p$ under B_3, but that are not in $B_\flat$. However, if $B = B_3$ then $B_\flat \subseteq [\![M_2 p]\!]_B$. Therefore, agent 2 can set q to zero under B_3, so $B_4 = F(B_3) = B_\flat$.

If $B = B_4$ then $[\![K_1 q]\!]_B = \emptyset$ and $B_\flat \cap [\![M_2 p]\!]_B = B_\flat$. Only agent 2 can act, therefore $B_5 = F(B_4)$ is equal to B_1. At this point the iteration sequence repeats. The sequence is drawn in figure 4.

In this example the semantics as defined in (11) yields B_2 as the semantics of the program, since $B_1 \supseteq B_2 \not\supseteq B_3$. Therefore, the semantics is that the system remains in its initial state.

5.4. An infinite iteration sequence

In this example we show that the iteration sequence may become infinite. There are again two agents 1 and 2. Agent 1 has private integer variables p and m, and agent 2 has private integer variables q and n. Initially $p = q = 0$ and $m = n = 1$. Agent 1 may always increment m. If agent 1 knows that q is

0, she can set p to 1. If agent 1 considers it possible that q is between 1 and m, she can advance p to $m+1$. Agent 2 can do similar actions. If we allow equalities and inequalities to appear in the epistemic formulas, then the KBP is

$$
\begin{array}{ll}
(& \Lambda \\
\cup & m := m+1 \\
\cup & n := n+1 \\
\cup & K_1(q=0)\,?\,;\; p := 1 \\
\cup & K_2(p=0)\,?\,;\; q := 1 \\
\cup & M_1(1 \le q \le m)\,?\,;\; p := m+1 \\
\cup & M_2(1 \le p \le n)\,?\,;\; q := n+1 \\
)^*. &
\end{array}
$$

Again $B_0 = \mathcal{G}^+$. When $\mathcal{G}^+$ is considered, agent 1 can set $p := m+1$. Since agent 1 cannot inspect q, she can always consider a trace in $\mathcal{G}^+$ possible such that $1 \le q \le m$. Agent 1 cannot set p to 1, because for $B = \mathcal{G}^+$ we have $[\![K_1(q=0)]\!]_B = \emptyset$. Agent 1 can always increment m. Similar arguments for agent 2 show that she cannot set q to 1. Therefore, $B_1 = F(B_0)$ consists of the traces of program

$$(\Lambda \cup m := m+1 \cup n := n+1 \cup p := m+1 \cup q := n+1)^*.$$

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

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

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

$$(\Lambda \cup m := m+1 \cup n := n+1)^*.$$

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

$$(\Lambda \cup m := m+1 \cup n := n+1 \cup p := 1 \cup q := 1)^*.$$

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

In this example, the semantics defined in (11) yields B_ω as the semantics of the program, since the sequence $B_1, B_2, \ldots, B_\omega$ is the longest decreasing sequence starting with $B_1 = B_{\omega+2}$. In this program agents cannot 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.

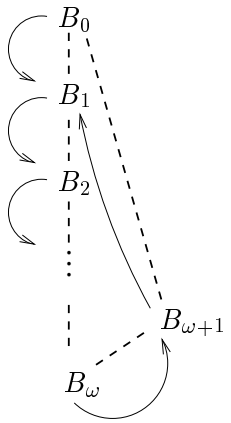

Figure 5. Example 5.4: an infinite sequence of iterations

5.5. Message passing

Message passing can be modelled by means of a variable that can be written by one agent and read by another. In this example, the agents 1 and 2 communicate via the integer variable q. Agent 1 has a private integer variable p and agent 2 has a private integer variable r.

