Verified Synthesis of Knowledge-Based Programs in Finite Synchronous Environments

Peter Gammie

March 17, 2025

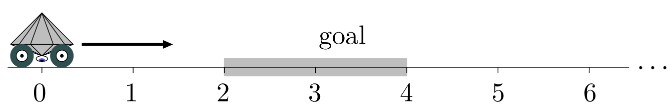
Abstract

Knowledge-based programs (KBPs) are a formalism for directly relating an agent's knowledge and behaviour. Here we present a general scheme for compiling KBPs to executable automata with a proof of correctness in Isabelle/HOL. We develop the algorithm top-down, using Isabelle's locale mechanism to structure these proofs, and show that two classic examples can be synthesised using Isabelle's code generator.

Contents

1 Introduction

Imagine a robot stranded at zero on a discrete number line, hoping to reach and remain in the goal region $\{2, 3, 4\}$. The environment helpfully pushes the robot to the right, zero or one steps per unit time, and the robot can sense the current position with an error of plus or minus one. If the only action the robot can take is to halt at its current position, what program should it execute?



An intuitive way to specify the robot's behaviour is with this *knowledge-based program* (KBP), using the syntax of Dijkstra's guarded commands:

do
[] $\mathbf{K}_{\text{robot}}$ goal → Halt
[] $\neg\mathbf{K}_{\text{robot}}$ goal → Nothing
od

Here "$\mathbf{K}_{\text{robot}}$ goal" intuitively denotes "the robot knows it is in the goal region" (Fagin et al. 1995, Example 7.2.2). We will make this precise in §3, but for now note that what the robot knows depends on the rest of the scenario, which in general may involve other agents also running KBPs. In a sense a KBP is a very literal rendition of a venerable artificial intelligence trope, that what an agent does should depend on its knowledge, and what an agent knows depends on what it does. It has been argued elsewhere Bickford et al. (2004); Engelhardt et al. (2000); Fagin et al. (1995) that this is a useful level of abstraction at which to reason about distributed systems, and some kinds of multi-agent systems Shoham and Leyton-Brown (2008). The cost is that these specifications are not directly executable, and it may take significant effort to find a concrete program that has the required behaviour.

The robot does have a simple implementation however: it should halt iff the sensor reads at least 3. That this is correct can be shown by an epistemic model checker such as MCK Gammie and van der Meyden (2004) or pencil-and-paper refinement Engelhardt et al. (2000). In contrast the goal of this work is to algorithmically discover such implementations, which is a step towards making the work of van der Meyden van der Meyden (1996) practical.

The contributions of this work are as follows: §2 develops enough of the theory of KBPs in Isabelle/HOL Nipkow et al. (2002) to support a formal proof of the possibility of their implementation by finite-state automata (§6). The later sections extend this development with a full top-down derivation of an original algorithm that constructs these implementations (§6.9) and two instances of it (§7.3 and §**??**), culminating in the mechanical synthesis of two standard examples from the literature: the aforementioned robot (§**??**) and the muddy children (§**??**).

We make judicious use of parametric polymorphism and Isabelle's locale mechanism Ballarin (2006) to establish and instantiate this theory in a top-down style. Isabelle's code generator Haftmann and Nipkow (2010) allows the algorithm developed here to be directly executed on the two examples, showing that the theory is both sound and usable. The complete development, available from the Archive of Formal Proofs Gammie (2011), includes the full formal details of all claims made in this paper.

In the following we adopt the Isabelle convention of using an apostrophe to prefix fixed but unknown types, such as *'a*, and postfix type constructors as in *'a* list. Other non-standard syntax will be explained as it arises.

2 A modal logic of knowledge

We begin with the standard syntax and semantics of the propositional logic of knowledge based on *Kripke structures*. More extensive treatments can be found in Lenzen (1978), Chellas (1980), Hintikka (1962) and Fagin et al. (1995, Chapter 2).

The syntax includes one knowledge modality per agent, and one for *common knowledge* amongst a set of agents. It is parameterised by the type *'a* of agents and *'p* of propositions.

datatype (*'a*, *'p*) *Kform*
  = *Kprop 'p*
  | *Knot* (*'a*, *'p*) *Kform*
  | *Kand* (*'a*, *'p*) *Kform* (*'a*, *'p*) *Kform*
  | *Kknows 'a* (*'a*, *'p*) *Kform* (‹**K**_ _›)
  | *Kcknows 'a list* (*'a*, *'p*) *Kform* (‹**C**_ _›)

A Kripke structure consists of a set of *worlds* of type *'w*, one *accessibility relation* between worlds for each agent and a *valuation function* that indicates the truth of a proposition at a world. This is a very general story that we will quickly specialise.

type-synonym *'w Relation* = (*'w* × *'w*) *set*

record (*'a*, *'p*, *'w*) *KripkeStructure* =
  worlds :: *'w set*

$relations :: 'a \Rightarrow 'w\ Relation$
$valuation :: 'w \Rightarrow 'p \Rightarrow bool$

definition $kripke :: ('a, 'p, 'w)\ KripkeStructure \Rightarrow bool$ **where**
$kripke\ M \equiv \forall a.\ relations\ M\ a \subseteq worlds\ M \times worlds\ M$

definition
$mkKripke :: 'w\ set \Rightarrow ('a \Rightarrow 'w\ Relation) \Rightarrow ('w \Rightarrow 'p \Rightarrow bool)$
$\Rightarrow ('a, 'p, 'w)\ KripkeStructure$
where
$mkKripke\ ws\ rels\ val \equiv$
$(\!|\ worlds = ws,\ relations = \lambda a.\ rels\ a \cap ws \times ws,\ valuation = val\ |\!)$⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

The standard semantics for knowledge is given by taking the accessibility relations to be equivalence relations, yielding the $S5_n$ structures, so-called due to their axiomatisation.

definition $S5n :: ('a, 'p, 'w)\ KripkeStructure \Rightarrow bool$ **where**
$S5n\ M \equiv \forall a.\ equiv\ (worlds\ M)\ (relations\ M\ a)$⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

Intuitively an agent considers two worlds to be equivalent if it cannot distinguish between them.

2.1 Satisfaction

A formula ϕ is satisfied at a world w in Kripke structure M in the following way:

fun $models :: ('a, 'p, 'w)\ KripkeStructure \Rightarrow 'w \Rightarrow ('a, 'p)\ Kform$
$\Rightarrow bool$ (‹(-, - ⊨ -)› [80,0,80] 80) **where**
$M, w \models (Kprop\ p) = valuation\ M\ w\ p$
$|\ M, w \models (Knot\ \varphi) = (\neg\ M, w \models \varphi)$
$|\ M, w \models (Kand\ \varphi\ \psi) = (M, w \models \varphi \wedge M, w \models \psi)$
$|\ M, w \models (\mathbf{K}_a\ \varphi) = (\forall w' \in relations\ M\ a\ \text{``}\ \{w\}.\ M, w' \models \varphi)$
$|\ M, w \models (\mathbf{C}_{as}\ \varphi) = (\forall w' \in (\bigcup a \in set\ as.\ relations\ M\ a)^+\ \text{``}\ \{w\}.\ M, w' \models \varphi)$

The first three clauses are standard.

The clause for $\mathbf{K}_a\ \varphi$ expresses the idea that an agent knows φ at world w in structure M iff φ is true at all worlds it considers possible.

The clause for $\mathbf{C}_{as}\ \varphi$ captures what it means for the set of agents as to commonly know φ; roughly, everyone knows φ and knows that everyone knows it, and so forth. Note that the transitive closure and the reflexive-transitive closure generate the same relation due to the reflexivity of the agents' accessibility relations; we use the former as it has a more pleasant induction principle.

⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

The relation between knowledge and common knowledge can be understood as follows, following Fagin et al. (1995, §2.4). Firstly, that ϕ is common knowledge to a set of agents as can be seen as asserting that everyone in as knows ϕ and moreover knows that it is common knowledge amongst as.

lemma *S5n-common-knowledge-fixed-point*:
assumes *S5n*: *S5n M*
assumes *w*: $w \in$ *worlds M*
assumes *a*: $a \in$ *set as*
shows *M*, $w \models$ *Kcknows as* φ
$\longleftrightarrow$ *M*, $w \models$ *Kand* (*Kknows a* φ) (*Kknows a* (*Kcknows as* φ))⟨*proof*⟩

Secondly we can provide an induction schema for the introduction of common knowledge: from everyone in *as* knows that ϕ implies $\phi \wedge \psi$, and that ϕ is satisfied at world w, infer that ψ is common knowledge amongst *as* at w.

lemma *S5n-common-knowledge-induct*:
assumes *S5n*: *S5n M*
assumes *w*: $w \in$ *worlds M*
assumes *E*: $\forall a \in$ *set as*. $\forall w \in$ *worlds M*.
M, $w \models \varphi \longrightarrow$ *M*, $w \models \mathbf{K}_a$ (*Kand* φ ψ)
assumes *p*: *M*, $w \models \varphi$
shows *M*, $w \models \mathbf{C}_{as}$ ψ⟨*proof*⟩⟨*proof*⟩

2.2 Generated models

The rest of this section introduces the technical machinery we use to relate Kripke structures.

Intuitively the truth of a formula at a world depends only on the worlds that are reachable from it in zero or more steps, using any of the accessibility relations at each step. Traditionally this result is called the *generated model property* (Chellas 1980, §3.4).

Given the model generated by w in M:

definition
gen-model :: ('a, 'p, 'w) *KripkeStructure* ⇒ 'w ⇒ ('a, 'p, 'w) *KripkeStructure*
where
gen-model M w ≡
let ws' = *worlds M* ∩ (⋃*a*. *relations M a*)* `` {*w*}
in ⦇ *worlds* = *ws'*,
relations = λ*a*. *relations M a* ∩ (*ws'* × *ws'*),
valuation = *valuation M* ⦈⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

where we take the image of w under the reflexive transitive closure of the agents' relations, we can show that the satisfaction of a formula φ at a world w' is preserved, provided w' is relevant to the world w that the sub-model is based upon:

lemma *gen-model-semantic-equivalence*:
assumes *M*: *kripke M*
assumes *w'*: $w' \in$ *worlds* (*gen-model M w*)
shows *M*, $w' \models \varphi \longleftrightarrow$ (*gen-model M w*), $w' \models \varphi$⟨*proof*⟩

This is shown by a straightforward structural induction over the formula φ.

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

2.3 Simulations

A *simulation*, or *p-morphism*, is a mapping from the worlds of one Kripke structure to another that preserves the truth of all formulas at related worlds (Chellas 1980, §3.4, Ex. 3.60). Such a function f must satisfy four properties. Firstly, the image of the set of worlds of M under f should equal the set of worlds of M'.

definition
$sim\text{-}range :: ('a, 'p, 'w1)\ KripkeStructure$
$\quad \Rightarrow ('a, 'p, 'w2)\ KripkeStructure \Rightarrow ('w1 \Rightarrow 'w2) \Rightarrow bool$
where
$sim\text{-}range\ M\ M'\ f \equiv worlds\ M' = f\ `\ worlds\ M$
$\quad \wedge (\forall a.\ relations\ M'\ a \subseteq worlds\ M' \times worlds\ M')$

The value of a proposition should be the same at corresponding worlds:

definition
$sim\text{-}val :: ('a, 'p, 'w1)\ KripkeStructure$
$\quad \Rightarrow ('a, 'p, 'w2)\ KripkeStructure \Rightarrow ('w1 \Rightarrow 'w2) \Rightarrow bool$
where
$sim\text{-}val\ M\ M'\ f \equiv \forall u \in worlds\ M.\ valuation\ M\ u = valuation\ M'\ (f\ u)$

If two worlds are related in M, then the simulation maps them to related worlds in M'; intuitively the simulation relates enough worlds. We term this the *forward* property.

definition
$sim\text{-}f :: ('a, 'p, 'w1)\ KripkeStructure$
$\quad \Rightarrow ('a, 'p, 'w2)\ KripkeStructure \Rightarrow ('w1 \Rightarrow 'w2) \Rightarrow bool$
where
$sim\text{-}f\ M\ M'\ f \equiv$
$\quad \forall a\ u\ v.\ (u, v) \in relations\ M\ a \longrightarrow (f\ u, f\ v) \in relations\ M'\ a$

Conversely, if two worlds $f\ u$ and v' are related in M', then there is a pair of related worlds u and v in M where $f\ v = v'$. Intuitively the simulation makes enough distinctions. We term this the *reverse* property.

definition
$sim\text{-}r :: ('a, 'p, 'w1)\ KripkeStructure$
$\quad \Rightarrow ('a, 'p, 'w2)\ KripkeStructure \Rightarrow ('w1 \Rightarrow 'w2) \Rightarrow bool$
where
$sim\text{-}r\ M\ M'\ f \equiv \forall a.\ \forall u \in worlds\ M.\ \forall v'.$
$\quad (f\ u, v') \in relations\ M'\ a$
$\quad \longrightarrow (\exists v.\ (u, v) \in relations\ M\ a \wedge f\ v = v')$

definition $sim\ M\ M'\ f \equiv sim\text{-}range\ M\ M'\ f \wedge sim\text{-}val\ M\ M'\ f$
$\quad \wedge sim\text{-}f\ M\ M'\ f \wedge sim\text{-}r\ M\ M'\ f \langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle$

Due to the common knowledge modality, we need to show the simulation properties lift through the transitive closure. In particular we can show that forward simulation is preserved:

lemma *sim-f-tc*:
  assumes *s*: *sim M M′ f*
  assumes *uv′*: $(u, v) \in (\bigcup a{\in}as.\ relations\ M\ a)^+$
  shows $(f\ u, f\ v) \in (\bigcup a{\in}as.\ relations\ M'\ a)^+$⟨*proof*⟩

Reverse simulation also:

lemma *sim-r-tc*:
  assumes *M*: *kripke M*
  assumes *s*: *sim M M′ f*
  assumes *u*: $u \in worlds\ M$
  assumes *fuv′*: $(f\ u, v') \in (\bigcup a{\in}as.\ relations\ M'\ a)^+$
 obtains *v* **where** $f\ v = v'$ **and** $(u, v) \in (\bigcup a{\in}as.\ relations\ M\ a)^+$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

Finally we establish the key property of simulations, that they preserve the satisfaction of all formulas in the following way:

lemma *sim-semantic-equivalence*:
  assumes *M*: *kripke M*
  assumes *s*: *sim M M′ f*
  assumes *u*: $u \in worlds\ M$
  shows $M, u \models \varphi \longleftrightarrow M', f\ u \models \varphi$⟨*proof*⟩

The proof is by structural induction over the formula φ. The knowledge cases appeal to our two simulation preservation lemmas.

Sangiorgi (2009) surveys the history of p-morphisms and the related concept of *bisimulation.*

This is all we need to know about Kripke structures.

3 Knowledge-based Programs

A knowledge-based programs (KBPs) encodes the dependency of action on knowledge by a sequence of guarded commands, and a *joint knowledge-based program* (JKBP) assigns a KBP to each agent:

record (′*a*, ′*p*, ′*aAct*) *GC* =
  guard :: (′*a*, ′*p*) *Kform*
  action :: ′*aAct*

type-synonym (′*a*, ′*p*, ′*aAct*) *KBP* = (′*a*, ′*p*, ′*aAct*) *GC list*
type-synonym (′*a*, ′*p*, ′*aAct*) *JKBP* = ′*a* ⇒ (′*a*, ′*p*, ′*aAct*) *KBP*

We use a list of guarded commands just so we can reuse this definition and others in algorithmic contexts; we would otherwise use a set as there is no problem with infinite programs or actions, and we always ignore the sequential structure.

Intuitively a KBP for an agent cannot directly evaluate the truth of an arbitrary formula as it may depend on propositions that the agent has no certainty about. For example, a card-playing agent cannot determine which cards are in the deck, despite being sure that those in her hand are not. Conversely agent *a* can evaluate formulas of the form $\mathbf{K}_a\ \varphi$ as these depend only on the worlds the agent thinks is possible.

Thus we restrict the guards of the JKBP to be boolean combinations of *subjective* formulas:

fun *subjective* :: $'a \Rightarrow ('a,\ 'p)$ *Kform* $\Rightarrow$ *bool* **where**
  subjective a (*Kprop p*) = *False*
| *subjective a* (*Knot f*) = *subjective a f*
| *subjective a* (*Kand f g*) = (*subjective a f* $\wedge$ *subjective a g*)
| *subjective a* (*Kknows a′ f*) = $(a = a')$
| *subjective a* (*Kcknows as f*) = ($a \in$ *set as*)

All JKBPs in the following sections are assumed to be subjective.

This syntactic restriction implies the desired semantic property, that we can evaluate a guard at an arbitrary world that is compatible with a given observation (Fagin et al. 1997, §3).

lemma *S5n-subjective-eq*:
  assumes *S5n*: *S5n M*
  assumes *subj*: *subjective a* φ
  assumes *ww′*: $(w,\ w') \in$ *relations M a*
  shows $M,\ w \models \varphi \longleftrightarrow M,\ w' \models \varphi$⟨*proof*⟩

The proof is by induction over the formula φ, using the properties of $S5_n$ Kripke structures in the knowledge cases.

We capture the fixed but arbitrary JKBP using a locale, and work in this context for the rest of this section.

locale *JKBP* =
  fixes *jkbp* :: $('a,\ 'p,\ 'aAct)$ *JKBP*
  assumes *subj*: $\forall$ *a gc. gc* $\in$ *set* (*jkbp a*) $\longrightarrow$ *subjective a* (*guard gc*)

context *JKBP*
begin

The action of the JKBP at a world is the list of all actions that are enabled at that world:

definition *jAction* :: $('a,\ 'p,\ 'w)$ *KripkeStructure* $\Rightarrow\ 'w \Rightarrow\ 'a \Rightarrow\ 'aAct$ *list*
where *jAction* $\equiv \lambda M\ w\ a.$ [*action gc. gc* $\leftarrow$ *jkbp a*, $M,\ w \models$ *guard gc*]

All of our machinery on Kripke structures lifts from the models relation of §2 through *jAction*, due to the subjectivity requirement. In particular, the KBP for agent a behaves the same at worlds that a cannot distinguish amongst:

lemma *S5n-jAction-eq*:
  assumes *S5n*: *S5n M*
  assumes *ww′*: $(w,\ w') \in$ *relations M a*
  shows *jAction M w a* = *jAction M w′ a*⟨*proof*⟩

Also the JKBP behaves the same on relevant generated models for all agents:

lemma *gen-model-jAction-eq*:
assumes *S*: *gen-model M w = gen-model M′ w*
assumes *w′*: *w′ ∈ worlds (gen-model M′ w)*
assumes *M*: *kripke M*
assumes *M′*: *kripke M′*
shows *jAction M w′ = jAction M′ w′*⟨*proof*⟩

Finally, *jAction* is invariant under simulations:

lemma *simulation-jAction-eq*:
assumes *M*: *kripke M*
assumes *sim*: *sim M M′ f*
assumes *w*: *w ∈ worlds M*
shows *jAction M w = jAction M′ (f w)*⟨*proof*⟩
end

4 Environments and Views

The previous section showed how a JKBP can be interpreted statically, with respect to a fixed Kripke structure. As we also wish to capture how agents interact, we adopt the *interpreted systems* and *contexts* of Fagin et al. (1995), which we term *environments* following van der Meyden (1996).

A *pre-environment* consists of the following:

- *envInit*, an arbitrary set of initial states;
- The protocol of the environment *envAction*, which depends on the current state;
- A transition function *envTrans*, which incorporates the environment's action and agents' behaviour into a state change; and
- A propositional evaluation function *envVal*.

We extend the *JKBP* locale with these constants:

locale *PreEnvironment = JKBP jkbp* **for** *jkbp* :: (′*a*, ′*p*, ′*aAct*) *JKBP*
\+ **fixes** *envInit* :: ′*s list*
and *envAction* :: ′*s* ⇒ ′*eAct list*
and *envTrans* :: ′*eAct* ⇒ (′*a* ⇒ ′*aAct*) ⇒ ′*s* ⇒ ′*s*
and *envVal* :: ′*s* ⇒ ′*p* ⇒ *bool*

We represent the possible evolutions of the system as finite sequences of states, represented by a left-recursive type ′*s Trace* with constructors *tInit s* and *t* ⇝ *s*, equipped with *tFirst*, *tLast*, *tLength* and *tMap* functions.

Constructing these traces requires us to determine the agents' actions at a given state. To do so we need to find an appropriate $S5_n$ structure for interpreting *jkbp*.

Given that we want the agents to make optimal use of the information they have access to, we allow these structures to depend on the entire history of the system, suitably conditioned by what the agents can observe. We capture this notion of observation with a *view* (van der Meyden 1996), which is an arbitrary function of a trace:

type-synonym $('s, 'tview)$ *View* $= 's$ *Trace* $\Rightarrow 'tview$
type-synonym $('a, 's, 'tview)$ *JointView* $= 'a \Rightarrow 's$ *Trace* $\Rightarrow 'tview$

We require views to be *synchronous*, i.e. that agents be able to tell the time using their view by distinguishing two traces of different lengths. As we will see in the next section, this guarantees that the JKBP has an essentially unique implementation.

We extend the *PreEnvironment* locale with a view:

locale *PreEnvironmentJView* =
  PreEnvironment jkbp envInit envAction envTrans envVal
    for *jkbp* :: $('a, 'p, 'aAct)$ *JKBP*
    and *envInit* :: $'s$ *list*
    and *envAction* :: $'s \Rightarrow 'eAct$ *list*
    and *envTrans* :: $'eAct \Rightarrow ('a \Rightarrow 'aAct) \Rightarrow 's \Rightarrow 's$
    and *envVal* :: $'s \Rightarrow 'p \Rightarrow$ *bool*
\+ **fixes** *jview* :: $('a, 's, 'tview)$ *JointView*
  assumes *sync*: $\forall\, a\ t\ t'.\ jview\ a\ t = jview\ a\ t' \longrightarrow tLength\ t = tLength\ t'$

The two principle synchronous views are the clock view and the perfect-recall view which we discuss further in §7. We will later derive an agent's concrete view from an instantaneous observation of the global state in §6.1.

We build a Kripke structure from a set of traces by relating traces that yield the same view. To obtain an $S5_n$ structure we also need a way to evaluate propositions: we apply *envVal* to the final state of a trace:

definition (**in** *PreEnvironmentJView*)
  mkM :: $'s$ *Trace set* $\Rightarrow ('a, 'p, 's\ Trace)$ *KripkeStructure*
where
  $mkM\ T \equiv$
    $(\!|\ worlds = T,$
      $relations = \lambda a.\ \{\ (t, t')\ .\ \{t, t'\} \subseteq T \wedge jview\ a\ t = jview\ a\ t'\ \},$
      $valuation = envVal \circ tLast\ |\!)\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle$

This construction supplants the role of the *local states* of Fagin et al. (1995).

The following section shows how we can canonically interpret the JKBP with respect to this structure.

5 Canonical Structures

Our goal in this section is to find the canonical set of traces for a given JKBP in a particular environment. As we will see, this always exists with respect to synchronous views.

We inductively define an *interpretation* of a JKBP with respect to an arbitrary set of traces T by constructing a sequence of sets of traces of increasing length:

fun *jkbpTn* :: *nat* ⇒ *'s Trace set* ⇒ *'s Trace set* **where**
$jkbpT_0\ T = \{\ tInit\ s\ |s.\ s \in set\ envInit\ \}$
$|\ jkbpT_{Suc\ n}\ T = \{\ t \leadsto envTrans\ eact\ aact\ (tLast\ t)\ |t\ eact\ aact.$
$t \in jkbpT_n\ T \wedge eact \in set\ (envAction\ (tLast\ t))$
$\wedge\ (\forall\ a.\ aact\ a \in set\ (jAction\ (mkM\ T)\ t\ a))\ \}$

This model reflects the failure of any agent to provide an action as failure of the entire system. In general *envTrans* may incorporate a scheduler and communication failure models.

The union of this sequence gives us a closure property:

definition *jkbpT* :: *'s Trace set* ⇒ *'s Trace set* **where**
$jkbpT\ T \equiv \bigcup n.\ jkbpT_n\ T$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

We say that a set of traces T *represents* a JKBP if it is closed under *jkbpT*:

definition *represents* :: *'s Trace set* ⇒ *bool* **where**
$represents\ T \equiv jkbpT\ T = T$⟨*proof*⟩⟨*proof*⟩

This is the vicious cycle that we break using our assumption that the view is synchronous. The key property of such views is that the satisfaction of an epistemic formula is determined by the set of traces in the model that have the same length. Lifted to *jAction*, we have:

⟨*proof*⟩⟨*proof*⟩**lemma** *sync-jview-jAction-eq*:
assumes *traces*: $\{\ t \in T\ .\ tLength\ t = n\ \} = \{\ t \in T'\ .\ tLength\ t = n\ \}$
assumes *tT*: $t \in \{\ t \in T\ .\ tLength\ t = n\ \}$
shows $jAction\ (mkM\ T)\ t = jAction\ (mkM\ T')\ t$⟨*proof*⟩

This implies that for a synchronous view we can inductively define the *canonical traces* of a JKBP. These are the traces that a JKBP generates when it is interpreted with respect to those very same traces. We do this by constructing the sequence $jkbpC_n$ of *(canonical) temporal slices* similarly to $jkbpT_n$:

fun *jkbpCn* :: *nat* ⇒ *'s Trace set* **where**
$jkbpC_0 = \{\ tInit\ s\ |s.\ s \in set\ envInit\ \}$
$|\ jkbpC_{Suc\ n} = \{\ t \leadsto envTrans\ eact\ aact\ (tLast\ t)\ |t\ eact\ aact.$
$t \in jkbpC_n \wedge eact \in set\ (envAction\ (tLast\ t))$
$\wedge\ (\forall\ a.\ aact\ a \in set\ (jAction\ (mkM\ jkbpC_n)\ t\ a))\ \}$

abbreviation *MCn* :: *nat* ⇒ (*'a*, *'p*, *'s Trace*) *KripkeStructure* **where**
$MC_n \equiv mkM\ jkbpC_n$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

The canonical set of traces for a JKBP with respect to a joint view is the set of canonical traces of all lengths.

definition *jkbpC* :: *'s Trace set* **where**
$jkbpC \equiv \bigcup n.\ jkbpC_n$

abbreviation *MC* :: (*'a*, *'p*, *'s Trace*) *KripkeStructure* **where**
$MC \equiv mkM\ jkbpC$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

We can show that *jkbpC* represents the joint knowledge-based program *jkbp* with respect to *jview*:

lemma *jkbpC-jkbpCn-jAction-eq*:
  assumes *tCn*: $t \in jkbpC_n$
  shows *jAction MC t* = *jAction* MC_n *t*⟨*proof*⟩

lemma *jkbpTn-jkbpCn-represents*: $jkbpT_n$ *jkbpC* = $jkbpC_n$
  ⟨*proof*⟩

theorem *jkbpC-represents*: *represents jkbpC*⟨*proof*⟩

We can show uniqueness too, by a similar argument:

theorem *jkbpC-represents-uniquely*:
  assumes *repT*: *represents T*
  shows *T* = *jkbpC*⟨*proof*⟩
end

Thus, at least with synchronous views, we are justified in talking about *the* representation of a JKBP in a given environment. More generally these results are also valid for the more general notion of *provides witnesses* as shown by Fagin et al. (1995, Lemma 7.2.4) and Fagin et al. (1997): it requires only that if a subjective knowledge formula is false on a trace then there is a trace of the same length or less that bears witness to that effect. This is a useful generalisation in asynchronous settings.

The next section shows how we can construct canonical representations of JKBPs using automata.

6 Automata Synthesis

Our attention now shifts to showing how we can synthesise standard automata that *implement* a JKBP under certain conditions. We proceed by defining *incremental views* following van der Meyden (1996), which provide the interface between the system and these automata. The algorithm itself is presented in §6.9.

6.1 Incremental views

Intuitively an agent instantaneously observes the system state, and so must maintain her view of the system *incrementally*: her new view must be a function of her current view and some new observation. We allow this observation to be an arbitrary projection *envObs a* of the system state for each agent *a*:

locale *Environment* =
  PreEnvironment jkbp envInit envAction envTrans envVal
    for *jkbp* :: (*'a*, *'p*, *'aAct*) *JKBP*
    and *envInit* :: *'s list*

```
    and envAction :: 's ⇒ 'eAct list
    and envTrans :: 'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's
    and envVal :: 's ⇒ 'p ⇒ bool
+ fixes envObs :: 'a ⇒ 's ⇒ 'obs
```

An incremental view therefore consists of two functions with these types:

```
type-synonym ('a, 'obs, 'tv) InitialIncrJointView = 'a ⇒ 'obs ⇒ 'tv
type-synonym ('a, 'obs, 'tv) IncrJointView = 'a ⇒ 'obs ⇒ 'tv ⇒ 'tv
```

These functions are required to commute with their corresponding trace-based joint view in the obvious way:

```
locale IncrEnvironment =
  Environment jkbp envInit envAction envTrans envVal envObs
+ PreEnvironmentJView jkbp envInit envAction envTrans envVal jview
    for jkbp :: ('a, 'p, 'aAct) JKBP
    and envInit :: 's list
    and envAction :: 's ⇒ 'eAct list
    and envTrans :: 'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's
    and envVal :: 's ⇒ 'p ⇒ bool
    and jview :: ('a, 's, 'tv) JointView
    and envObs :: 'a ⇒ 's ⇒ 'obs
+ fixes jviewInit :: ('a, 'obs, 'tv) InitialIncrJointView
  fixes jviewIncr :: ('a, 'obs, 'tv) IncrJointView
  assumes jviewInit: ∀ a s. jviewInit a (envObs a s) = jview a (tInit s)
  assumes jviewIncr: ∀ a t s. jview a (t ⇝ s)
                          = jviewIncr a (envObs a s) (jview a t)
```

Armed with these definitions, the following sections show that there are automata that implement a JKBP in a given environment.

6.2 Automata

Our implementations of JKBPs take the form of deterministic Moore automata, where transitions are labelled by observation and states with the action to be performed. We will use the term *protocols* interchangeably with automata, following the KBP literature, and adopt *joint protocols* for the assignment of one such to each agent:

```
record ('obs, 'aAct, 'ps) Protocol =
  pInit :: 'obs ⇒ 'ps
  pTrans :: 'obs ⇒ 'ps ⇒ 'ps
  pAct :: 'ps ⇒ 'aAct list

type-synonym ('a, 'obs, 'aAct, 'ps) JointProtocol
    = 'a ⇒ ('obs, 'aAct, 'ps) Protocol

context IncrEnvironment
begin
```

To ease composition with the system we adopt the function *pInit* which maps the initial observation to an initial automaton state.

van der Meyden (1996) shows that even non-deterministic JKBPs can be implemented with deterministic transition functions; intuitively all relevant uncertainty the agent has about the system must be encoded into each automaton state, so there is no benefit to doing this non-deterministically. In contrast we model the non-deterministic choice of action by making *pAct* a relation.

Running a protocol on a trace is entirely standard, as is running a joint protocol, and determining their actions:

fun *runJP* :: ($'a$, $'obs$, $'aAct$, $'ps$) *JointProtocol*
  $\Rightarrow$ $'s$ *Trace* $\Rightarrow$ $'a$ $\Rightarrow$ $'ps$
where
  runJP jp (*tInit s*) *a* = *pInit* (*jp a*) (*envObs a s*)
| *runJP jp* ($t \leadsto s$) *a* = *pTrans* (*jp a*) (*envObs a s*) (*runJP jp t a*)

abbreviation *actJP* :: ($'a$, $'obs$, $'aAct$, $'ps$) *JointProtocol*
  $\Rightarrow$ $'s$ *Trace* $\Rightarrow$ $'a$ $\Rightarrow$ $'aAct$ *list* **where**
  actJP jp $\equiv$ λt *a*. *pAct* (*jp a*) (*runJP jp t a*)

Similarly to §5 we will reason about the set of traces generated by a joint protocol in a fixed environment:

inductive-set
  jpTraces :: ($'a$, $'obs$, $'aAct$, $'ps$) *JointProtocol* $\Rightarrow$ $'s$ *Trace set*
  for *jp* :: ($'a$, $'obs$, $'aAct$, $'ps$) *JointProtocol*
where
  $s \in$ *set envInit* $\Longrightarrow$ *tInit s* $\in$ *jpTraces jp*
| $[\![$ $t \in$ *jpTraces jp*; *eact* $\in$ *set* (*envAction* (*tLast t*));
  $\bigwedge a$. *aact a* $\in$ *set* (*actJP jp t a*); *s* = *envTrans eact aact* (*tLast t*) $]\!]$
  $\Longrightarrow$ $t \leadsto s \in$ *jpTraces jp*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩
end

6.3 The Implementation Relation

With this machinery in hand, we now relate automata with JKBPs. We say a joint protocol *jp implements* a JKBP when they perform the same actions on the canonical of traces. Note that the behaviour of *jp* on other traces is arbitrary.

context *IncrEnvironment*
begin

definition
  implements :: ($'a$, $'obs$, $'aAct$, $'ps$) *JointProtocol* $\Rightarrow$ *bool*
where
  implements jp $\equiv$ ($\forall t \in$ *jkbpC*. *set* $\circ$ *actJP jp t* = *set* $\circ$ *jAction MC t*)

Clearly there are environments where the canonical trace set *jkbpC* can be generated by actions that differ from those prescribed by the JKBP. We can show that the *implements* relation is a stronger requirement than the mere trace-inclusion required by the *represents* relation of §5.

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩**lemma** *implements-represents*:
  assumes *impl*: *implements jp*
  shows *represents* (*jpTraces jp*)⟨*proof*⟩⟨*proof*⟩

The proof is by a straightfoward induction over the lengths of traces generated by the joint protocol.

Our final piece of technical machinery allows us to refine automata definitions: we say that two joint protocols are *behaviourally equivalent* if the actions they propose coincide for each canonical trace. The implementation relation is preserved by this relation.

definition
  behaviourally-equiv :: ($'a$, $'obs$, $'aAct$, $'ps$) *JointProtocol*
      $\Rightarrow$ ($'a$, $'obs$, $'aAct$, $'ps'$) *JointProtocol*
      $\Rightarrow$ *bool*
where
  $behaviourally\text{-}equiv\ jp\ jp' \equiv \forall\, t \in jkbpC.\ set \circ actJP\ jp\ t = set \circ actJP\ jp'\ t$
⟨*proof*⟩
lemma *behaviourally-equiv-implements*:
  assumes *behaviourally-equiv jp jp′*
  shows *implements jp* $\longleftrightarrow$ *implements jp′*⟨*proof*⟩

end

6.4 Automata using Equivalence Classes

We now show that there is an implementation of every JKBP with respect to every incremental synchronous view. Intuitively the states of the automaton for agent a represent the equivalence classes of traces that a considers possible, and the transitions update these sets according to her KBP.

context *IncrEnvironment*
begin

definition
  mkAutoEC :: ($'a$, $'obs$, $'aAct$, $'s$ *Trace set*) *JointProtocol*
where
  $mkAutoEC \equiv \lambda a.$
  $(\!|\ pInit = \lambda obs.\ \{\ t \in jkbpC\ .\ jviewInit\ a\ obs = jview\ a\ t\ \},$
  $pTrans = \lambda obs\ ps.\ \{\ t\ |t\ t'.\ t \in jkbpC \wedge t' \in ps$
  $\wedge\ jview\ a\ t = jviewIncr\ a\ obs\ (jview\ a\ t')\ \},$
  $pAct = \lambda ps.\ jAction\ MC\ (SOME\ t.\ t \in ps)\ a\ |\!)$

The function *SOME* is Hilbert's indefinite description operator ε, used here to choose an arbitrary trace from the protocol state.

That this automaton maintains the correct equivalence class on a trace t follows from an easy induction over t.

lemma *mkAutoEC-ec*:
  assumes $t \in jkbpC$
  shows $runJP\ mkAutoEC\ t\ a = \{\ t' \in jkbpC\ .\ jview\ a\ t' = jview\ a\ t\ \}$⟨*proof*⟩

We can show that the construction yields an implementation by appealing to the previous lemma and showing that the *pAct* functions coincide.

lemma *mkAutoEC-implements*: *implements mkAutoEC*⟨*proof*⟩

This definition leans on the canonical trace set jkbpC, and is indeed effective: we can enumerate all canonical traces and are sure to find one that has the view we expect. Then it is sufficient to consider other traces of the same length due to synchrony. We would need to do this computation dynamically, as the automaton will (in general) have an infinite state space.

end

6.5 Simulations

Our goal now is to reduce the space required by the automaton constructed by *mkAutoEC* by *simulating* the equivalence classes (§2.3).

The following locale captures the framework of van der Meyden (1996):

locale *SimIncrEnvironment* =
  IncrEnvironment jkbp envInit envAction envTrans envVal jview envObs
    jviewInit jviewIncr
  for $jkbp :: ('a, 'p, 'aAct)\ JKBP$

  and $envInit :: 's\ list$
  and $envAction :: 's \Rightarrow 'eAct\ list$
  and $envTrans :: 'eAct \Rightarrow ('a \Rightarrow 'aAct) \Rightarrow 's \Rightarrow 's$
  and $envVal :: 's \Rightarrow 'p \Rightarrow bool$
  and $jview :: ('a, 's, 'tv)\ JointView$
  and $envObs :: 'a \Rightarrow 's \Rightarrow 'obs$
  and $jviewInit :: ('a, 'obs, 'tv)\ InitialIncrJointView$
  and $jviewIncr :: ('a, 'obs, 'tv)\ IncrJointView$
+ **fixes** $simf :: 's\ Trace \Rightarrow 'ss$
  fixes $simRels :: 'a \Rightarrow 'ss\ Relation$
  fixes $simVal :: 'ss \Rightarrow 'p \Rightarrow bool$
  assumes *simf*: *sim MC* (*mkKripke* (*simf* ‘ *jkbpC*) *simRels simVal*) *simf*

context *SimIncrEnvironment*
begin

Note that the back tick ‘ is Isabelle/HOL’s relational image operator. In context it says that *simf* must be a simulation from *jkbpC* to its image under *simf*.

Firstly we lift our familiar canonical trace sets and Kripke structures through the simulation.

abbreviation $jkbpCSn :: nat \Rightarrow 'ss\ set$ **where**
  $jkbpCS_n \equiv simf$ ‘ $jkbpC_n$

abbreviation *jkbpCS* :: *'ss set* **where**
jkbpCS ≡ *simf* ‘ *jkbpC*

abbreviation *MCSn* :: *nat* ⇒ (*'a*, *'p*, *'ss*) *KripkeStructure* **where**
MCS_n ≡ *mkKripke* $jkbpCS_n$ *simRels simVal*

abbreviation *MCS* :: (*'a*, *'p*, *'ss*) *KripkeStructure* **where**
MCS ≡ *mkKripke jkbpCS simRels simVal*⟨*proof*⟩

We will be often be concerned with the equivalence class of traces generated by agent *a*'s view:

abbreviation *sim-equiv-class* :: *'a* ⇒ *'s Trace* ⇒ *'ss set* **where**
sim-equiv-class a t ≡ *simf* ‘ { *t'* ∈ *jkbpC* . *jview a t'* = *jview a t* }

abbreviation *jkbpSEC* :: *'ss set set* **where**
jkbpSEC ≡ ⋃ *a*. *sim-equiv-class a* ‘ *jkbpC*

With some effort we can show that the temporal slice of the simulated structure is adequate for determining the actions of the JKBP. The proof is tedious and routine, exploiting the sub-model property (§2.2).

⟨*proof*⟩
lemma *jkbpC-jkbpCSn-jAction-eq*:
assumes *tCn*: *t* ∈ *jkbpCn n*
shows *jAction MC t* = *jAction* (*MCSn n*) (*simf t*)⟨*proof*⟩
end

It can be shown that a suitable simulation into a finite structure is adequate to establish the existence of finite-state implementations (van der Meyden 1996, Theorem 2): essentially we apply the simulation to the states of *mkAutoEC*. However this result does not make it clear how the transition function can be incrementally constructed. One approach is to maintain *jkbpC* while extending the automaton, which is quite space inefficient.