All variables are initially 0. The state space consists of triples (p, q, r), and we have

$$(p, q, r) \sim_1 (p', q', r') \Leftrightarrow p = p' \wedge q = q',$$
$$(p, q, r) \sim_2 (p', q', r') \Leftrightarrow q = q' \wedge r = r'.$$

Agent 1 may increment p when she considers it possible that $p \leq r$. She may increment q when she knows that $q < p$. Agent 2 may increment r when she knows that $r < p$. The program is

$$\begin{array}{ll} Pg = & (\ \Lambda \\ & \cup M_1\,(p \leq r)\,?\ ;\ p := p + 1 \\ & \cup \quad\ (q < p)\,?\ ;\ q := q + 1 \\ & \cup\ K_2\,(r < p)\,?\ ;\ r := r + 1 \\ &)^* \ . \end{array}$$

Because agent 1 can read both p and q, the test $q < p$ is equivalent to $K_1(q < p)$.

Intuitively, one could expect the following behaviours from this program. Agent 1 may increment p, followed by q. Then agent 2 may increment r. Since agent 1 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. In appendix A.1 the iteration sequence, sketched in figure 6, is worked out. As seen in figure 6, a fixpoint occurs. It turns out that this fixpoint corresponds with our intuition.

5.6. The Unexpected Hanging Paradox

The last example is inspired on a well-known paradox, first mentioned in [16] as the case of the “Class A blackout”. Presently, it is commonly known as “The Surprise Examination”, or “The Unexpected Hanging”.

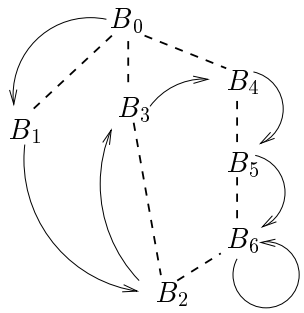

Figure 6. Example 5.5: nonmonotonic iteration that leads to a fixpoint.

In the unexpected hanging paradox, a convicted prisoner is to be executed at noon within seven days (numbered 0 to 6), but the judge tells him that he will not know the day of his execution, even on the beginning of that day itself. The prisoner might then reason that he cannot be executed on day 6, because when he would still be alive at the beginning of day 6, 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 $\{0, .., 6\}$. The convicted agent 1 can observe *day* and *dead*, but not *exec*. The program is

$$
\begin{array}{ll}
(& \Lambda \\
\cup & (\mathit{day} = \mathit{exec} \ \wedge \ \neg K_1(\mathit{day} = \mathit{exec}) \ \wedge \ \neg \mathit{dead})\,?\,;\ \mathit{dead} := 1 \\
\cup & (\mathit{day} \neq \mathit{exec} \ \vee \ \mathit{dead})\,?\,;\ \mathit{day} := \mathit{day} + 1 \\
)^* &
\end{array}
$$

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

We illustrate the iterations 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 iterations B_1 and B_2 are given in figures 7 and 8. The arrows corresponding to Λ are not shown. The dashed ellipses give the uncertainty of the agent, and are induced by $\sim_1$. 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 all days. In $B_2 = F(B_1)$, the agent cannot be executed on day 3, but he can be executed on the earlier days. The iteration B_2 is a fixpoint of F, and is chosen as the semantics of the program.

Technically, our formalism models deadlock by forced stutterings, since the alternative Λ 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 deadlock. This resolves the paradox. This is in agreement with the philosophical analysis based on dynamic

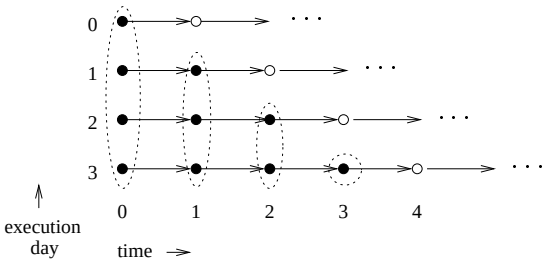

Figure 7. Traces of $B_1 = F(B_0)$ for *exec* $\in \{0, .., 3\}$

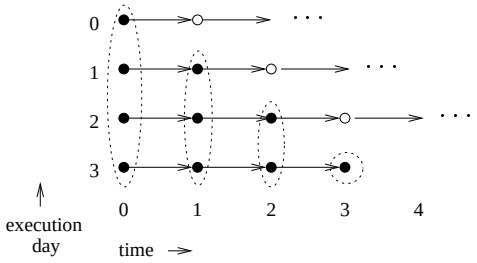

Figure 8. Traces of $B_2 = F(B_1)$ for *exec* $\in \{0, .., 3\}$

epistemic logic given in [10].