Intuitively we would like to compute the possible *sim-equiv-class* successors of a given *sim-equiv-class* without reference to *jkbpC*, and this should be possible as the reachable simulated worlds must contain enough information to differentiate themselves from every other simulated world (reachable or not) that represents a trace that is observationally distinct to their own.

This leads us to asking for some extra functionality of our simulation, which we do in the following section.

6.6 Automata using simulations

The locale in Figure 1 captures our extra requirements of a simulation.

Firstly we relate the concrete representation *'rep* of equivalence classes under simulation to differ from the abstract representation *'ss set* using the abstraction

```
locale AlgSimIncrEnvironment =
  SimIncrEnvironment jkbp envInit envAction envTrans envVal
                  jview envObs jviewInit jviewIncr simf simRels simVal
    for jkbp :: ('a, 'p, 'aAct) JKBP
    and envInit :: 's list
    and envAction :: 's ⇒ 'eAct list
    and envTrans :: 'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's
    and envVal :: 's ⇒ 'p ⇒ bool

    and jview :: ('a, 's, 'tv) JointView
    and envObs :: 'a ⇒ 's ⇒ 'obs
    and jviewInit :: ('a, 'obs, 'tv) InitialIncrJointView
    and jviewIncr :: ('a, 'obs, 'tv) IncrJointView

    and simf :: 's Trace ⇒ 'ss
    and simRels :: 'a ⇒ 'ss Relation
    and simVal :: 'ss ⇒ 'p ⇒ bool

+ fixes simAbs :: 'rep ⇒ 'ss set

    and simObs :: 'a ⇒ 'rep ⇒ 'obs
    and simInit :: 'a ⇒ 'obs ⇒ 'rep
    and simTrans :: 'a ⇒ 'rep ⇒ 'rep list
    and simAction :: 'a ⇒ 'rep ⇒ 'aAct list

  assumes simInit:
          ∀ a iobs. iobs ∈ envObs a ` set envInit
              ⟶ simAbs (simInit a iobs)
                = simf ` { t' ∈ jkbpC. jview a t' = jviewInit a iobs }
    and simObs:
          ∀ a ec t. t ∈ jkbpC ∧ simAbs ec = sim-equiv-class a t
              ⟶ simObs a ec = envObs a (tLast t)
    and simAction:
          ∀ a ec t. t ∈ jkbpC ∧ simAbs ec = sim-equiv-class a t
              ⟶ set (simAction a ec) = set (jAction MC t a)
    and simTrans:
          ∀ a ec t. t ∈ jkbpC ∧ simAbs ec = sim-equiv-class a t
              ⟶ simAbs ` set (simTrans a ec)
                = { sim-equiv-class a (t' ↝ s)
                   |t' s. t' ↝ s ∈ jkbpC ∧ jview a t' = jview a t }
```

Figure 1: The *SimEnvironment* locale extends the *Environment* locale with simulation and algorithmic operations. The backtick ` is Isabelle/HOL's image-of-a-set-under-a-function operator.

function *simAbs* (de Roever and Engelhardt 1998); there is no one-size-fits-all concrete representation, as we will see.

Secondly we ask for a function *simInit a iobs* that faithfully generates a representation of the equivalence class of simulated initial states that are possible for agent *a* given the valid initial observation *iobs*.

Thirdly the *simObs* function allows us to partition the results of *simTrans* according to the recurrent observation that agent *a* makes of the equivalence class.

Fourthly, the function *simAction* computes a list of actions enabled by the JKBP on a state that concretely represents a canonical equivalence class.

Finally we expect to compute the list of represented *sim-equiv-class* successors of a given *sim-equiv-class* using *simTrans*.

Note that these definitions are stated relative to the environment and the JKBP, allowing us to treat specialised cases such as broadcast (§7.4 and §7.5).

With these functions in hand, we can define our desired automaton:

definition (**in** *AlgSimIncrEnvironment*)
  mkAutoSim :: (*'a*, *'obs*, *'aAct*, *'rep*) *JointProtocol*
where
  mkAutoSim ≡ λ*a*.
    ⦇ *pInit* = *simInit a*,
      pTrans = λ*obs ec*. (*SOME ec'*. *ec'* ∈ *set* (*simTrans a ec*)
                                  ∧ *simObs a ec'* = *obs*),
      pAct = *simAction a* ⦈⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

The automaton faithfully constructs the simulated equivalence class of the given trace:

lemma (**in** *AlgSimIncrEnvironment*) *mkAutoSim-ec*:
  assumes *tC*: *t* ∈ *jkbpC*
  shows *simAbs* (*runJP mkAutoSim t a*) = *sim-equiv-class a t*⟨*proof*⟩

This follows from a simple induction on *t*.

The following is a version of the Theorem 2 of van der Meyden (1996).

theorem (**in** *AlgSimIncrEnvironment*) *mkAutoSim-implements*:
  implements mkAutoSim⟨*proof*⟩

The reader may care to contrast these structures with the *progression structures* of van der Meyden (1997), where states contain entire Kripke structures, and expanding the automaton is alternated with bisimulation reduction to ensure termination when a finite-state implementation exists (see §**??**) We also use simulations in Appendix **??** to show the complexity of some related model checking problems.

We now review a simple *depth-first search* (DFS) theory, and an abstraction of finite maps, before presenting the algorithm for KBP synthesis.

```
locale DFS =
  fixes succs :: 'a ⇒ 'a list
  and isNode :: 'a ⇒ bool
  and invariant :: 'b ⇒ bool
  and ins :: 'a ⇒ 'b ⇒ 'b
  and memb :: 'a ⇒ 'b ⇒ bool
  and empt :: 'b
  and nodeAbs :: 'a ⇒ 'c
  assumes ins-eq: ⋀x y S. ⟦ isNode x; isNode y; invariant S; ¬ memb y S ⟧
                   ⟹ memb x (ins y S)
                   ⟷ ((nodeAbs x = nodeAbs y) ∨ memb x S)
  and succs: ⋀x y. ⟦ isNode x; isNode y; nodeAbs x = nodeAbs y ⟧
                   ⟹ nodeAbs ‘ set (succs x) = nodeAbs ‘ set (succs y)
  and empt: ⋀x. isNode x ⟹ ¬ memb x empt
  and succs-isNode: ⋀x. isNode x ⟹ list-all isNode (succs x)
  and empt-invariant: invariant empt
  and ins-invariant: ⋀x S. ⟦ isNode x; invariant S; ¬ memb x S ⟧
                    ⟹ invariant (ins x S)
  and graph-finite: finite (nodeAbs ‘ { x . isNode x})
```

Figure 2: The *DFS* locale.

6.7 Generic DFS

We use a generic DFS to construct the transitions and action function of the implementation of the JKBP. This theory is largely due to Stefan Berghofer and Alex Krauss (Berghofer and Reiter 2009). All proofs are elided as the fine details of how we explore the state space are inessential to the synthesis algorithm.

The DFS itself is defined in the standard tail-recursive way:

```
partial-function (tailrec) gen-dfs where
  gen-dfs succs ins memb S wl = (case wl of
     [] ⇒ S
   | (x # xs) ⇒
       if memb x S then gen-dfs succs ins memb S xs
       else gen-dfs succs ins memb (ins x S) (succs x @ xs))⟨proof⟩
```

The proofs are carried out in the locale of Figure 2, which details our requirements on the parameters for the DFS to behave as one would expect. Intuitively we are traversing a graph defined by *succs* from some initial work list *wl*, constructing an object of type *'b* as we go. The function *ins* integrates the current node into this construction. The predicate *isNode* is invariant over the set of states reachable from the initial work list, and is respected by *empt* and *ins*. We can also supply an invariant for the constructed object (*invariant*). Inside the locale, *dfs* abbreviates *gen-dfs* partially applied to the fixed parameters.

To support our data refinement (§6.6) we also require that the representation of nodes be adequate via the abstraction function *nodeAbs*, which the transition relation *succs* and visited predicate *memb* must respect. To ensure termination it must be the case that there are only a finite number of states, though there might be an infinity of representations.

We characterise the DFS traversal using the reflexive transitive closure operator:

definition (**in** *DFS*) *reachable* :: *'a set* ⇒ *'a set* **where**
  reachable xs ≡ $\{(x, y).\ y \in set\ (succs\ x)\}^*$ `` *xs*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

We make use of two results about the traversal. Firstly, that some representation of each reachable node has been incorporated into the final construction:

theorem (**in** *DFS*) *reachable-imp-dfs*:
  assumes *y*: *isNode y*
    and *xs*: *list-all isNode xs*
    and *m*: $y \in reachable\ (set\ xs)$
  shows $\exists\, y'.\ nodeAbs\ y' = nodeAbs\ y \wedge memb\ y'\ (dfs\ empt\ xs)$⟨*proof*⟩⟨*proof*⟩

Secondly, that if an invariant holds on the initial object then it holds on the final one:

theorem (**in** *DFS*) *dfs-invariant*:
  assumes *invariant S*
  assumes *list-all isNode xs*
  shows *invariant* (*dfs S xs*)⟨*proof*⟩

6.8 Finite map operations

The algorithm represents an automaton as a pair of maps, which we capture abstractly with a record and a predicate:

record (*'m*, *'k*, *'e*) *MapOps* =
  empty :: *'m*
  lookup :: $'m \Rightarrow 'k \rightharpoonup 'e$
  update :: $'k \Rightarrow 'e \Rightarrow 'm \Rightarrow 'm$

definition
  MapOps :: $('k \Rightarrow 'kabs) \Rightarrow 'kabs\ set \Rightarrow ('m, 'k, 'e)\ MapOps \Rightarrow bool$
where
  MapOps α *d ops* ≡
    $(\forall k.\ \alpha\ k \in d \longrightarrow lookup\ ops\ (empty\ ops)\ k = None)$
  $\wedge\ (\forall e\ k\ k'\ M.\ \alpha\ k \in d \wedge \alpha\ k' \in d$
      $\longrightarrow lookup\ ops\ (update\ ops\ k\ e\ M)\ k'$
       $= (if\ \alpha\ k' = \alpha\ k\ then\ Some\ e\ else\ lookup\ ops\ M\ k'))$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

The function α abstracts concrete keys of type *'k*, and the parameter d specifies the valid abstract keys.

This approach has the advantage over a locale that we can pass records to functions, while for a locale we would need to pass the three functions separately (as in the DFS theory of §6.7).

We use the following function to test for membership in the domain of the map:

definition *isSome* :: *'a option* ⇒ *bool* **where**
isSome opt ≡ *case opt of None* ⇒ *False* | *Some* - ⇒ *True*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

6.9 An algorithm for automata synthesis

We now show how to construct the automaton defined by *mkAutoSim* (§6.6) using the DFS of §6.7.

From here on we assume that the environment consists of only a finite set of states:

locale *FiniteEnvironment* =
Environment jkbp envInit envAction envTrans envVal envObs
for *jkbp* :: (*'a*, *'p*, *'aAct*) *JKBP*
and *envInit* :: (*'s* :: *finite*) *list*
and *envAction* :: *'s* ⇒ *'eAct list*
and *envTrans* :: *'eAct* ⇒ (*'a* ⇒ *'aAct*) ⇒ *'s* ⇒ *'s*
and *envVal* :: *'s* ⇒ *'p* ⇒ *bool*
and *envObs* :: *'a* ⇒ *'s* ⇒ *'obs*

The *Algorithm* locale, shown in Figure 3, also extends the *AlgSimIncrEnvironment* locale with a pair of finite map operations: *aOps* is used to map automata states to lists of actions, and *tOps* handles simulated transitions. In both cases the maps are only required to work on the abstract domain of simulated canonical traces. Note also that the space of simulated equivalence classes of type *'ss* must be finite, but there is no restriction on the representation type *'rep*.

We develop the algorithm for a single, fixed agent, which requires us to define a new locale *AlgorithmForAgent* that extends *Algorithm* with an extra parameter designating the agent:

locale *AlgorithmForAgent* =
Algorithm jkbp envInit envAction envTrans envVal jview envObs
jviewInit jviewIncr
simf simRels simVal simAbs simObs simInit simTrans simAction
aOps tOps
— ...
\+ **fixes** *a* :: *'a*

6.9.1 DFS operations

We represent the automaton under construction using a record:

record (*'ma*, *'mt*) *AlgState* =
aActs :: *'ma*
aTrans :: *'mt*

context *AlgorithmForAgent*
begin

We instantiate the DFS theory with the following functions.

A node is an equivalence class of represented simulated traces.

```
locale Algorithm =
  FiniteEnvironment jkbp envInit envAction envTrans envVal envObs
+ AlgSimIncrEnvironment jkbp envInit envAction envTrans envVal jview envObs
            jviewInit jviewIncr
            simf simRels simVal simAbs simObs simInit simTrans simAction
  for jkbp :: ('a, 'p, 'aAct) JKBP
  and envInit :: ('s :: finite) list
  and envAction :: 's ⇒ 'eAct list
  and envTrans :: 'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's
  and envVal :: 's ⇒ 'p ⇒ bool
  and jview :: ('a, 's, 'tobs) JointView

  and envObs :: 'a ⇒ 's ⇒ 'obs
  and jviewInit :: ('a, 'obs, 'tobs) InitialIncrJointView
  and jviewIncr :: ('a, 'obs, 'tobs) IncrJointView

  and simf :: 's Trace ⇒ 'ss :: finite
  and simRels :: 'a ⇒ 'ss Relation
  and simVal :: 'ss ⇒ 'p ⇒ bool

  and simAbs :: 'rep ⇒ 'ss set

  and simObs :: 'a ⇒ 'rep ⇒ 'obs
  and simInit :: 'a ⇒ 'obs ⇒ 'rep
  and simTrans :: 'a ⇒ 'rep ⇒ 'rep list
  and simAction :: 'a ⇒ 'rep ⇒ 'aAct list

+ fixes aOps :: ('ma, 'rep, 'aAct list) MapOps
  and tOps :: ('mt, 'rep × 'obs, 'rep) MapOps

 assumes aOps: MapOps simAbs jkbpSEC aOps
   and tOps: MapOps (λk. (simAbs (fst k), snd k)) (jkbpSEC × UNIV) tOps
```

Figure 3: The *Algorithm* locale.

definition $k\text{-}isNode :: {'rep} \Rightarrow bool$ **where**
$k\text{-}isNode\ ec \equiv simAbs\ ec \in sim\text{-}equiv\text{-}class\ a\ `\ jkbpC$

The successors of a node are those produced by the simulated transition function.

abbreviation $k\text{-}succs :: {'rep} \Rightarrow {'rep}\ list$ **where**
$k\text{-}succs \equiv simTrans\ a$

The initial automaton has no transitions and no actions.

definition $k\text{-}empt :: ({'ma}, {'mt})\ AlgState$ **where**
$k\text{-}empt \equiv (\!|\ aActs = empty\ aOps,\ aTrans = empty\ tOps\ |\!)$

We use the domain of the action map to track the set of nodes the DFS has visited.

definition $k\text{-}memb :: {'rep} \Rightarrow ({'ma}, {'mt})\ AlgState \Rightarrow bool$ **where**
$k\text{-}memb\ s\ A \equiv isSome\ (lookup\ aOps\ (aActs\ A)\ s)$

We integrate a new equivalence class into the automaton by updating the action and transition maps:

definition $actsUpdate :: {'rep} \Rightarrow ({'ma}, {'mt})\ AlgState \Rightarrow {'ma}$ **where**
$actsUpdate\ ec\ A \equiv update\ aOps\ ec\ (simAction\ a\ ec)\ (aActs\ A)$

definition $transUpdate :: {'rep} \Rightarrow {'rep} \Rightarrow {'mt} \Rightarrow {'mt}$ **where**
$transUpdate\ ec\ ec'\ at \equiv update\ tOps\ (ec,\ simObs\ a\ ec')\ ec'\ at$

definition $k\text{-}ins :: {'rep} \Rightarrow ({'ma}, {'mt})\ AlgState \Rightarrow ({'ma}, {'mt})\ AlgState$ **where**
$k\text{-}ins\ ec\ A \equiv (\!|\ aActs = actsUpdate\ ec\ A,$
$\quad aTrans = foldr\ (transUpdate\ ec)\ (k\text{-}succs\ ec)\ (aTrans\ A)\ |\!)$

The required properties are straightforward to show.

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

6.9.2 Algorithm invariant

The invariant for the automata construction is straightforward, viz that at each step of the process the state represents an automaton that concords with *mkAutoSim* on the visited equivalence classes. We also need to know that the state has preserved the *MapOps* invariants.

definition $k\text{-}invariant :: ({'ma}, {'mt})\ AlgState \Rightarrow bool$ **where**
$k\text{-}invariant\ A \equiv$
$(\forall\, ec\ ec'.\ k\text{-}isNode\ ec \wedge k\text{-}isNode\ ec' \wedge simAbs\ ec' = simAbs\ ec$
$\quad \longrightarrow lookup\ aOps\ (aActs\ A)\ ec = lookup\ aOps\ (aActs\ A)\ ec')$
$\wedge\ (\forall\, ec\ ec'\ obs.\ k\text{-}isNode\ ec \wedge k\text{-}isNode\ ec' \wedge simAbs\ ec' = simAbs\ ec$
$\quad \longrightarrow lookup\ tOps\ (aTrans\ A)\ (ec,\ obs) = lookup\ tOps\ (aTrans\ A)\ (ec',\ obs))$
$\wedge\ (\forall\, ec.\ k\text{-}isNode\ ec \wedge k\text{-}memb\ ec\ A$
$\quad \longrightarrow (\exists\, acts.\ lookup\ aOps\ (aActs\ A)\ ec = Some\ acts$
$\qquad \wedge\ set\ acts = set\ (simAction\ a\ ec)))$

$\wedge$ ($\forall$ *ec obs. k-isNode ec* $\wedge$ *k-memb ec A*
$\wedge$ *obs* $\in$ *simObs a ‘ set* (*simTrans a ec*)
$\longrightarrow$ ($\exists$ *ec′. lookup tOps* (*aTrans A*) (*ec*, *obs*) = *Some ec′*
$\wedge$ *simAbs ec′* $\in$ *simAbs ‘ set* (*simTrans a ec*)
$\wedge$ *simObs a ec′* = *obs*))⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

Showing that the invariant holds of *k-empt* and is respected by *k-ins* is routine.

The initial frontier is the partition of the set of initial states under the initial observation function.

definition (**in** *Algorithm*) *k-frontier* :: *′a* $\Rightarrow$ *′rep list* **where**
k-frontier a $\equiv$ *map* (*simInit a* $\circ$ *envObs a*) *envInit*⟨*proof*⟩
end

We now instantiate the *DFS* locale with respect to the *AlgorithmForAgent* locale. The instantiated lemmas are given the mandatory prefix *KBPAlg* in the *AlgorithmForAgent* locale.

sublocale *AlgorithmForAgent*
< *KBPAlg*: *DFS k-succs k-isNode k-invariant k-ins k-memb k-empt simAbs*
⟨*proof*⟩ ⟨*proof*⟩**context** *AlgorithmForAgent*
begin

The final algorithm, with the constants inlined, is shown in Figure 4. The rest of this section shows its correctness.

Firstly it follows immediately from *dfs-invariant* that the invariant holds of the result of the DFS:

⟨*proof*⟩**lemma** *k-dfs-invariant*: *k-invariant k-dfs*⟨*proof*⟩

Secondly we can see that the set of reachable equivalence classes coincides with the partition of *jkbpC* under the simulation and representation functions:

lemma *k-reachable*:
simAbs ‘ KBPAlg.reachable (*set* (*k-frontier a*)) = *sim-equiv-class a ‘ jkbpC*⟨*proof*⟩

Left to right follows from an induction on the reflexive, transitive closure, and right to left by induction over canonical traces.

This result immediately yields the same result at the level of representations:

lemma *k-memb-rep*:
assumes *N*: *k-isNode rec*
shows *k-memb rec k-dfs*⟨*proof*⟩
end

This concludes our agent-specific reasoning; we now show that the algorithm works for all agents. The following command generalises all our lemmas in the *AlgorithmForAgent* to the *Algorithm* locale, giving them the mandatory prefix *KBP*:

sublocale *Algorithm*
< *KBP*: *AlgorithmForAgent*

definition

$$\begin{array}{l}
\mathit{alg\text{-}dfs} :: ('ma,\ 'rep,\ 'aAct\ list)\ \mathit{MapOps}\\
\quad \Rightarrow ('mt,\ 'rep \times 'obs,\ 'rep)\ \mathit{MapOps}\\
\quad \Rightarrow ('rep \Rightarrow 'obs)\\
\quad \Rightarrow ('rep \Rightarrow 'rep\ list)\\
\quad \Rightarrow ('rep \Rightarrow 'aAct\ list)\\
\quad \Rightarrow 'rep\ list\\
\quad \Rightarrow ('ma,\ 'mt)\ \mathit{AlgState}
\end{array}$$

where

$$\begin{array}{l}
\mathit{alg\text{-}dfs}\ aOps\ tOps\ simObs\ simTrans\ simAction \equiv\\
\quad \mathit{let}\ k\text{-}empt = (\!|\ aActs = empty\ aOps,\ aTrans = empty\ tOps\ |\!);\\
\qquad k\text{-}memb = (\lambda s\ A.\ isSome\ (lookup\ aOps\ (aActs\ A)\ s));\\
\qquad k\text{-}succs = simTrans;\\
\qquad actsUpdate = \lambda ec\ A.\ update\ aOps\ ec\ (simAction\ ec)\ (aActs\ A);\\
\qquad transUpdate = \lambda ec\ ec'\ at.\ update\ tOps\ (ec,\ simObs\ ec')\ ec'\ at;\\
\qquad k\text{-}ins = \lambda ec\ A.\ (\!|\ aActs = actsUpdate\ ec\ A,\\
\qquad\qquad\qquad aTrans = foldr\ (transUpdate\ ec)\ (k\text{-}succs\ ec)\ (aTrans\ A)\ |\!)\\
\quad \mathit{in}\ gen\text{-}dfs\ k\text{-}succs\ k\text{-}ins\ k\text{-}memb\ k\text{-}empt
\end{array}$$

definition

$$\begin{array}{l}
\mathit{mkAlgAuto} :: ('ma,\ 'rep,\ 'aAct\ list)\ \mathit{MapOps}\\
\quad \Rightarrow ('mt,\ 'rep \times 'obs,\ 'rep)\ \mathit{MapOps}\\
\quad \Rightarrow ('a \Rightarrow 'rep \Rightarrow 'obs)\\
\quad \Rightarrow ('a \Rightarrow 'obs \Rightarrow 'rep)\\
\quad \Rightarrow ('a \Rightarrow 'rep \Rightarrow 'rep\ list)\\
\quad \Rightarrow ('a \Rightarrow 'rep \Rightarrow 'aAct\ list)\\
\quad \Rightarrow ('a \Rightarrow 'rep\ list)\\
\quad \Rightarrow ('a,\ 'obs,\ 'aAct,\ 'rep)\ \mathit{JointProtocol}
\end{array}$$

where

$$\begin{array}{l}
\mathit{mkAlgAuto}\ aOps\ tOps\ simObs\ simInit\ simTrans\ simAction\ frontier \equiv \lambda a.\\
\quad \mathit{let}\ auto = alg\text{-}dfs\ aOps\ tOps\ (simObs\ a)\ (simTrans\ a)\ (simAction\ a)\\
\qquad\qquad (frontier\ a)\\
\quad \mathit{in}\ (\!|\ pInit = simInit\ a,\\
\qquad pTrans = \lambda obs\ ec.\ the\ (lookup\ tOps\ (aTrans\ auto)\ (ec,\ obs)),\\
\qquad pAct = \lambda ec.\ the\ (lookup\ aOps\ (aActs\ auto)\ ec)\ |\!)
\end{array}$$

Figure 4: The algorithm. The function *the* projects a value from the *'a option* type, diverging on *None*.

jkbp envInit envAction envTrans envVal jview envObs
jviewInit jviewIncr simf simRels simVal simAbs simObs
simInit simTrans simAction aOps tOps a **for** *a*⟨*proof*⟩
context *Algorithm*
begin

abbreviation
k-mkAlgAuto ≡
mkAlgAuto aOps tOps simObs simInit simTrans simAction k-frontier⟨*proof*⟩

Running the automata produced by the DFS on a canonical trace t yields some representation of the expected equivalence class:

lemma *k-mkAlgAuto-ec*:
assumes *tC*: $t \in jkbpC$
shows *simAbs* (*runJP k-mkAlgAuto t a*) = *sim-equiv-class a t*⟨*proof*⟩

This involves an induction over the canonical trace t.

That the DFS and *mkAutoSim* yield the same actions on canonical traces follows immediately from this result and the invariant:

lemma *k-mkAlgAuto-mkAutoSim-act-eq*:
assumes *tC*: $t \in jkbpC$
shows *set* ∘ *actJP k-mkAlgAuto t* = *set* ∘ *actJP mkAutoSim t*⟨*proof*⟩

Therefore these two constructions are behaviourally equivalent, and so the DFS generates an implementation of *jkbp* in the given environment:

theorem *k-mkAlgAuto-implements*: *implements k-mkAlgAuto*⟨*proof*⟩
end

Clearly the automata generated by this algorithm are large. We discuss this issue in §**??**.

7 Concrete views

Following van der Meyden (1996), we provide two concrete synchronous views that illustrate how the theory works. For each view we give a simulation and a representation that satisfy the requirements of the *Algorithm* locale in Figure 3.

7.1 The Clock View

The *clock view* records the current time and the observation for the most recent state:

definition (**in** *Environment*)
clock-jview :: (*'a*, *'s*, *nat* × *'obs*) *JointView*
where
clock-jview ≡ λ*a t*. (*tLength t*, *envObs a* (*tLast t*))⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

This is the least-information synchronous view, given the requirements of §4. We show that finite-state implementations exist for all environments with respect to this view as per van der Meyden (1996).

The corresponding incremental view simply increments the counter records the new observation.

definition (**in** *Environment*)
  $clock\text{-}jviewInit :: 'a \Rightarrow 'obs \Rightarrow nat \times 'obs$
where
  $clock\text{-}jviewInit \equiv \lambda a\ obs.\ (0,\ obs)$

definition (**in** *Environment*)
  $clock\text{-}jviewIncr :: 'a \Rightarrow 'obs \Rightarrow nat \times 'obs \Rightarrow nat \times 'obs$
where
  $clock\text{-}jviewIncr \equiv \lambda a\ obs'\ (l,\ obs).\ (l + 1,\ obs')$

It is straightforward to demonstrate the assumptions of the incremental environment locale (§6.1) with respect to an arbitrary environment.

sublocale *Environment*
  < *Clock*: *IncrEnvironment jkbp envInit envAction envTrans envVal*
    clock-jview envObs clock-jviewInit clock-jviewIncr⟨*proof*⟩

As we later show, satisfaction of a formula at a trace $t \in Clock.jkbpC_n$ is determined by the set of final states of traces in *Clock.jkbpCn*:

context *Environment*
begin

abbreviation *clock-commonAbs* :: *'s Trace* ⇒ *'s set* **where**
  $clock\text{-}commonAbs\ t \equiv tLast\ `\ Clock.jkbpCn\ (tLength\ t)$

Intuitively this set contains the states that the agents commonly consider possible at time n, which is sufficient for determining knowledge as the clock view ignores paths. Therefore we can simulate trace t by pairing this abstraction of t with its final state:

type-synonym (**in** −) $'s\ clock\text{-}simWorlds = 's\ set \times 's$

definition *clock-sim* :: *'s Trace* ⇒ *'s clock-simWorlds* **where**
  $clock\text{-}sim \equiv \lambda t.\ (clock\text{-}commonAbs\ t,\ tLast\ t)$

In the Kripke structure for our simulation, we relate worlds for a if the sets of commonly-held states coincide, and the observation of the final states of the traces is the same. Propositions are evaluated at the final state.

definition *clock-simRels* :: *'a* ⇒ *'s clock-simWorlds Relation* **where**
  $clock\text{-}simRels \equiv \lambda a.\ \{\ ((X,\ s),\ (X',\ s'))\ |X\ X'\ s\ s'.$
    $X = X' \wedge \{s,\ s'\} \subseteq X \wedge envObs\ a\ s = envObs\ a\ s'\ \}$

definition *clock-simVal* :: *'s clock-simWorlds* ⇒ *'p* ⇒ *bool* **where**
  $clock\text{-}simVal \equiv envVal \circ snd$

abbreviation *clock-simMC* :: ('a, 'p, 's *clock-simWorlds*) *KripkeStructure* **where**
clock-simMC ≡ *mkKripke* (*clock-sim* ‘ *Clock.jkbpC*) *clock-simRels clock-simVal*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

That this is in fact a simulation (§2.3) is entirely straightforward.

lemma *clock-sim*:
sim Clock.MC clock-simMC clock-sim⟨*proof*⟩
end

The *SimIncrEnvironment* of §6.5 only requires that we provide it an *Environment* and a simulation.

sublocale *Environment*
< *Clock*: *SimIncrEnvironment jkbp envInit envAction envTrans envVal*
clock-jview envObs clock-jviewInit clock-jviewIncr
clock-sim clock-simRels clock-simVal⟨*proof*⟩

We next consider algorithmic issues.

7.1.1 Representations

We now turn to the issue of how to represent equivalence classes of states. As these are used as map keys, it is easiest to represent them canonically. A simple approach is to use *ordered distinct lists* of type *'a odlist* for the sets and *tries* for the maps. Therefore we ask that environment states *'s* belong to the class *linorder* of linearly-ordered types, and moreover that the set *agents* be effectively presented. We introduce a new locale capturing these requirements:

locale *FiniteLinorderEnvironment* =
Environment jkbp envInit envAction envTrans envVal envObs
for *jkbp* :: ('a::{*finite*, *linorder*}, 'p, 'aAct) *JKBP*
and *envInit* :: ('s::{*finite*, *linorder*}) *list*
and *envAction* :: 's ⇒ 'eAct *list*
and *envTrans* :: 'eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's
and *envVal* :: 's ⇒ 'p ⇒ *bool*
and *envObs* :: 'a ⇒ 's ⇒ 'obs
\+ **fixes** *agents* :: 'a *odlist*
assumes *agents*: *ODList.toSet agents* = *UNIV*

context *FiniteLinorderEnvironment*
begin

For a fixed agent *a*, we can reduce the number of worlds in *clock-simMC* by taking its quotient with respect to the equivalence relation for *a*. In other words, we represent a simulated equivalence class by a pair of the set of all states reachable at a particular time, and the subset of these that *a* considers possible. The worlds in our representational Kripke structure are therefore a pair of ordered, distinct lists:

type-synonym (**in** −) 's *clock-simWorldsRep* = 's *odlist* × 's *odlist*

We can readily abstract a representation to a set of simulated equivalence classes:

definition (**in** −)
  clock-simAbs :: $'s$::*linorder clock-simWorldsRep* ⇒ $'s$ *clock-simWorlds set*
where
  clock-simAbs X ≡ { (*ODList.toSet* (*fst X*), *s*) |*s*. *s* ∈ *ODList.toSet* (*snd X*) }

Assuming X represents a simulated equivalence class for $t \in jkbpC$, *clock-simAbs* X decomposes into these two functions:

definition
  agent-abs :: $'a \Rightarrow 's$ *Trace* ⇒ $'s$ *set*
where
  agent-abs a t ≡
    { *tLast t′* |*t′*. *t′* ∈ *Clock.jkbpC* ∧ *clock-jview a t′* = *clock-jview a t*}

definition
  common-abs :: $'s$ *Trace* ⇒ $'s$ *set*
where
  common-abs t ≡ *tLast* ‘ *Clock.jkbpCn* (*tLength t*)⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

This representation is canonical on the domain of interest (though not in general):

lemma *clock-simAbs-inj-on*:
  inj-on clock-simAbs { *x* . *clock-simAbs x* ∈ *Clock.jkbpSEC* }⟨*proof*⟩

We could further compress this representation by labelling each element of the set of states reachable at time n with a bit to indicate whether the agent considers that state possible. Note, however, that the representation would be non-canonical: if (s, *True*) is in the representation, indicating that the agent considers s possible, then (s, *False*) may or may not be. The associated abstraction function is not injective and hence would obfuscate the following. Repairing this would entail introducing a new type, which would again complicate this development.

The following lemmas make use of this Kripke structure, constructed from the set of final states of a temporal slice X:

definition
  clock-repRels :: $'a \Rightarrow ('s \times 's)$ *set*
where
  clock-repRels ≡ λ*a*. { (*s*, *s′*). *envObs a s* = *envObs a s′* }

abbreviation
  clock-repMC :: $'s$ *set* ⇒ ($'a$, $'p$, $'s$) *KripkeStructure*
where
  clock-repMC ≡ λ*X*. *mkKripke X clock-repRels envVal*⟨*proof*⟩⟨*proof*⟩

We can show that this Kripke structure retains sufficient information from *clock-simMC* by showing simulation. This is eased by introducing an intermediary structure that focusses on a particular trace:

abbreviation
  clock-jkbpCSt :: *'b Trace* ⇒ *'s clock-simWorlds set*
where
  clock-jkbpCSt t ≡ *clock-sim* ‘ *Clock.jkbpCn* (*tLength t*)

abbreviation
  clock-simMCt :: *'b Trace* ⇒ (*'a*, *'p*, *'s clock-simWorlds*) *KripkeStructure*
where
  clock-simMCt t ≡ *mkKripke* (*clock-jkbpCSt t*) *clock-simRels clock-simVal*

definition *clock-repSim* :: *'s clock-simWorlds* ⇒ *'s* **where**
  clock-repSim ≡ *snd*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

lemma *clock-repSim*:
  assumes *tC*: *t* ∈ *Clock.jkbpC*
  shows *sim* (*clock-simMCt t*)
      ((*clock-repMC* ∘ *fst*) (*clock-sim t*))
      clock-repSim⟨*proof*⟩

The following sections show how we satisfy the remaining requirements of the *Algorithm* locale of Figure 3. Where the proof is routine, we simply present the lemma without proof or comment.

Due to a limitation in the code generator in the present version of Isabelle (2011), we need to define the equations we wish to execute outside of a locale; the syntax (*in* −) achieves this by making definitons at the theory top-level. We then define (but elide) locale-local abbreviations that supply the locale-bound variables to these definitions.

7.1.2 Initial states

The initial states of the automaton for an agent is simply *envInit* paired with the partition of *envInit* under the agent's observation.

definition (**in** −)
  clock-simInit :: (*'s::linorder*) *list* ⇒ (*'a* ⇒ *'s* ⇒ *'obs*)
      ⇒ *'a* ⇒ *'obs* ⇒ *'s clock-simWorldsRep*
where
  clock-simInit envInit envObs ≡ λ*a iobs*.
    let cec = *ODList.fromList envInit*
     in (*cec*, *ODList.filter* (λ*s*. *envObs a s* = *iobs*) *cec*)

lemma *clock-simInit*:
  assumes *iobs* ∈ *envObs a* ‘ *set envInit*
  shows *clock-simAbs* (*clock-simInit a iobs*)
    = *clock-sim* ‘ { *t'* ∈ *Clock.jkbpC*.
        clock-jview a t' = *clock-jviewInit a iobs* }⟨*proof*⟩

7.1.3 Simulated observations

Agent a will make the same observation at any of the worlds that it considers possible, so we choose the first one in the list:

definition (**in** $-$)
  clock-simObs :: $('a \Rightarrow ('s :: linorder) \Rightarrow 'obs)$
    $\Rightarrow 'a \Rightarrow 's$ *clock-simWorldsRep* $\Rightarrow 'obs$
where
  clock-simObs envObs $\equiv \lambda a.$ *envObs a* $\circ$ *ODList.hd* $\circ$ *snd*

lemma *clock-simObs*:
  assumes *tC*: $t \in$ *Clock.jkbpC*
    and *ec*: *clock-simAbs ec* = *Clock.sim-equiv-class a t*
  shows *clock-simObs a ec* = *envObs a* (*tLast t*)⟨*proof*⟩

7.1.4 Evaluation

We define our *eval* function in terms of *evalS*, which implements boolean logic over $'s$ *odlist* in the usual way – see §7.3.4 for the relevant clauses. It requires three functions specific to the representation: one each for propositions, knowledge and common knowledge.

Propositions define subsets of the worlds considered possible:

abbreviation (**in** $-$)
  clock-evalProp :: $(('s::linorder) \Rightarrow 'p \Rightarrow bool)$
    $\Rightarrow 's$ *odlist* $\Rightarrow 'p \Rightarrow 's$ *odlist*
where
  clock-evalProp envVal $\equiv \lambda X\ p.$ *ODList.filter* $(\lambda s.$ *envVal s p*$)$ X

The knowledge relation computes the subset of the commonly-held-possible worlds *cec* that agent a considers possible at world s:

definition (**in** $-$)
  clock-knowledge :: $('a \Rightarrow ('s :: linorder) \Rightarrow 'obs) \Rightarrow 's$ *odlist*
    $\Rightarrow 'a \Rightarrow 's \Rightarrow 's$ *odlist*
where
  clock-knowledge envObs cec $\equiv \lambda a\ s.$
    ODList.filter $(\lambda s'.$ *envObs a s* = *envObs a* $s')$ *cec*

Similarly the common knowledge operation computes the transitive closure of the union of the knowledge relations for the agents *as*:

definition (**in** $-$)
  clock-commonKnowledge :: $('a \Rightarrow ('s :: linorder) \Rightarrow 'obs) \Rightarrow 's$ *odlist*
    $\Rightarrow 'a$ *list* $\Rightarrow 's \Rightarrow 's$ *odlist*
where
  clock-commonKnowledge envObs cec $\equiv \lambda as\ s.$
    let $r = \lambda a.$ *ODList.fromList* [(s', s'') . $s' \leftarrow$ *toList cec*, $s'' \leftarrow$ *toList cec*,
      envObs a s' = *envObs a* s''];
      R = *toList* (*ODList.big-union r as*)

in ODList.fromList (*memo-list-trancl R s*)

The function *memo-list-trancl* comes from the executable transitive closure theory of (Sternagel and Thiemann 2011).