6. Conclusions and directions for future research

We have presented an asynchronous version of knowledge-based programming based on stutterings and interleaving semantics, that can be compared with the framework of systems and runs of Fagin et al. in [8]. The fixpoint equation that the semantics of asynchronous 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 proposed a method to assign semantics to all KBPs in our framework, by calculating a sequence of iterations. If this sequence ends in a fixpoint, that fixpoint is taken to be the semantics of the KBP. If not, we choose a specific iteration as the semantics. To justify this choice, a number of examples have been worked out.

We have not found a suitable subclass of KBPs for which the automorphism F is monotonic. It would be useful to study what conditions, if any, would guarantee monotonicity of F.

The idea to approximate the semantics via sets B_λ of traces under consideration came up while trying to attach meanings to the example programs of Section 5. Our semantics definition is a proposal: one could advocate other choices within the same or similar sequences of iterations, such as the re-entry iteration (i.e. the first iteration B_κ with $B_\kappa = B_{\kappa+\lambda}$ for some $\lambda > 0$) or the first local minimum (i.e. the first iteration B_κ for which $B_{\kappa+1} \not\subseteq B_\kappa$). The re-entry semantics gives a different meaning than our proposed semantics for the example of Section 5.4, and the semantics of the first local minimum differs from our proposed semantics in the example of Section 5.5. We find our semantics more intuitively

appealing than the re-entry or first local minimum semantics, but it is very well possible that our choice can be improved. This may especially be the case when attention is restricted to an interesting subclass of asynchronous KBPs.

We have not looked at simplifications of our semantics that would make model checking feasible. In general, we think that reasoning about a system during design (by using proof rules) is to be preferred above reasoning afterwards. As a consequence, our main concern is the development of useful proof rules. We must admit, however, to fear that the current semantics does not admit very useful proof rules, and it remains to be seen whether alternative and more useful semantics can be found. The relation with the predicate transformer approach of [20] could be explored in more detail.

In our current framework we do not consider temporal formulas. No fairness assumptions or progress properties can be expressed. It would be interesting to extend the framework to infinite traces (runs) and progress properties.

Another direction of research would be to study refinement relations between KBPs in our framework. A KBP can then gradually be refined to a knowledge-free program. This would eliminate the need to explicitly calculate an iteration sequence.

It may be useful to express the automorphism F defined in (10) in terms of automata. 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.

References

[1] M. Abadi and L. Lamport, *The existence of refinement mappings*, Theoretical Computer Science 82, 1991, pp. 253–284.

[2] K.R. Apt and E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer Verlag, 1991.

[3] J.W. de Bakker, *Mathematical Theory of Program Correctness*, Prentice–Hall, 1980.

[4] A. Baltag, *A logic for suspicious players: epistemic actions and belief updates in games*, Bulletin of Economic Research 54 (1), 2002, pp. 1–45.

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

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

[7] 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.

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

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

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

[11] H.W. de Haan, W.H. Hesselink, G.R. Renardel de Lavalette, *Knowledge-based programming inspired by an asynchronous hardware leader election problem* In: B. Dunin-Kęplicz, R. Verbrugge (eds.): FAMAS'03, ETAPS 2003, Warsaw, Poland. pp. 117–132.

[12] 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.

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

[14] W.H. Hesselink, *Programs, Recursion and Unbounded Choice, Predicate Transformation Semantics and Transformation Rules*, Cambridge University Press, 1992, (Cambridge Tracts in Theoretical Computer Science 27).