The evaluation function evaluates a subjective knowledge formula on the representation of an equivalence class:

definition (**in** −)
  eval :: ((*'s* :: *linorder*) ⇒ *'p* ⇒ *bool*)
     ⇒ (*'a* ⇒ *'s* ⇒ *'obs*)
     ⇒ *'s clock-simWorldsRep* ⇒ (*'a*, *'p*) *Kform* ⇒ *bool*
where
  eval envVal envObs ≡ λ(*cec*, *aec*). *evalS* (*clock-evalProp envVal*)
     (*clock-knowledge envObs cec*)
     (*clock-commonKnowledge envObs cec*)
     aec

This function corresponds with the standard semantics:

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩**lemma** *eval-models*:
  assumes *tC*: *t* ∈ *Clock.jkbpC*
    and *aec*: *ODList.toSet aec* = *agent-abs a t*
    and *cec*: *ODList.toSet cec* = *common-abs t*
    and *subj-phi*: *subjective a* φ
    and *s*: *s* ∈ *ODList.toSet aec*
  shows *eval envVal envObs* (*cec*, *aec*) φ
    ⟷ *clock-repMC* (*ODList.toSet cec*), *s* ⊨ φ⟨*proof*⟩

7.1.5 Simulated actions

From a common equivalence class and a subjective equivalence class for agent *a*, we can compute the actions enabled for *a*:

definition (**in** −)
  clock-simAction :: (*'a*, *'p*, *'aAct*) *JKBP* ⇒ ((*'s* :: *linorder*) ⇒ *'p* ⇒ *bool*)
     ⇒ (*'a* ⇒ *'s* ⇒ *'obs*)
     ⇒ *'a* ⇒ *'s clock-simWorldsRep* ⇒ *'aAct list*
where
  clock-simAction jkbp envVal envObs ≡ λ*a* (*Y*, *X*).
    [*action gc*. *gc* ← *jkbp a*, *eval envVal envObs* (*Y*, *X*) (*guard gc*)]

Using the above result about evaluation, we can relate *clock-simAction* to *jAction*. Firstly, *clock-simAction* behaves the same as *jAction* using the *clock-repMC* structure:

lemma *clock-simAction-jAction*:
  assumes *tC*: *t* ∈ *Clock.jkbpC*
    and *aec*: *ODList.toSet aec* = *agent-abs a t*
    and *cec*: *ODList.toSet cec* = *common-abs t*
  shows *set* (*clock-simAction a* (*cec*, *aec*))
    = *set* (*jAction* (*clock-repMC* (*ODList.toSet cec*)) (*tLast t*) *a*)⟨*proof*⟩⟨*proof*⟩

We can connect the agent's choice of actions on the *clock-repMC* structure to those on the *Clock.MC* structure using our earlier results about actions being preserved by generated models and simulations.

lemma *clock-simAction'*:
  assumes *tC*: $t \in$ *Clock.jkbpC*
  assumes *aec*: *ODList.toSet aec* = *agent-abs a t*
  assumes *cec*: *ODList.toSet cec* = *common-abs t*
  shows *set* (*clock-simAction a* (*cec*, *aec*)) = *set* (*jAction Clock.MC t a*)⟨*proof*⟩

The *Algorithm* locale requires a specialisation of this lemma:

lemma *clock-simAction*:
  assumes *tC*: $t \in$ *Clock.jkbpC*
  assumes *ec*: *clock-simAbs ec* = *Clock.sim-equiv-class a t*
  shows *set* (*clock-simAction a ec*) = *set* (*jAction Clock.MC t a*)⟨*proof*⟩

7.1.6 Simulated transitions

We need to determine the image of the set of commonly-held-possible states under the transition function, and also for the agent's subjective equivalence class. We do this with the *clock-trans* function:

definition (**in** −)
  clock-trans :: ('*a* :: *linorder*) *odlist* ⇒ ('*a*, '*p*, '*aAct*) *JKBP*
      ⇒ (('*s* :: *linorder*) ⇒ '*eAct list*)
      ⇒ ('*eAct* ⇒ ('*a* ⇒ '*aAct*) ⇒ '*s* ⇒ '*s*)
      ⇒ ('*s* ⇒ '*p* ⇒ *bool*) ⇒ ('*a* ⇒ '*s* ⇒ '*obs*)
      ⇒ '*s odlist* ⇒ '*s odlist* ⇒ '*s odlist*
where
  clock-trans agents jkbp envAction envTrans envVal envObs ≡ λ*cec X*.
    ODList.fromList (*concat*
      [[*envTrans eact aact s* .
            eact ← *envAction s*,
            aact ← *listToFuns* (λ*a*. *clock-simAction jkbp envVal envObs a*
                    (*cec*, *clock-knowledge envObs cec a s*))
                    (*toList agents*)] .
          s ← *toList X*])⟨*proof*⟩

The function *listToFuns* exhibits the isomorphism between ('*a* × '*b list*) *list* and ('*a* ⇒ '*b*) *list* for finite types '*a*.

We can show that the transition function works for both the commonly-held set of states and the agent subjective one. The proofs are straightforward.

lemma *clock-trans-common*:
  assumes *tC*: $t \in$ *Clock.jkbpC*
  assumes *ec*: *clock-simAbs ec* = *Clock.sim-equiv-class a t*
  shows *ODList.toSet* (*clock-trans* (*fst ec*) (*fst ec*))
    = { *s* |*t' s*. *t'* ⇝ *s* ∈ *Clock.jkbpC* ∧ *tLength t'* = *tLength t* }⟨*proof*⟩

lemma *clock-trans-agent*:

assumes *tC*: $t \in$ *Clock.jkbpC*
assumes *ec*: *clock-simAbs ec* = *Clock.sim-equiv-class a t*
shows *ODList.toSet* (*clock-trans* (*fst ec*) (*snd ec*))
= { s |t' s. $t' \rightsquigarrow s \in$ *Clock.jkbpC* $\wedge$ *clock-jview a t'* = *clock-jview a t* }⟨*proof*⟩

Note that the clock semantics disregards paths, so we simply compute the successors of the temporal slice and partition that. Similarly the successors of the agent's subjective equivalence class tell us what the set of possible observations are:

definition (**in** −)
clock-mkSuccs :: (*'s* :: *linorder* ⇒ *'obs*) ⇒ *'obs* ⇒ *'s odlist*
⇒ *'s clock-simWorldsRep*
where
clock-mkSuccs envObs obs Y' ≡ (*Y'*, *ODList.filter* (λ*s*. *envObs s* = *obs*) *Y'*)

Finally we can define our transition function on simulated states:

definition (**in** −)
clock-simTrans :: (*'a* :: *linorder*) *odlist* ⇒ (*'a*, *'p*, *'aAct*) *JKBP*
⇒ ((*'s* :: *linorder*) ⇒ *'eAct list*)
⇒ (*'eAct* ⇒ (*'a* ⇒ *'aAct*) ⇒ *'s* ⇒ *'s*)
⇒ (*'s* ⇒ *'p* ⇒ *bool*) ⇒ (*'a* ⇒ *'s* ⇒ *'obs*)
⇒ *'a* ⇒ *'s clock-simWorldsRep* ⇒ *'s clock-simWorldsRep list*
where
clock-simTrans agents jkbp envAction envTrans envVal envObs ≡ λ*a* (*Y*, *X*).
let X' = *clock-trans agents jkbp envAction envTrans envVal envObs Y X*;
Y' = *clock-trans agents jkbp envAction envTrans envVal envObs Y Y*
in [*clock-mkSuccs* (*envObs a*) *obs Y'* .
obs ← *map* (*envObs a*) (*toList X'*)]

Showing that this respects the property asked of it by the *Algorithm* locale is straightforward:

lemma *clock-simTrans*:
assumes *tC*: $t \in$ *Clock.jkbpC*
and *ec*: *clock-simAbs ec* = *Clock.sim-equiv-class a t*
shows *clock-simAbs* ' *set* (*clock-simTrans a ec*)
= { *Clock.sim-equiv-class a* ($t' \rightsquigarrow s$)
|t' s. $t' \rightsquigarrow s \in$ *Clock.jkbpC* $\wedge$ *clock-jview a t'* = *clock-jview a t* }⟨*proof*⟩
end

7.1.7 Maps

As mentioned above, the canonicity of our ordered, distinct list representation of automaton states allows us to use them as keys in a digital trie; a value of type (*'key*, *'val*) *trie* maps keys of type *'key list* to values of type *'val*.

In this specific case we track automaton transitions using a two-level structure mapping sets of states to an association list mapping observations to sets of states, and for actions automaton states map directly to agent actions.

type-synonym (*'s*, *'obs*) *clock-trans-trie*

= ('s, ('s, ('obs, 's clock-simWorldsRep) mapping) trie) trie
type-synonym ('s, 'aAct) clock-acts-trie = ('s, ('s, 'aAct) trie) trie⟨proof⟩⟨proof⟩

We define two records *acts-MapOps* and *trans-MapOps* satisfying the *MapOps* predicate (§6.8). Discharging the obligations in the *Algorithm* locale is routine, leaning on the work of Lammich and Lochbihler (2010).

7.1.8 Locale instantiation

Finally we assemble the algorithm and discharge the proof obligations.

sublocale *FiniteLinorderEnvironment*
< *Clock*: *Algorithm*
jkbp envInit envAction envTrans envVal
clock-jview envObs clock-jviewInit clock-jviewIncr
clock-sim clock-simRels clock-simVal
clock-simAbs clock-simObs clock-simInit clock-simTrans clock-simAction
acts-MapOps trans-MapOps⟨proof⟩

Explicitly, the algorithm for this case is:

definition
mkClockAuto ≡ λ*agents jkbp envInit envAction envTrans envVal envObs*.
mkAlgAuto acts-MapOps
trans-MapOps
(*clock-simObs envObs*)
(*clock-simInit envInit envObs*)
(*clock-simTrans agents jkbp envAction envTrans envVal envObs*)
(*clock-simAction jkbp envVal envObs*)
(λ*a*. *map* (*clock-simInit envInit envObs a* ∘ *envObs a*) *envInit*)

lemma (**in** *FiniteLinorderEnvironment*) *mkClockAuto-implements*:
Clock.implements
(*mkClockAuto agents jkbp envInit envAction envTrans envVal envObs*)⟨proof⟩⟨proof⟩

We discuss the clock semantics further in §**??**.

7.2 The Synchronous Perfect-Recall View

The synchronous perfect-recall (SPR) view records all observations the agent has made on a given trace. This is the canonical full-information synchronous view; all others are functions of this one.

Intuitively it maintains a list of all observations made on the trace, with the length of the list recording the time:

definition (**in** *Environment*) *spr-jview* :: ('a, 's, 'obs Trace) *JointView* **where**
spr-jview a = *tMap* (*envObs a*)⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

The corresponding incremental view appends a new observation to the existing ones:

definition (**in** *Environment*) *spr-jviewInit* :: $'a \Rightarrow 'obs \Rightarrow 'obs$ *Trace* **where**
spr-jviewInit $\equiv \lambda a\ obs.\ tInit\ obs$

definition (**in** *Environment*)
spr-jviewIncr :: $'a \Rightarrow 'obs \Rightarrow 'obs$ *Trace* $\Rightarrow 'obs$ *Trace*
where
spr-jviewIncr $\equiv \lambda a\ obs'\ tobs.\ tobs \rightsquigarrow obs'$

sublocale *Environment*
< *SPR*: *IncrEnvironment jkbp envInit envAction envTrans envVal*
spr-jview envObs spr-jviewInit spr-jviewIncr⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

van der Meyden (1996, Theorem 5) showed that it is not the case that finite-state implementations always exist with respect to the SPR view, and so we consider three special cases:

§7.3 where there is a single agent;

§7.4 when the protocols of the agents are deterministic and communication is by broadcast; and

§7.5 when the agents use non-deterministic protocols and again use broadcast to communicate.

Note that these cases do overlap but none is wholly contained in another.

7.3 Perfect Recall for a Single Agent

We capture our expectations of a single-agent scenario in the following locale:

locale *FiniteSingleAgentEnvironment* =
FiniteEnvironment jkbp envInit envAction envTrans envVal envObs
for *jkbp* :: $('a, 'p, 'aAct)$ *JKBP*
and *envInit* :: $('s :: \{finite, linorder\})$ *list*
and *envAction* :: $'s \Rightarrow 'eAct$ *list*
and *envTrans* :: $'eAct \Rightarrow ('a \Rightarrow 'aAct) \Rightarrow 's \Rightarrow 's$
and *envVal* :: $'s \Rightarrow 'p \Rightarrow bool$
and *envObs* :: $'a \Rightarrow 's \Rightarrow 'obs$
+ **fixes** *agent* :: $'a$
assumes *envSingleAgent*: $a = agent$

As per the clock semantics of §7.1, we assume that the set of states is finite and linearly ordered. We give the sole agent the name *agent*.

Our simulation is quite similar to the one for the clock semantics of §7.1: it records the set of worlds that the agent considers possible relative to a trace and the SPR view. The key difference is that it is path-sensitive:

context *FiniteSingleAgentEnvironment*
begin

definition *spr-abs* :: *'s Trace* ⇒ *'s set* **where**
spr-abs t ≡
tLast ‘ { t' ∈ *SPR.jkbpC* . *spr-jview agent* t' = *spr-jview agent t* }

type-synonym (**in** −) *'s spr-simWorlds* = *'s set* × *'s*

definition *spr-sim* :: *'s Trace* ⇒ *'s spr-simWorlds* **where**
spr-sim ≡ λ*t*. (*spr-abs t*, *tLast t*)⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

The Kripke structure for this simulation relates worlds for *agent* if the sets of states it considers possible coincide, and the observation of the final states of the trace is the same. Propositions are evaluated at the final state.

definition *spr-simRels* :: *'a* ⇒ *'s spr-simWorlds Relation* **where**
spr-simRels ≡ λ*a*. { ((*U*, *u*), (*V*, *v*)) |*U u V v*.
U = *V* ∧ {*u*, *v*} ⊆ *U* ∧ *envObs a u* = *envObs a v* }

definition *spr-simVal* :: *'s spr-simWorlds* ⇒ *'p* ⇒ *bool* **where**
spr-simVal ≡ *envVal* ∘ *snd*

abbreviation *spr-simMC* :: (*'a*, *'p*, *'s spr-simWorlds*) *KripkeStructure* **where**
spr-simMC ≡ *mkKripke* (*spr-sim* ‘ *SPR.jkbpC*) *spr-simRels spr-simVal*⟨*proof*⟩⟨*proof*⟩

Demonstrating that this is a simulation (§2.3) is straightforward.

lemma *spr-sim*: *sim SPR.MC spr-simMC spr-sim*⟨*proof*⟩

end

sublocale *FiniteSingleAgentEnvironment*
< *SPRsingle*: *SimIncrEnvironment jkbp envInit envAction envTrans envVal*
spr-jview envObs spr-jviewInit spr-jviewIncr
spr-sim spr-simRels spr-simVal⟨*proof*⟩

7.3.1 Representations

As in §7.1.1, we quotient *'s spr-simWorlds* by *spr-simRels*. Because there is only a single agent, an element of this quotient corresponding to a cononical trace *t* is isomorphic to the set of states that are possible given the sequence of observations made by *agent* on *t*. Therefore we have a very simple representation:

context *FiniteSingleAgentEnvironment*
begin

type-synonym (**in** −) *'s spr-simWorldsRep* = *'s odlist*

It is very easy to map these representations back to simulated equivalence classes:

definition
spr-simAbs :: *'s spr-simWorldsRep* ⇒ *'s spr-simWorlds set*
where

spr-simAbs ≡ *λss*. { (*toSet ss*, *s*) |*s*. *s* ∈ *toSet ss* }

This time our representation is unconditionally canonical:

lemma *spr-simAbs-inj*: *inj spr-simAbs*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

We again make use of the following Kripke structure, where the worlds are the final states of the subset of the temporal slice that *agent* believes possible:

definition *spr-repRels* :: *'a* ⇒ (*'s* × *'s*) *set* **where**
spr-repRels ≡ *λa*. { (*s*, *s'*). *envObs a s'* = *envObs a s* }

abbreviation *spr-repMC* :: *'s set* ⇒ (*'a*, *'p*, *'s*) *KripkeStructure* **where**
spr-repMC ≡ *λX*. *mkKripke X spr-repRels envVal*

Similarly we show that this Kripke structure is adequate by introducing an intermediate structure and connecting them all with a tower of simulations:

abbreviation *spr-jkbpCSt* :: *'s Trace* ⇒ *'s spr-simWorlds set* **where**
spr-jkbpCSt t ≡ *SPRsingle.sim-equiv-class agent t*

abbreviation
spr-simMCt :: *'s Trace* ⇒ (*'a*, *'p*, *'s spr-simWorlds*) *KripkeStructure*
where
spr-simMCt t ≡ *mkKripke* (*spr-jkbpCSt t*) *spr-simRels spr-simVal*

definition *spr-repSim* :: *'s spr-simWorlds* ⇒ *'s* **where**
spr-repSim ≡ *snd*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

lemma *spr-repSim*:
assumes *tC*: *t* ∈ *SPR.jkbpC*
shows *sim* (*spr-simMCt t*)
((*spr-repMC* ∘ *fst*) (*spr-sim t*))
spr-repSim⟨*proof*⟩

As before, the following sections discharge the requirements of the *Algorithm* locale of Figure 3.

7.3.2 Initial states

The initial states of the automaton for *agent* is simply the partition of *envInit* under *agent*'s observation.

definition (**in** −)
spr-simInit :: (*'s* :: *linorder*) *list* ⇒ (*'a* ⇒ *'s* ⇒ *'obs*)
⇒ *'a* ⇒ *'obs* ⇒ *'s spr-simWorldsRep*
where
spr-simInit envInit envObs ≡ *λa iobs*.
ODList.fromList [*s*. *s* ← *envInit*, *envObs a s* = *iobs*]

lemma *spr-simInit*:

assumes *iobs* $\in$ *envObs a* ‘ *set envInit*
shows *spr-simAbs* (*spr-simInit a iobs*)
$= spr\text{-}sim$ ‘ $\{ t' \in SPR.jkbpC.\ spr\text{-}jview\ a\ t' = spr\text{-}jviewInit\ a\ iobs \}$⟨*proof*⟩

7.3.3 Simulated observations

As the agent makes the same observation on the entire equivalence class, we arbitrarily choose the first element of the representation:

definition (**in** $-$)
spr-simObs :: $('a \Rightarrow 's \Rightarrow 'obs)$
$\Rightarrow 'a \Rightarrow ('s :: linorder)\ spr\text{-}simWorldsRep \Rightarrow 'obs$
where
spr-simObs envObs $\equiv \lambda a.\ envObs\ a \circ ODList.hd$

lemma *spr-simObs*:
assumes *tC*: $t \in SPR.jkbpC$
assumes *ec*: *spr-simAbs ec* = *SPRsingle.sim-equiv-class a t*
shows *spr-simObs a ec* = *envObs a* (*tLast t*)⟨*proof*⟩

7.3.4 Evaluation

As the single-agent case is much simpler than the multi-agent ones, we define an evaluation function specialised to its representation.

Intuitively *eval* yields the subset of X where the formula holds, where X is taken to be a representation of a canonical equivalence class for *agent*.

fun (**in** $-$)
eval :: $(('s :: linorder) \Rightarrow 'p \Rightarrow bool)$
$\Rightarrow 's\ odlist \Rightarrow ('a, 'p)\ Kform \Rightarrow 's\ odlist$
where
eval val X (*Kprop p*) = *ODList.filter* ($\lambda s.$ *val s p*) *X*
| *eval val X* (*Knot* φ) = *ODList.difference X* (*eval val X* φ)
| *eval val X* (*Kand* φ ψ) = *ODList.intersect* (*eval val X* φ) (*eval val X* ψ)
| *eval val X* (*Kknows a* φ) = (*if eval val X* φ = *X then X else ODList.empty*)
| *eval val X* (*Kcknows as* φ) =
(*if as* = [] $\vee$ *eval val X* φ = *X then X else ODList.empty*)

In general this is less efficient than the tableau approach of Fagin et al. (1995, Proposition 3.2.1), which labels all states with all formulas. However it is often the case that the set of relevant worlds is much smaller than the set of all system states.

Showing that this corresponds with the standard models relation is routine.

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩
lemma *eval-models*:
assumes *ec*: *spr-simAbs ec* = *SPRsingle.sim-equiv-class agent t*
assumes *subj*: *subjective agent* φ
assumes *s*: $s \in$ *toSet ec*
shows *toSet* (*eval envVal ec* φ) $\neq \{\} \longleftrightarrow$ *spr-repMC* (*toSet ec*), $s \models \varphi$⟨*proof*⟩

7.3.5 Simulated actions

The actions enabled on a canonical equivalence class are those for which *eval* yields a non-empty set of states:

definition (**in** −)
spr-simAction :: ('*a*, '*p*, '*aAct*) *KBP* ⇒ (('*s* :: *linorder*) ⇒ '*p* ⇒ *bool*)
⇒ '*a* ⇒ '*s spr-simWorldsRep* ⇒ '*aAct list*
where
spr-simAction kbp envVal ≡ λ*a X*.
[*action gc*. *gc* ← *kbp*, *eval envVal X* (*guard gc*) ≠ *ODList.empty*]

The key lemma relates the agent's behaviour on an equivalence class to that on its representation:

lemma *spr-simAction-jAction*:
assumes *tC*: *t* ∈ *SPR.jkbpC*
assumes *ec*: *spr-simAbs ec* = *SPRsingle.sim-equiv-class agent t*
shows *set* (*spr-simAction agent ec*)
= *set* (*jAction* (*spr-repMC* (*toSet ec*)) (*tLast t*) *agent*)⟨*proof*⟩⟨*proof*⟩

The *Algorithm* locale requires the following lemma, which is a straightforward chaining of the above simulations.

lemma *spr-simAction*:
assumes *tC*: *t* ∈ *SPR.jkbpC*
and *ec*: *spr-simAbs ec* = *SPRsingle.sim-equiv-class a t*
shows *set* (*spr-simAction a ec*) = *set* (*jAction SPR.MC t a*)⟨*proof*⟩

7.3.6 Simulated transitions

It is straightforward to determine the possible successor states of a given canonical equivalence class X:

definition (**in** −)
spr-trans :: ('*a*, '*p*, '*aAct*) *KBP*
⇒ ('*s* ⇒ '*eAct list*)
⇒ ('*eAct* ⇒ ('*a* ⇒ '*aAct*) ⇒ '*s* ⇒ '*s*)
⇒ ('*s* ⇒ '*p* ⇒ *bool*)
⇒ '*a* ⇒ ('*s* :: *linorder*) *spr-simWorldsRep* ⇒ '*s list*
where
spr-trans kbp envAction envTrans val ≡ λ*a X*.
[*envTrans eact* (λ*a*'. *aact*) *s* .
s ← *toList X*, *eact* ← *envAction s*, *aact* ← *spr-simAction kbp val a X*]

Using this function we can determine the set of possible successor equivalence classes from X:

abbreviation (**in** −) *envObs-rel* :: ('*s* ⇒ '*obs*) ⇒ '*s* × '*s* ⇒ *bool* **where**
envObs-rel envObs ≡ λ(*s*, *s*'). *envObs s*' = *envObs s*

definition (**in** −)

```
spr-simTrans :: ('a, 'p, 'aAct) KBP
              ⇒ (('s::linorder) ⇒ 'eAct list)
              ⇒ ('eAct ⇒ ('a ⇒ 'aAct) ⇒ 's ⇒ 's)
              ⇒ ('s ⇒ 'p ⇒ bool)
              ⇒ ('a ⇒ 's ⇒ 'obs)
              ⇒ 'a ⇒ 's spr-simWorldsRep ⇒ 's spr-simWorldsRep list
where
  spr-simTrans kbp envAction envTrans val envObs ≡ λa X.
    map ODList.fromList (partition (envObs-rel (envObs a))
                         (spr-trans kbp envAction envTrans val a X))⟨proof⟩⟨proof⟩
```

The *partition* function splits a list into equivalence classes under the given equivalence relation.

The property asked for by the *Algorithm* locale follows from the properties of *partition* and *spr-trans*:

```
lemma spr-simTrans:
  assumes tC: t ∈ SPR.jkbpC
  assumes ec: spr-simAbs ec = SPRsingle.sim-equiv-class a t
  shows spr-simAbs ‘ set (spr-simTrans a ec)
      = { SPRsingle.sim-equiv-class a (t' ⇝ s)
          |t' s. t' ⇝ s ∈ SPR.jkbpC ∧ spr-jview a t' = spr-jview a t}⟨proof⟩

end
```

7.3.7 Maps

As in §7.1.7, we use a pair of tries and an association list to handle the automata representation. Recall that the keys of these tries are lists of system states.

```
type-synonym ('s, 'obs) spr-trans-trie = ('s, ('obs, 's odlist) mapping) trie
type-synonym ('s, 'aAct) spr-acts-trie = ('s, ('s, 'aAct) trie) trie⟨proof⟩
```

7.3.8 Locale instantiation

The above is sufficient to instantiate the *Algorithm* locale.

```
sublocale FiniteSingleAgentEnvironment
        < SPRsingle: Algorithm
            jkbp envInit envAction envTrans envVal
            spr-jview envObs spr-jviewInit spr-jviewIncr
            spr-sim spr-simRels spr-simVal
            spr-simAbs spr-simObs spr-simInit spr-simTrans spr-simAction
            trie-odlist-MapOps trans-MapOps⟨proof⟩⟨proof⟩⟨proof⟩
```

We use this theory to synthesise a solution to the robot of §1 in §8.1.

record (**overloaded**) $('a,\ 'es,\ 'ps)\ \mathit{BEState} =$
$\quad es :: 'es$
$\quad ps :: ('a \times 'ps)\ \mathit{odlist}$

locale $\mathit{FiniteDetBroadcastEnvironment} =$
$\quad \mathit{Environment\ jkbp\ envInit\ envAction\ envTrans\ envVal\ envObs}$
$\quad$ **for** $\mathit{jkbp} :: 'a \Rightarrow ('a :: \{\mathit{finite},\mathit{linorder}\},\ 'p,\ '\mathit{aAct})\ \mathit{KBP}$
$\quad$ **and** $\mathit{envInit}$
$\qquad :: ('a,\ 'es :: \{\mathit{finite},\mathit{linorder}\},\ 'as :: \{\mathit{finite},\mathit{linorder}\})\ \mathit{BEState\ list}$
$\quad$ **and** $\mathit{envAction} :: ('a,\ 'es,\ 'as)\ \mathit{BEState} \Rightarrow '\mathit{eAct\ list}$
$\quad$ **and** $\mathit{envTrans} :: '\mathit{eAct} \Rightarrow ('a \Rightarrow '\mathit{aAct})$
$\qquad \Rightarrow ('a,\ 'es,\ 'as)\ \mathit{BEState} \Rightarrow ('a,\ 'es,\ 'as)\ \mathit{BEState}$
$\quad$ **and** $\mathit{envVal} :: ('a,\ 'es,\ 'as)\ \mathit{BEState} \Rightarrow 'p \Rightarrow \mathit{bool}$
$\quad$ **and** $\mathit{envObs} :: 'a \Rightarrow ('a,\ 'es,\ 'as)\ \mathit{BEState} \Rightarrow ('\mathit{cobs} \times '\mathit{as\ option})$

\+ **fixes** $\mathit{agents} :: 'a\ \mathit{odlist}$
fixes $\mathit{envObsC} :: 'es \Rightarrow '\mathit{cobs}$
defines $\mathit{envObs}\ a\ s \equiv (\mathit{envObsC}\ (es\ s),\ \mathit{ODList.lookup}\ (ps\ s)\ a)$
assumes agents: $\mathit{ODList.toSet\ agents} = \mathit{UNIV}$
assumes $\mathit{envTrans}$: $\forall\, s\ s'\ a\ \mathit{eact}\ \mathit{eact}'\ \mathit{aact}\ \mathit{aact}'.$
$\qquad \mathit{ODList.lookup}\ (ps\ s)\ a = \mathit{ODList.lookup}\ (ps\ s')\ a \wedge \mathit{aact}\ a = \mathit{aact}'\ a$
$\qquad \longrightarrow \mathit{ODList.lookup}\ (ps\ (\mathit{envTrans\ eact\ aact}\ s))\ a$
$\qquad\quad = \mathit{ODList.lookup}\ (ps\ (\mathit{envTrans}\ \mathit{eact}'\ \mathit{aact}'\ s'))\ a$
assumes $\mathit{jkbpDet}$: $\forall\, a.\ \forall\, t \in \mathit{SPR.jkbpC}.\ \mathit{length}\ (\mathit{jAction\ SPR.MC}\ t\ a) \leq 1$

Figure 5: Finite broadcast environments with a deterministic JKBP.

7.4 Perfect Recall in Deterministic Broadcast Environments

It is well known that simultaneous broadcast has the effect of making information *common knowledge*; roughly put, the agents all learn the same things at the same time as the system evolves, so the relation amongst the agents' states of knowledge never becomes more complex than it is in the initial state (Fagin et al. 1995, Chapter 6). For this reason we might hope to find finite-state implementations of JKBPs in broadcast environments.

The broadcast assumption by itself is insufficient in general, however (van der Meyden 1996, §7), and so we need to further constrain the scenario. Here we require that for each canonical trace the JKBP prescribes at most one action. In practice this constraint is easier to verify than the circularity would suggest; we return to this point at the end of this section.

⟨*proof*⟩⟨*proof*⟩

We encode our expectations of the scenario in the *FiniteBroadcastEnvironment* locale of Figure 5. The broadcast is modelled by having all agents make the same common observation of the shared state of type $'es$. We also allow each agent to maintain a private state of type $'ps$; that other agents cannot influence it or directly observe it is enforced by the constraint *envTrans* and the definition of *envObs*.

We do however allow the environment's protocol to be non-deterministic and a function of the entire system state, including private states.

context *FiniteDetBroadcastEnvironment*
begin⟨*proof*⟩

We seek a suitable simulation space by considering what determines an agent's knowledge. Intuitively any set of traces that is relevant to the agents' states of knowledge with respect to $t \in jkbpC$ need include only those with the same common observation as t:

definition *tObsC* :: ('*a*, '*es*, '*as*) *BEState Trace* ⇒ '*cobs Trace* **where**
  tObsC ≡ *tMap* (*envObsC* ∘ *es*)

Clearly this is an abstraction of the SPR jview of the given trace.

lemma *spr-jview-tObsC*:
  assumes *spr-jview a t* = *spr-jview a t'*
  shows *tObsC t* = *tObsC t'*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

Unlike the single-agent case of §7.3, it is not sufficient for a simulation to record only the final states; we need to relate the initial private states of the agents with the final states they consider possible, as the initial states may contain information that is not common knowledge. This motivates the following abstraction:

definition
  tObsC-abs :: ('*a*, '*es*, '*as*) *BEState Trace* ⇒ ('*a*, '*es*, '*as*) *BEState Relation*
where
  tObsC-abs t ≡ { (*tFirst t'*, *tLast t'*)
            |*t'*. *t'* ∈ *SPR.jkbpC* ∧ *tObsC t'* = *tObsC t*}⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

end

We use the following record to represent the worlds of the simulated Kripke structure:

record (**overloaded**) ('*a*, '*es*, '*as*) *spr-simWorld* =
  sprFst :: ('*a*, '*es*, '*as*) *BEState*
  sprLst :: ('*a*, '*es*, '*as*) *BEState*
  sprCRel :: ('*a*, '*es*, '*as*) *BEState Relation*
⟨*proof*⟩
context *FiniteDetBroadcastEnvironment*
begin

The simulation of a trace $t \in jkbpC$ records its initial and final states, and the relation between initial and final states of all commonly-plausible traces:

definition
  spr-sim :: ('*a*, '*es*, '*as*) *BEState Trace* ⇒ ('*a*, '*es*, '*as*) *spr-simWorld*
where

spr-sim ≡ λ*t*. (| *sprFst* = *tFirst t*, *sprLst* = *tLast t*, *sprCRel* = *tObsC-abs t* |)

The associated Kripke structure relates two worlds for an agent if the agent's observation on the the first and last states corresponds, and the worlds have the same common observation relation. As always, we evaluate propositions on the final state of the trace.

definition
spr-simRels :: *'a* ⇒ (*'a*, *'es*, *'as*) *spr-simWorld Relation*
where
spr-simRels ≡ λ*a*. { (*s*, *s'*) |*s s'*.
envObs a (*sprFst s*) = *envObs a* (*sprFst s'*)
∧ *envObs a* (*sprLst s*) = *envObs a* (*sprLst s'*)
∧ *sprCRel s* = *sprCRel s'* }

definition *spr-simVal* :: (*'a*, *'es*, *'as*) *spr-simWorld* ⇒ *'p* ⇒ *bool* **where**
spr-simVal ≡ *envVal* ∘ *sprLst*

abbreviation
spr-simMC ≡ *mkKripke* (*spr-sim* ' *SPR.jkbpC*) *spr-simRels spr-simVal*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*pro*

All the properties of a simulation are easy to show for *spr-sim* except for reverse simulation.

The critical lemma states that if we have two traces that yield the same common observations, and an agent makes the same observation on their initial states, then that agent's private states at each point on the two traces are identical.

lemma *spr-jview-det-ps*:
assumes *tt'C*: {*t*, *t'*} ⊆ *SPR.jkbpC*
assumes *obsCtt'*: *tObsC t* = *tObsC t'*
assumes *first*: *envObs a* (*tFirst t*) = *envObs a* (*tFirst t'*)
shows *tMap* (λ*s*. *ODList.lookup* (*ps s*) *a*) *t*
= *tMap* (λ*s*. *ODList.lookup* (*ps s*) *a*) *t'*⟨*proof*⟩⟨*proof*⟩

The proof proceeds by lock-step induction over t and t', appealing to the *jkbpDet* assumption, the definition of *envObs* and the constraint *envTrans*.

It is then a short step to showing reverse simulation, and hence simulation:

lemma *spr-sim*: *sim SPR.MC spr-simMC spr-sim*⟨*proof*⟩
end

sublocale *FiniteDetBroadcastEnvironment*
< *SPRdet*: *SimIncrEnvironment jkbp envInit envAction envTrans envVal*
spr-jview envObs spr-jviewInit spr-jviewIncr
spr-sim spr-simRels spr-simVal⟨*proof*⟩

7.4.1 Representations

As before we canonically represent the quotient of the simulated worlds (*'a*, *'es*, *'as*) *spr-simWorld* under *spr-simRels* using ordered, distinct lists. In particular, we use the type $('a \times 'a)$ *odlist* (abbreviated *'a odrelation*) to canonically represent relations.

context *FiniteDetBroadcastEnvironment*
begin

type-synonym (**in** −) $('a,\ 'es,\ 'as)$ *spr-simWorldsECRep*
$= ('a,\ 'es,\ 'as)$ *BEState odrelation*
type-synonym (**in** −) $('a,\ 'es,\ 'as)$ *spr-simWorldsRep*
$= ('a,\ 'es,\ 'as)$ *spr-simWorldsECRep* $\times$ $('a,\ 'es,\ 'as)$ *spr-simWorldsECRep*

We can abstract such a representation into a set of simulated equivalence classes:

definition
spr-simAbs :: $('a,\ 'es,\ 'as)$ *spr-simWorldsRep*
$\Rightarrow ('a,\ 'es,\ 'as)$ *spr-simWorld set*
where
$spr\text{-}simAbs \equiv \lambda(cec,\ aec).\ \{\ (\!|\ sprFst = s,\ sprLst = s',\ sprCRel = toSet\ cec\ |\!)$
$|s\ s'.\ (s,\ s') \in toSet\ aec\ \}$

Assuming X represents a simulated equivalence class for $t \in jkbpC$, we can decompose *spr-simAbs X* in terms of *tObsC-abs t* and *agent-abs t*:

definition
agent-abs :: $'a \Rightarrow ('a,\ 'es,\ 'as)$ *BEState Trace*
$\Rightarrow ('a,\ 'es,\ 'as)$ *BEState Relation*
where
$agent\text{-}abs\ a\ t \equiv \{\ (tFirst\ t',\ tLast\ t')$
$|t'.\ t' \in SPR.jkbpC \wedge spr\text{-}jview\ a\ t' = spr\text{-}jview\ a\ t\ \}$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

This representation is canonical on the domain of interest (though not in general):

lemma *spr-simAbs-inj-on*:
$inj\text{-}on\ spr\text{-}simAbs\ \{\ x\ .\ spr\text{-}simAbs\ x \in SPRdet.jkbpSEC\ \}$⟨*proof*⟩

The following sections make use of a Kripke structure constructed over *tObsC-abs t* for some canonical trace *t*. Note that we use the relation in the generated code.

type-synonym (**in** −) $('a,\ 'es,\ 'as)$ *spr-simWorlds*
$= ('a,\ 'es,\ 'as)$ *BEState* $\times$ $('a,\ 'es,\ 'as)$ *BEState*

definition (**in** −)
spr-repRels :: $('a \Rightarrow ('a,\ 'es,\ 'as)\ BEState \Rightarrow 'cobs \times 'as\ option)$
$\Rightarrow 'a \Rightarrow ('a,\ 'es,\ 'as)$ *spr-simWorlds Relation*
where
$spr\text{-}repRels\ envObs \equiv \lambda a.\ \{\ ((u,\ v),\ (u',\ v')).$
$envObs\ a\ u = envObs\ a\ u' \wedge envObs\ a\ v = envObs\ a\ v'\ \}$

definition
spr-repVal :: $('a,\ 'es,\ 'as)$ *spr-simWorlds* $\Rightarrow 'p \Rightarrow bool$
where

spr-repVal ≡ *envVal* ∘ *snd*

abbreviation
spr-repMC :: ('*a*, '*es*, '*as*) *BEState Relation*
⇒ ('*a*, '*p*, ('*a*, '*es*, '*as*) *spr-simWorlds*) *KripkeStructure*
where
spr-repMC ≡ λ*tcobsR*. *mkKripke tcobsR* (*spr-repRels envObs*) *spr-repVal*⟨*proof*⟩⟨*proof*⟩

As before we can show that this Kripke structure is adequate for a particular canonical trace *t* by showing that it simulates *spr-repMC* We introduce an intermediate structure:

abbreviation
spr-jkbpCSt :: ('*a*, '*es*, '*as*) *BEState Trace* ⇒ ('*a*, '*es*, '*as*) *spr-simWorld set*
where
spr-jkbpCSt t ≡ *spr-sim* ' { *t'* . *t'* ∈ *SPR.jkbpC* ∧ *tObsC t* = *tObsC t'* }

abbreviation
spr-simMCt :: ('*a*, '*es*, '*as*) *BEState Trace*
⇒ ('*a*, '*p*, ('*a*, '*es*, '*as*) *spr-simWorld*) *KripkeStructure*
where
spr-simMCt t ≡ *mkKripke* (*spr-jkbpCSt t*) *spr-simRels spr-simVal*

definition
spr-repSim :: ('*a*, '*es*, '*as*) *spr-simWorld* ⇒ ('*a*, '*es*, '*as*) *spr-simWorlds*
where
spr-repSim ≡ λ*s*. (*sprFst s*, *sprLst s*)⟨*proof*⟩⟨*proof*⟩

lemma *spr-repSim*:
assumes *tC*: *t* ∈ *SPR.jkbpC*
shows *sim* (*spr-simMCt t*)
((*spr-repMC* ∘ *sprCRel*) (*spr-sim t*))
spr-repSim⟨*proof*⟩

As before we define a set of constants that satisfy the *Algorithm* locale given the assumptions of the *FiniteDetBroadcastEnvironment* locale.