[15] 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.

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

[17] 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.

[18] R. van der Meyden, *Common knowledge and update in finite environments*, Information and Computation 140 (2), 1998, pp. 115–157.

[19] 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.

[20] 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.

[21] F. Stulp and R. Verbrugge, *A knowledge-based algorithm for the Internet protocol TCP*, Bulletin of Economic Research 54 (1), 2002, pp. 69–94.

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

[23] G. Winskel, *The Formal Semantics of Programming Languages: an Introduction*, The MIT Press, 1993.

A. Appendix

A.1. Message Passing

In this section we work out the example from Section 5.5. Recall that there are two agents 1 and 2, and three integer variable p, q and r which are all initially 0. Variable q is shared, p is private to agent 1 and r is private to agent 2. So the state space consists of triples (p, q, r), and we have

$$(p, q, r) \sim_1 (p', q', r') \Leftrightarrow p = p' \wedge q = q',$$
$$(p, q, r) \sim_2 (p', q', r') \Leftrightarrow q = q' \wedge r = r'.$$

The program is

$$\begin{array}{ll} Pg = & (\quad \Lambda \\ & \cup\, M_1\,(p \le r)\,?\,;\; p := p + 1 \\ & \cup \qquad (q < p)\,?\,;\; q := q + 1 \\ & \cup\; K_2\,(r < p)\,?\,;\; r := r + 1 \\ &)^* \,. \end{array}$$

Agent 2 can infer $r < p$ from $r < q$, since she can read q and agent 1 preserves the invariant $q \leq p$. Operationally speaking, agent 1 uses variable q as a message to agent 2 that p has been incremented.

In $B_0 = \mathcal{G}^+$, agent 1 considers it possible that $p \leq r$, but agent 2 has no knowledge whether $r < p$. Therefore the traces in $B_1 = F(B_0)$ are the traces generated by the knowledge-free program $P1$

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

For the next iterations it is useful to only consider traces that can possibly be generated by program *Pg*. Therefore, while determining the set of traces that satisfy $K_2(r < p)$ 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

$$(\ \Lambda\ \cup\ p := p+1\ \cup\ q := q+1\ \cup\ r := r+1\ )^*.$$

For $B = B_1$, we have that $[\![M_1(p \leq r)]\!]_B$ consists of traces $p \leq 0$, since r remains 0 in B_1. On the other hand, $B_\flat \cap [\![K_2(r < p)]\!]_B$ 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

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

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 $[\![M_1(p \leq r)]\!]_B$ consists of traces with $p \leq q\ \wedge\ p \leq 1$, since $r \leq 1$ holds in B_1. On the other hand, $B_\flat \cap [\![K_2(r < p)]\!]_B$ 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 2 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

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

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 $[\![M_1(p \leq r)]\!]_B$ 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, $B_\flat \cap [\![K_2(r < p)]\!]_B$ consists of traces of $B_\flat$ that end with $r < q \vee\ 3 \leq q$. No trace satisfying $3 \leq q$ is indistinguishable for 2 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

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

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 $[\![M_1(p \leq r)]\!]_B$ 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, $B_\flat \cap [\![K_2(r < p)]\!]_B$ consists of those traces of $B_\flat$ 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

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

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

For $B = B_5$, we have that $[\![M_1(p \leq r)]\!]_B$ consists of the traces with $p \leq q$ since r is only bounded by q in B. On the other hand, $B_\flat \cap [\![K_2(r < p)]\!]_B$ consists of those traces of $B_\flat$ 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

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

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 $[\![M_1(p \leq r)]\!]_B$ consists of traces with $p \leq q$ since r is only bounded by q in B. On the other hand, $B_\flat \cap [\![K_2(r < p)]\!]_B$ consists of those traces of $B_\flat$ 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 in Section 5.5 and is chosen as the semantics defined in (11). The iteration sequence is shown in figure 6.