7.4.2 Initial states

The initial states for agent *a* given an initial observation *iobs* consist of the set of states that yield a common observation consonant with *iobs* paired with the set of states where *a* observes *iobs*:

definition (**in** −)
spr-simInit ::
('*a*, '*es*, '*as*) *BEState list* ⇒ ('*es* ⇒ '*cobs*)
⇒ ('*a* ⇒ ('*a*, '*es*, '*as*) *BEState* ⇒ '*cobs* × '*obs*)
⇒ '*a* ⇒ ('*cobs* × '*obs*)
⇒ ('*a* :: *linorder*, '*es* :: *linorder*, '*as* :: *linorder*) *spr-simWorldsRep*
where

spr-simInit envInit envObsC envObs ≡ *λa iobs*.
(*ODList.fromList* [(*s*, *s*). *s* ← *envInit*, *envObsC* (*es s*) = *fst iobs*],
ODList.fromList [(*s*, *s*). *s* ← *envInit*, *envObs a s* = *iobs*])

lemma *spr-simInit*:
assumes *iobs* ∈ *envObs a* ' *set envInit*
shows *spr-simAbs* (*spr-simInit a iobs*)
= *spr-sim* ' { *t'* ∈ *SPR.jkbpC*. *spr-jview a t'* = *spr-jviewInit a iobs* }⟨*proof*⟩

7.4.3 Simulated observations

An observation can be made at any element of the representation of a simulated equivalence class of a canonical trace:

definition (**in** −)
spr-simObs ::
('*es* ⇒ '*cobs*)
⇒ '*a* ⇒ ('*a* :: *linorder*, '*es* :: *linorder*, '*as* :: *linorder*) *spr-simWorldsRep*
⇒ '*cobs* × '*as option*
where
spr-simObs envObsC ≡ *λa*. (*λs*. (*envObsC* (*es s*), *ODList.lookup* (*ps s*) *a*))
∘ *snd* ∘ *ODList.hd* ∘ *snd*

lemma *spr-simObs*:
assumes *tC*: *t* ∈ *SPR.jkbpC*
assumes *ec*: *spr-simAbs ec* = *SPRdet.sim-equiv-class a t*
shows *spr-simObs a ec* = *envObs a* (*tLast t*)⟨*proof*⟩

7.4.4 Evaluation

As for the clock semantics (§7.1.4), we use the general evalation function *evalS*. Once again we propositions are used to filter the set of possible worlds X:

abbreviation (**in** −)
spr-evalProp ::
(('*a*::*linorder*, '*es*::*linorder*, '*as*::*linorder*) *BEState* ⇒ '*p* ⇒ *bool*)
⇒ ('*a*, '*es*, '*as*) *BEState odrelation*
⇒ '*p* ⇒ ('*a*, '*es*, '*as*) *BEState odrelation*
where
spr-evalProp envVal ≡ *λX p*. *ODList.filter* (*λs*. *envVal* (*snd s*) *p*) *X*

The knowledge operation computes the subset of possible worlds *cec* that yield the same observation as *s* for agent *a*:

definition (**in** −)
spr-knowledge ::
('*a* ⇒ ('*a*::*linorder*, '*es*::*linorder*, '*as*::*linorder*) *BEState*
⇒ '*cobs* × '*as option*)
⇒ ('*a*, '*es*, '*as*) *BEState odrelation*
⇒ '*a* ⇒ ('*a*, '*es*, '*as*) *spr-simWorlds*
⇒ ('*a*, '*es*, '*as*) *spr-simWorldsECRep*

where
  spr-knowledge envObs cec ≡ λ*a s*.
    ODList.fromList [s' . s' ← *toList cec*, (s, s') ∈ *spr-repRels envObs a*]⟨*proof*⟩

Similarly the common knowledge operation computes the transitive closure (Sternagel and Thiemann 2011) of the union of the knowledge relations for the agents *as*:

definition (**in** −)
  spr-commonKnowledge ::
    ($'a$ ⇒ ($'a$::*linorder*, $'es$::*linorder*, $'as$::*linorder*) *BEState*
        ⇒ $'cobs$ × $'as$ *option*)
      ⇒ ($'a$, $'es$, $'as$) *BEState odrelation*
      ⇒ $'a$ *list*
      ⇒ ($'a$, $'es$, $'as$) *spr-simWorlds*
      ⇒ ($'a$, $'es$, $'as$) *spr-simWorldsECRep*
where
  spr-commonKnowledge envObs cec ≡ λ*as s*.
    let r = λ*a*. *ODList.fromList*
              [(s', s'') . s' ← *toList cec*, s'' ← *toList cec*,
                            (s', s'') ∈ *spr-repRels envObs a*];
        R = *toList* (*ODList.big-union r as*)
     in ODList.fromList (*memo-list-trancl R s*)⟨*proof*⟩

The evaluation function evaluates a subjective knowledge formula on the representation of an equivalence class:

definition (**in** −)
  eval envVal envObs ≡ λ(*cec*, *X*).
    evalS (*spr-evalProp envVal*)
          (*spr-knowledge envObs cec*)
          (*spr-commonKnowledge envObs cec*)
        X⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

This function corresponds with the standard semantics:

lemma *eval-models*:
  assumes *tC*: *t* ∈ *SPR.jkbpC*
  assumes *ec*: *spr-simAbs ec* = *SPRdet.sim-equiv-class a t*
  assumes *subj-phi*: *subjective a* φ
  assumes *s*: *s* ∈ *toSet* (*snd ec*)
  shows *eval envVal envObs ec* φ ⟷ *spr-repMC* (*toSet* (*fst ec*)), *s* ⊨ φ⟨*proof*⟩

7.4.5 Simulated actions

From a common equivalence class and a subjective equivalence class for agent *a*, we can compute the actions enabled for *a*:

definition (**in** −)
  spr-simAction ::
      ($'a$, $'p$, $'aAct$) *JKBP* ⇒ (($'a$, $'es$, $'as$) *BEState* ⇒ $'p$ ⇒ *bool*)
    ⇒ ($'a$ ⇒ ($'a$, $'es$, $'as$) *BEState* ⇒ $'cobs$ × $'as$ *option*)

⇒ 'a
⇒ ('a::linorder, 'es::linorder, 'as::linorder) spr-simWorldsRep
⇒ 'aAct list
where
spr-simAction jkbp envVal envObs ≡ *λa ec.*
[*action gc. gc* ← *jkbp a, eval envVal envObs ec* (*guard gc*)]

Using the above result about evaluation, we can relate *spr-simAction* to *jAction*. Firstly, *spr-simAction* behaves the same as *jAction* using the *spr-repMC* structure:

lemma *spr-action-jaction*:
assumes *tC*: *t* ∈ *SPR.jkbpC*
assumes *ec*: *spr-simAbs ec* = *SPRdet.sim-equiv-class a t*
shows *set* (*spr-simAction a ec*)
= *set* (*jAction* (*spr-repMC* (*toSet* (*fst ec*))) (*tFirst t*, *tLast t*) *a*)⟨*proof*⟩⟨*proof*⟩

We can connect the agent's choice of actions on the *spr-repMC* structure to those on the *SPR.MC* structure using our earlier results about actions being preserved by generated models and simulations.

lemma *spr-simAction*:
assumes *tC*: *t* ∈ *SPR.jkbpC*
assumes *ec*: *spr-simAbs ec* = *SPRdet.sim-equiv-class a t*
shows *set* (*spr-simAction a ec*) = *set* (*jAction SPR.MC t a*)⟨*proof*⟩

7.4.6 Simulated transitions

The story of simulated transitions takes some doing. We begin by computing the successor relation of a given equivalence class *X* with respect to the common equivalence class *cec*:

abbreviation (**in** −)
spr-jAction jkbp envVal envObs cec s ≡ *λa.*
spr-simAction jkbp envVal envObs a (*cec*, *spr-knowledge envObs cec a s*)

definition (**in** −)
spr-trans :: 'a odlist
⇒ ('a, 'p, 'aAct) JKBP
⇒ (('a::linorder, 'es::linorder, 'as::linorder) BEState ⇒ 'eAct list)
⇒ ('eAct ⇒ ('a ⇒ 'aAct)
⇒ ('a, 'es, 'as) BEState ⇒ ('a, 'es, 'as) BEState)
⇒ (('a, 'es, 'as) BEState ⇒ 'p ⇒ bool)
⇒ ('a ⇒ ('a, 'es, 'as) BEState ⇒ 'cobs × 'as option)
⇒ ('a, 'es, 'as) spr-simWorldsECRep
⇒ ('a, 'es, 'as) spr-simWorldsECRep
⇒ (('a, 'es, 'as) BEState × ('a, 'es, 'as) BEState) list
where
spr-trans agents jkbp envAction envTrans envVal envObs ≡ *λcec X.*
[(*initialS*, *succS*) .
(*initialS*, *finalS*) ← *toList X*,

```
  eact ← envAction finalS,
  succS ← [ envTrans eact aact finalS .
            aact ← listToFuns (spr-jAction jkbp envVal envObs cec
                                          (initialS, finalS))
                         (toList agents) ] ]
```

We will split the result of this function according to the common observation and also agent a's observation, where a is the agent we are constructing the automaton for.

```
definition (in −)
  spr-simObsC :: ('es ⇒ 'cobs)
           ⇒ (('a::linorder, 'es::linorder, 'as::linorder) BEState
             × ('a, 'es, 'as) BEState) odlist
           ⇒ 'cobs
where
  spr-simObsC envObsC ≡ envObsC ∘ es ∘ snd ∘ ODList.hd

abbreviation (in −)
  envObs-rel :: (('a, 'es, 'as) BEState ⇒ 'cobs × 'as option)
          ⇒ ('a, 'es, 'as) spr-simWorlds × ('a, 'es, 'as) spr-simWorlds ⇒ bool
where
  envObs-rel envObs ≡ λ(s, s'). envObs (snd s') = envObs (snd s)
```

The above combine to yield the successor equivalence classes like so:

```
definition (in −)
  spr-simTrans :: 'a odlist
          ⇒ ('a, 'p, 'aAct) JKBP
          ⇒ (('a::linorder, 'es::linorder, 'as::linorder) BEState ⇒ 'eAct list)
          ⇒ ('eAct ⇒ ('a ⇒ 'aAct)
              ⇒ ('a, 'es, 'as) BEState ⇒ ('a, 'es, 'as) BEState)
          ⇒ (('a, 'es, 'as) BEState ⇒ 'p ⇒ bool)
          ⇒ ('es ⇒ 'cobs)
          ⇒ ('a ⇒ ('a, 'es, 'as) BEState ⇒ 'cobs × 'as option)
           ⇒ 'a
           ⇒ ('a, 'es, 'as) spr-simWorldsRep
           ⇒ ('a, 'es, 'as) spr-simWorldsRep list
where
  spr-simTrans agents jkbp envAction envTrans envVal envObsC envObs ≡ λa ec.
    let aSuccs = spr-trans agents jkbp envAction envTrans envVal envObs
                        (fst ec) (snd ec);
        cec' = ODList.fromList
                (spr-trans agents jkbp envAction envTrans envVal envObs
                        (fst ec) (fst ec))
     in [ (ODList.filter (λs. envObsC (es (snd s)) = spr-simObsC envObsC aec') cec',
          aec') .
           aec' ← map ODList.fromList (partition (envObs-rel (envObs a)) aSuccs)
]⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩
```

Showing that *spr-simTrans* works requires a series of auxiliary lemmas that show we do in fact compute the correct successor equivalence classes. We elide the unedifying details, skipping straight to the lemma that the *Algorithm* locale expects:

lemma *spr-simTrans*:
  assumes *tC*: $t \in SPR.jkbpC$
  assumes *ec*: $spr\text{-}simAbs\ ec = SPRdet.sim\text{-}equiv\text{-}class\ a\ t$
  shows $spr\text{-}simAbs$ ' $set\ (spr\text{-}simTrans\ a\ ec)$
    $= \{ SPRdet.sim\text{-}equiv\text{-}class\ a\ (t' \leadsto s)$
      $| t'\ s.\ t' \leadsto s \in SPR.jkbpC \land spr\text{-}jview\ a\ t' = spr\text{-}jview\ a\ t\} \langle proof \rangle$

The explicit-state approach sketched above is quite inefficient, and also some distance from the symbolic techniques we use in §**??**. However it does suffice to demonstrate the theory on the muddy children example in §8.2.

end

7.4.7 Maps

As always we use a pair of tries. The domain of these maps is the pair of relations.

type-synonym ('*a*, '*es*, '*obs*, '*as*) *trans-trie*
  = (('*a*, '*es*, '*as*) *BEState*,
    (('*a*, '*es*, '*as*) *BEState*,
     (('*a*, '*es*, '*as*) *BEState*,
      (('*a*, '*es*, '*as*) *BEState*,
       ('*obs*, ('*a*, '*es*, '*as*) *spr-simWorldsRep*) *mapping*) *trie*) *trie*) *trie*) *trie*

type-synonym
  ('*a*, '*es*, '*aAct*, '*as*) *acts-trie*
  = (('*a*, '*es*, '*as*) *BEState*,
    (('*a*, '*es*, '*as*) *BEState*,
     (('*a*, '*es*, '*as*) *BEState*,
      (('*a*, '*es*, '*as*) *BEState*, '*aAct*) *trie*) *trie*) *trie*) *trie*⟨*proof*⟩⟨*proof*⟩

This suffices to placate the *Algorithm* locale.

sublocale *FiniteDetBroadcastEnvironment*
    < *SPRdet*: *Algorithm*
      jkbp envInit envAction envTrans envVal
      spr-jview envObs spr-jviewInit spr-jviewIncr
      spr-sim spr-simRels spr-simVal
      spr-simAbs spr-simObs spr-simInit spr-simTrans spr-simAction
      acts-MapOps trans-MapOps⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

As we remarked earlier in this section, in general it may be difficult to establish the determinacy of a KBP as it is a function of the environment. However in many cases determinism is syntactically manifest as the guards are logically disjoint, independently of the knowledge subformulas. The following lemma generates the required proof obligations for this case:

lemma (**in** *PreEnvironmentJView*) *jkbpDetI*:

assumes *tC*: $t \in jkbpC$
assumes *jkbpSynDet*: $\forall a.\ distinct\ (map\ guard\ (jkbp\ a))$
assumes *jkbpSemDet*: $\forall a\ gc\ gc'.$
$gc \in set\ (jkbp\ a) \wedge gc' \in set\ (jkbp\ a) \wedge t \in jkbpC$
$\longrightarrow guard\ gc = guard\ gc' \vee \neg(MC,\ t \models guard\ gc \wedge MC,\ t \models guard\ gc')$
shows $length\ (jAction\ MC\ t\ a) \leq 1$⟨*proof*⟩

The scenario presented here is a variant of the broadcast environments treated by van der Meyden (1996), which we cover in the next section.

7.5 Perfect Recall in Non-deterministic Broadcast Environments

record $('a,\ 'ePubAct,\ 'es,\ 'pPubAct,\ 'ps)\ BEState =$
$es :: 'es$
$ps :: 'a \Rightarrow 'ps$
$pubActs :: 'ePubAct \times ('a \Rightarrow 'pPubAct)$

locale *FiniteBroadcastEnvironment* =
Environment jkbp envInit envAction envTrans envVal envObs
for $jkbp :: ('a :: finite,\ 'p,\ ('pPubAct :: finite \times 'ps :: finite))\ JKBP$
and *envInit*
$:: ('a,\ 'ePubAct :: finite,\ 'es :: finite,\ 'pPubAct,\ 'ps)\ BEState\ list$
and $envAction :: ('a,\ 'ePubAct,\ 'es,\ 'pPubAct,\ 'ps)\ BEState$
$\Rightarrow ('ePubAct \times 'ePrivAct)\ list$
and $envTrans :: ('ePubAct \times 'ePrivAct)$
$\Rightarrow ('a \Rightarrow ('pPubAct \times 'ps))$
$\Rightarrow ('a,\ 'ePubAct,\ 'es,\ 'pPubAct,\ 'ps)\ BEState$
$\Rightarrow ('a,\ 'ePubAct,\ 'es,\ 'pPubAct,\ 'ps)\ BEState$
and $envVal :: ('a,\ 'ePubAct,\ 'es,\ 'pPubAct,\ 'ps)\ BEState \Rightarrow 'p \Rightarrow bool$
and $envObs :: 'a \Rightarrow ('a,\ 'ePubAct,\ 'es,\ 'pPubAct,\ 'ps)\ BEState$
$\Rightarrow ('cobs \times 'ps \times ('ePubAct \times ('a \Rightarrow 'pPubAct)))$

\+ **fixes** $envObsC :: 'es \Rightarrow 'cobs$
and $envActionES :: 'es \Rightarrow ('ePubAct \times ('a \Rightarrow 'pPubAct))$
$\Rightarrow ('ePubAct \times 'ePrivAct)\ list$
and $envTransES :: ('ePubAct \times 'ePrivAct) \Rightarrow ('a \Rightarrow 'pPubAct)$
$\Rightarrow 'es \Rightarrow 'es$
defines *envObs-def*: $envObs\ a \equiv (\lambda s.\ (envObsC\ (es\ s),\ ps\ s\ a,\ pubActs\ s))$
and *envAction-def*: $envAction\ s \equiv envActionES\ (es\ s)\ (pubActs\ s)$
and *envTrans-def*:
$envTrans\ eact\ aact\ s \equiv (\!|\ es = envTransES\ eact\ (fst \circ aact)\ (es\ s)$
$,\ ps = snd \circ aact$
$,\ pubActs = (fst\ eact,\ fst \circ aact)\ |\!)$

Figure 6: Finite broadcast environments with non-deterministic KBPs.

⟨*proof*⟩

For completeness we reproduce the results of van der Meyden (1996) regarding non-deterministic KBPs in broadcast environments.

The determinism requirement is replaced by the constraint that actions be split into public and private components, where the private part influences the agents' private states, and the public part is broadcast and recorded in the system state. Moreover the protocol of the environment is only a function of the environment state, and not the agents' private states. Once again an agent's view consists of the common observation and their private state. The situation is described by the locale in Figure 6. Note that as we do not intend to generate code for this case, we adopt more transparent but less effective representations.

Our goal in the following is to instantiate the *SimIncrEnvironment* locale with respect to the assumptions made in the *FiniteBroadcastEnvironment* locale. We begin by defining similar simulation machinery to the previous section.

context *FiniteBroadcastEnvironment*
begin

As for the deterministic variant, we abstract traces using the common observation. Note that this now includes the public part of the agents' actions.

definition
  tObsC :: ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState Trace*
    ⇒ ('*cobs* × '*ePubAct* × ('*a* ⇒ '*pPubAct*)) *Trace*
where
  tObsC ≡ *tMap* (λ*s*. (*envObsC* (*es s*), *pubActs s*))⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

Similarly we introduce common and agent-specific abstraction functions:

definition
  tObsC-abs :: ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState Trace*
    ⇒ ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState Relation*
where
  tObsC-abs t ≡ { (*tFirst t*′, *tLast t*′)
    |*t*′. *t*′ ∈ *SPR.jkbpC* ∧ *tObsC t*′ = *tObsC t* }

definition
  agent-abs :: '*a* ⇒ ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState Trace*
    ⇒ ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState Relation*
where
  agent-abs a t ≡ { (*tFirst t*′, *tLast t*′)
    |*t*′. *t*′ ∈ *SPR.jkbpC* ∧ *spr-jview a t*′ = *spr-jview a t* }⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩
end

The simulation is identical to that in the previous section:

record ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *SPRstate* =
  sprFst :: ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState*
  sprLst :: ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState*
  sprCRel :: ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState Relation*

context *FiniteBroadcastEnvironment*

begin

definition
  $spr\text{-}sim :: ('a, 'ePubAct, 'es, 'pPubAct, 'ps)\ BEState\ Trace$
    $\Rightarrow ('a, 'ePubAct, 'es, 'pPubAct, 'ps)\ SPRstate$
where
  $spr\text{-}sim \equiv \lambda t.\ (\!|\ sprFst = tFirst\ t,\ sprLst = tLast\ t,\ sprCRel = tObsC\text{-}abs\ t\ |\!)\langle proof\rangle$

The Kripke structure over simulated traces is also the same:

definition
  $spr\text{-}simRels :: 'a \Rightarrow ('a, 'ePubAct, 'es, 'pPubAct, 'ps)\ SPRstate\ Relation$
where
  $spr\text{-}simRels \equiv \lambda a.\ \{\ (s, s')\ |s\ s'.$
    $envObs\ a\ (sprFst\ s) = envObs\ a\ (sprFst\ s')$
    $\wedge\ envObs\ a\ (sprLst\ s) = envObs\ a\ (sprLst\ s')$
    $\wedge\ sprCRel\ s = sprCRel\ s'\ \}$

definition
  $spr\text{-}simVal :: ('a, 'ePubAct, 'es, 'pPubAct, 'ps)\ SPRstate \Rightarrow 'p \Rightarrow bool$
where
  $spr\text{-}simVal \equiv envVal \circ sprLst$

abbreviation
  $spr\text{-}simMC \equiv mkKripke\ (spr\text{-}sim\ `\ SPR.jkbpC)\ spr\text{-}simRels\ spr\text{-}simVal\langle proof\rangle$

As usual, showing that *spr-sim* is in fact a simulation is routine for all properties except for reverse simulation. For that we use proof techniques similar to those of Lomuscio et al. (2000): the goal is to show that, given $t \in jkbpC$, we can construct a trace $t' \in jkbpC$ indistinguishable from t by agent a, based on the public actions, the common observation and a's private and initial states.

To do this we define a splicing operation:

definition
  $sSplice :: 'a$
    $\Rightarrow ('a, 'ePubAct, 'es, 'pPubAct, 'ps)\ BEState$
    $\Rightarrow ('a, 'ePubAct, 'es, 'pPubAct, 'ps)\ BEState$
    $\Rightarrow ('a, 'ePubAct, 'es, 'pPubAct, 'ps)\ BEState$
where
  $sSplice\ a\ s\ s' \equiv s(\!|\ ps := (ps\ s)(a := ps\ s'\ a)\ |\!)\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle\langle proof\rangle$

The effect of $sSplice\ a\ s\ s'$ is to update s with a's private state in s'. The key properties are that provided the common observation on s and s' are the same, then agent a's observation on $sSplice\ a\ s\ s'$ is the same as at s', while everyone else's is the same as at s.

We hoist this operation pointwise to traces:

abbreviation
  $tSplice :: ('a, 'ePubAct, 'es, 'pPubAct, 'ps)\ BEState\ Trace$
    $\Rightarrow 'a$
    $\Rightarrow ('a, 'ePubAct, 'es, 'pPubAct, 'ps)\ BEState\ Trace$

$\Rightarrow$ ($'a$, $'ePubAct$, $'es$, $'pPubAct$, $'ps$) *BEState Trace*
(‹- ⋈- -› [55, 1000, 56] 55)
where
$t \bowtie_a t' \equiv tZip\ (sSplice\ a)\ t\ t'$⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

The key properties are that after splicing, if t and t' have the same common observation, then so does $t \bowtie_a t'$, and for all agents $a' \neq a$, the view a' has of $t \bowtie_a t'$ is the same as it has of t, while for a it is the same as t'.

We can conclude that provided the two traces are initially indistinguishable to a, and not commonly distinguishable, then $t \bowtie_a t'$ is a canonical trace:

lemma *tSplice-jkbpC*:
assumes *tt'*: $\{t, t'\} \subseteq SPR.jkbpC$
assumes *init*: $envObs\ a\ (tFirst\ t) = envObs\ a\ (tFirst\ t')$
assumes *tObsC*: $tObsC\ t = tObsC\ t'$
shows $t \bowtie_a t' \in SPR.jkbpC$⟨*proof*⟩⟨*proof*⟩

The proof is by induction over t and t', and depends crucially on the public actions being recorded in the state and commonly observed. Showing the reverse simulation property is then straightforward.

lemma *spr-sim*: *sim SPR.MC spr-simMC spr-sim*⟨*proof*⟩**end**

sublocale *FiniteBroadcastEnvironment*
< *SPR*: *SimIncrEnvironment jkbp envInit envAction envTrans envVal*
spr-jview envObs spr-jviewInit spr-jviewIncr
spr-sim spr-simRels spr-simVal⟨*proof*⟩

The algorithmic representations and machinery of the deterministic JKBP case suffice for this one too, and so we omit the details.

7.5.1 Perfect Recall in Independently-Initialised Non-deterministic Broadcast Environments

If the private and environment parts of the initial states are independent we can simplify the construction of the previous section and consider only sets of states rather than relations. This greatly reduces the state space that the algorithm traverses.

We capture this independence by adding some assumptions to the *FiniteBroadcastEnvironment* locale of Figure 6; the result is the *FiniteBroadcastEnvironmentIndependentInit* locale shown in Figure 7. We ask that the initial states be the Cartesian product of possible private and environment states; in other words there is nothing for the agents to learn about correlations amongst the initial states. As there are initially no public actions from the previous round, we use the *default* class to indicate that there is a fixed but arbitrary choice to be made here.

Again we introduce common and agent-specific abstraction functions:

context *FiniteBroadcastEnvironmentIndependentInit*

locale *FiniteBroadcastEnvironmentIndependentInit* =
FiniteBroadcastEnvironment jkbp envInit envAction envTrans envVal envObs envObsC envActionES envTransES
for *jkbp* :: ('*a*::*finite*, '*p*, ('*pPubAct*::{*default*,*finite*} × '*ps*::*finite*)) *JKBP*
and *envInit* :: ('*a*, '*ePubAct* :: {*default*, *finite*}, '*es* :: *finite*, '*pPubAct*, '*ps*) *BEState list*
and *envAction* :: ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState* ⇒ ('*ePubAct* × '*ePrivAct*) *list*
and *envTrans* :: ('*ePubAct* × '*ePrivAct*)
⇒ ('*a* ⇒ ('*pPubAct* × '*ps*))
⇒ ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState*
⇒ ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState*
and *envVal* :: ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState* ⇒ '*p* ⇒ *bool*
and *envObs* :: '*a* ⇒ ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState*
⇒ ('*cobs* × '*ps* × ('*ePubAct* × ('*a* ⇒ '*pPubAct*)))
and *envObsC* :: '*es* ⇒ '*cobs*
and *envActionES* :: '*es* ⇒ ('*ePubAct* × ('*a* ⇒ '*pPubAct*))
⇒ ('*ePubAct* × '*ePrivAct*) *list*
and *envTransES* :: ('*ePubAct* × '*ePrivAct*) ⇒ ('*a* ⇒ '*pPubAct*)
⇒ '*es* ⇒ '*es*

\+ **fixes** *agents* :: '*a list*
fixes *envInitBits* :: '*es list* × ('*a* ⇒ '*ps list*)
defines *envInit-def*:
envInit ≡ [(| *es* = *esf*, *ps* = *psf*, *pubActs* = (*default*, λ-. *default*) |)
. *psf* ← *listToFuns* (*snd envInitBits*) *agents*
, *esf* ← *fst envInitBits*]
assumes *agents*: *set agents* = *UNIV distinct agents*

Figure 7: Finite broadcast environments with non-deterministic KBPs, where the initial private and environment states are independent.

begin

definition
tObsC-ii-abs :: ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState Trace*
⇒ ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState set*
where *tObsC-ii-abs t* ≡
{ *tLast t'* |*t'*. *t'* ∈ *SPR.jkbpC* ∧ *tObsC t'* = *tObsC t* }

definition
agent-ii-abs :: '*a* ⇒ ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState Trace*
⇒ ('*a*, '*ePubAct*, '*es*, '*pPubAct*, '*ps*) *BEState set*
where *agent-ii-abs a t* ≡
{ *tLast t'* |*t'*. *t'* ∈ *SPR.jkbpC* ∧ *spr-jview a t'* = *spr-jview a t* }⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

The simulation is similar to the single-agent case (§7.3); for a given canonical trace t it pairs the set of worlds that any agent considers possible with the final state of t:

type-synonym (**in** −) ($'a$, $'ePubAct$, $'es$, $'pPubAct$, $'ps$) *SPRstate* =
($'a$, $'ePubAct$, $'es$, $'pPubAct$, $'ps$) *BEState set*
× ($'a$, $'ePubAct$, $'es$, $'pPubAct$, $'ps$) *BEState*

definition
spr-ii-sim :: ($'a$, $'ePubAct$, $'es$, $'pPubAct$, $'ps$) *BEState Trace*
⇒ ($'a$, $'ePubAct$, $'es$, $'pPubAct$, $'ps$) *SPRstate*
where *spr-ii-sim* ≡ $\lambda t.$ (*tObsC-ii-abs t*, *tLast t*)⟨*proof*⟩

The Kripke structure over simulated traces is also quite similar:

definition
spr-ii-simRels :: $'a$ ⇒ ($'a$, $'ePubAct$, $'es$, $'pPubAct$, $'ps$) *SPRstate Relation*
where *spr-ii-simRels* ≡ $\lambda a.$
{ (s, s') |$s\ s'$. *envObs a* (*snd s*) = *envObs a* (*snd s'*) ∧ *fst s* = *fst s'* }

definition
spr-ii-simVal :: ($'a$, $'ePubAct$, $'es$, $'pPubAct$, $'ps$) *SPRstate* ⇒ $'p$ ⇒ *bool*
where *spr-ii-simVal* ≡ *envVal* ∘ *snd*

abbreviation
spr-ii-simMC ≡ *mkKripke* (*spr-ii-sim* ‘ *SPR.jkbpC*) *spr-ii-simRels spr-ii-simVal*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

The proofs that this simulation is adequate are similar to those in the previous section. We elide the details.

lemma *spr-ii-sim*: *sim SPR.MC spr-ii-simMC spr-ii-sim*⟨*proof*⟩
end

sublocale *FiniteBroadcastEnvironmentIndependentInit*
< *SPRii*: *SimIncrEnvironment jkbp envInit envAction envTrans envVal*
spr-jview envObs spr-jviewInit spr-jviewIncr
spr-ii-sim spr-ii-simRels spr-ii-simVal⟨*proof*⟩

8 Examples

We demonstrate the theory by using Isabelle's code generator to run it on two standard examples: the Robot from §1, and the classic muddy children puzzle.

8.1 The Robot

Recall the autonomous robot of §1: we are looking for an implementation of the KBP:

do
[] $\mathbf{K}_{\text{robot}}$ goal → Halt
[] $\neg\mathbf{K}_{\text{robot}}$ goal → Nothing
od

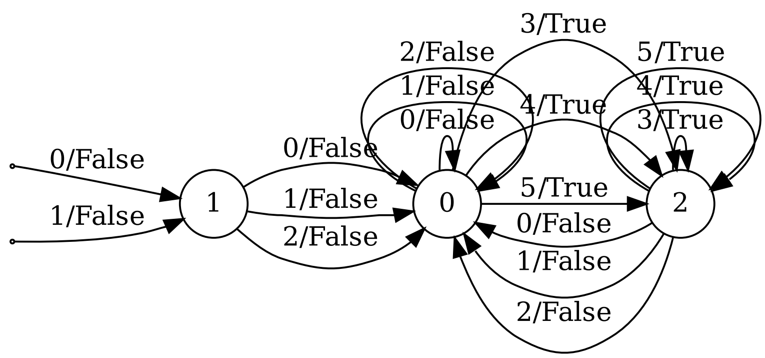


Figure 8: The implementation of the robot using the clock semantics.

in an environment where positions are identified with the natural numbers, the robot's sensor is within one of the position, and the proposition goal is true when the position is in $\{2, 3, 4\}$. The robot is initially at position zero, and the effect of its Halt action is to cause the robot to instantaneously stop at its current position. A later Nothing action may allow the environment to move the robot further to the right.

To obtain a finite environment, we truncate the number line at 5, which is intuitively sound for determinining the robot's behaviour due to the synchronous view, and the fact that if it reaches this rightmost position then it can never satisfy its objective. Running the Haskell code generated by Isabelle yields the automata shown in Figure 8 and Figure 9 for the clock and synchronous perfect recall semantics respectively. These have been minimised using Hopcroft's algorithm (Gries 1973).

The (inessential) labels on the states are an upper bound on the set of positions that the robot considers possible when it is in that state. Transitions are annotated with the observations yielded by the sensor. Double-circled states are those in which the robot performs the Halt action, the others Nothing. We observe that the synchronous perfect-recall view yields a "ratchet" protocol, i.e. if the robot learns that it is in the goal region then it halts for all time, and that it never overshoots the goal region. Conversely the clock semantics allows the robot to infinitely alternate its actions depending on the sensor reading. This is effectively the behaviour of the intuitive implementation that halts iff the sensor reads three or more.

We can also see that minimisation does not yield the smallest automata we could

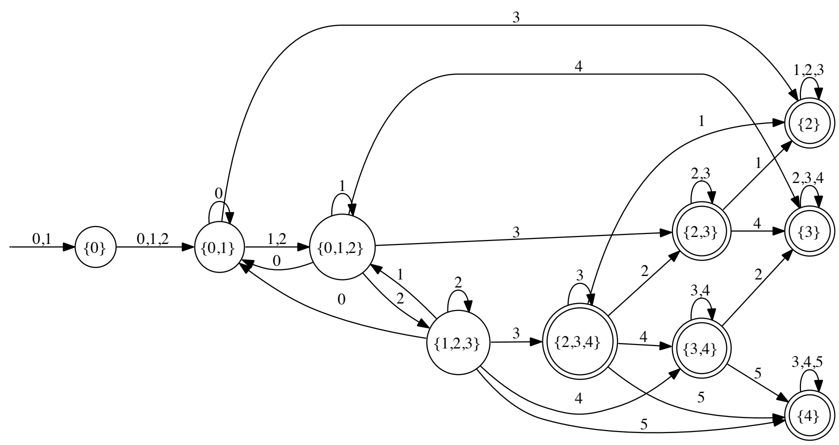


Figure 9: The implementation of the robot using the SPR semantics.

hope for; in particular there are a lot of redundant states where the prescribed behaviour is the same but the robot's state of knowledge different. This is because our implementations do not specify what happens on invalid observations, which we have modelled as errors instead of don't-cares, and these extraneous distinctions are preserved by bisimulation reduction. We discuss this further in §**??**.

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

8.2 The Muddy Children

Our first example of a multi-agent broadcast scenario is the classic muddy children puzzle, one of a class of such puzzles that exemplify non-obvious reasoning about mutual states of knowledge. It goes as follows (Fagin et al. 1995, §1.1, Example 7.2.5):

> N children are playing together, k of whom get mud on their foreheads. Each can see the mud on the others' foreheads but not their own.
>
> A parental figure appears and says "At least one of you has mud on your forehead.", expressing something already known to each of them if $k > 1$.
>
> The parental figure then asks "Does any of you know whether you have mud on your own forehead?" over and over.
>
> Assuming the children are perceptive, intelligent, truthful and they answer simultaneously, what will happen?

This puzzle relies essentially on *synchronous public broadcasts* making particular facts *common knowledge*, and that agents are capable of the requisite logical inference.

As the mother has complete knowledge of the situation, we integrate her behaviour into the environment. Each agent child_i reasons with the following KBP:

do

$[]\ \hat{\mathbf{K}}_{\text{child}_i}\text{muddy}_i \quad \rightarrow$ Say "I know if my forehead is muddy"

$[]\ \neg\hat{\mathbf{K}}_{\text{child}_i}\text{muddy}_i \quad \rightarrow$ Say nothing

od

where $\hat{\mathbf{K}}_a\varphi$ abbreviates $\mathbf{K}_a\varphi \vee \mathbf{K}_a\neg\varphi$.

As this protocol is deterministic, we use the SPR algorithm of §7.4.

The model records a child's initial observation of the mother's pronouncement and the muddiness of the other children in her initial private state, and these states are not changed by *envTrans*. The recurring common observation is all of the children's public responses to the mother's questions. Being able to distinguish these observations is crucial to making this a broadcast scenario.

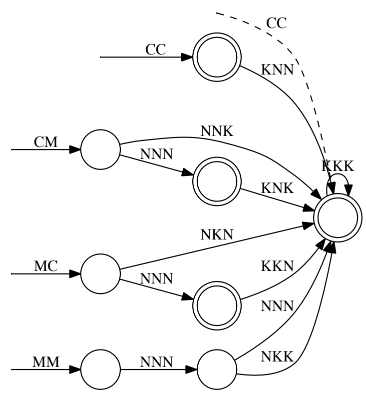


Figure 10: The protocol of child_0.

Running the algorithm for three children and minimising using Hopcroft's algorithm yields the automaton in Figure 10 for child_0. The initial transitions are labelled with the initial observation, i.e., the cleanliness "C" or muddiness "M" of the other two children. The dashed initial transition covers the case where everyone is clean; in the others the mother has announced that someone is dirty. Later transitions simply record the actions performed by each of the agents, where "K" is the first action in the above KBP, and "N" the second. Double-circled states are those in which child_0 knows whether she is muddy, and single-circled where she does not.

In essence the child counts the number of muddy foreheads she sees and waits that many rounds before announcing that she knows.

Note that a solution to this puzzle is beyond the reach of the clock semantics as it requires (in general) remembering the sequence of previous broadcasts of length proportional to the number of children. We discuss this further in §**??**.

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

9 Perspective and related work

The most challenging and time-consuming aspect of mechanising this theory was making definitions suitable for the code generator. For example, we could have used a locale to model the interface to the maps in §6.9, but as as the code generator presently does not cope with functions arising from locale interpretation, we are forced to say things at least twice if we try to use both features, as we implicitly did in §6.9. Whether it is more convenient or even necessary to use a record and predicate or a locale presently requires experimentation and guidance from experienced users.

As reflected by the traffic on the Isabelle mailing list, a common stumbling block when using the code generator is the treatment of sets. The existing libraries are insufficiently general: Florian Haftmann's *Cset* theory[1] does not readily support a choice operator, such as the one we used in §**??**. Even the heroics of the Isabelle Collections Framework Lammich and Lochbihler (2010) are insufficient as there equality on keys is structural (i.e., HOL equality), forcing us to either use a canonical representation (such as ordered distinct lists) or redo the relevant proofs with reified operations (equality, orderings, etc.). Neither of these is satisfying from the perspective of reuse.

Working with suitably general theories, e.g., using data refinement, is difficult as the simplifier is significantly less helpful for reasoning under abstract quotients, such as those in §6.9; what could typically be shown by equational rewriting now involves reasoning about existentials. For this reason we have only allowed some types to be refined; the representations of observations and system states are constant throughout our development, which may preclude some optimisations. The recent work of Kaliszyk and Urban Kaliszyk and Urban (2011) addresses these issues for concrete quotients, but not for the abstract ones that arise in this kind of top-down development.

As for the use of knowledge in formally reasoning about systems, this and similar semantics are under increasing scrutiny due to their relation to security properties. Despite the explosion in number of epistemic model checkers van Eijck and Orzan (2005); Gammie and van der Meyden (2004); Kacprzak et al. (2008); Lomuscio et al. (2009), finding implementations of knowledge-based programs remains a substantially manual affair Al-Bataineh and van der Meyden (2010). A refinement framework has also been developed Bickford et al. (2004); Engelhardt et al. (2000).

The theory presented here supports a more efficient implementation using symbolic techniques, ala MCK; recasting the operations of the SimEnvironment locale into boolean decision diagrams is straightforward. It is readily generalised to other synchronous views, as alluded to in §7.3, and adding a common knowledge modality, useful for talking about consensus (Fagin et al. 1995, Chapter 6), is routine. We hope that such an implementation will lead to more exploration of the KBP formalism.

[1] The theory *Cset* accompanies the Isabelle/HOL distribution.

10 Acknowledgements

Thanks to Kai Engelhardt for general discussions and for his autonomous robot graphic. Florian Haftmann provided much advice on using Isabelle/HOL's code generator and Andreas Lochbihler illuminated the darker corners of the locale mechanism. The implementation of Hopcroft's algorithm is due to Gerwin Klein. I am grateful to David Greenaway, Gerwin Klein, Toby Murray and Bernie Pope for their helpful comments.

This work was completed while I was employed by the L4.verified project at NICTA. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the IT Centre of Excellence program.

References

O. Al-Bataineh and R. van der Meyden. Epistemic model checking for knowledge-based program implementation: an application to anonymous broadcast. In *SecureComm*, 2010.

C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. M. Borwein and W. M. Farmer, editors, *MKM*, volume 4108 of *LNCS*. Springer, 2006. ISBN 3-540-37104-4.

S. Berghofer and M. Reiter. Formalizing the logic-automaton connection. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 147–163. Springer, 2009.

M. Bickford, R. L. Constable, J. Y. Halpern, and S. Petride. Knowledge-based synthesis of distributed systems using event structures. In F. Baader and A. Voronkov, editors, *LPAR*, volume 3452 of *LNCS*, pages 449–465. Springer, 2004. ISBN 3-540-25236-3.

B. Chellas. *Modal Logic: an introduction.* Cambridge University Press, 1980.

W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1998.

K. Engelhardt, R. van der Meyden, and Y. Moses. A program refinement framework supporting reasoning about knowledge and time. In Jerzy Tiuryn, editor, *FOSSACS*, volume 1784 of *LNCS*. Springer, March 2000.

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

R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4), 1997.

P. Gammie. KBPs. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. http://isa-afp.org/entries/KBPs.shtml, February 2011. Formal proof development.

P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *LNCS*. Springer, 2004. ISBN 3-540-22342-8.

D. Gries. Describing an Algorithm by Hopcroft. *Acta Informatica*, 2, 1973.

Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *FLOPS*, volume 6009 of *LNCS*. Springer, 2010.

J. Hintikka. *Knowledge and Belief: An Introduction to the Logic of Two Notions*. Cornell University Press, 1962.

M. Kacprzak, W. Nabialek, A. Niewiadomski, W. Penczek, A. Pólrola, M. Szreter, B. Wozna, and A. Zbrzezny. VerICS 2007 - a model checker for knowledge and real-time. *Fundamenta Informaticae*, 85(1-4), 2008.

C. Kaliszyk and C. Urban. Quotients revisited for Isabelle/HOL. In *SAC*. ACM, 2011.

P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *LNCS*. Springer, 2010. ISBN 978-3-642-14051-8.

W. Lenzen. Recent Work in Epistemic Logic. *Acta Philosophica Fennica*, 30(1), 1978.

A. Lomuscio, R. van der Meyden, and M. Ryan. Knowledge in multiagent systems: initial configurations and broadcast. *ACM Trans. Comput. Log.*, 1 (2):247–284, 2000.

A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *LNCS*. Springer, 2009. ISBN 978-3-642-02657-7.

T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4), 2009.

Y. Shoham and K. Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521899435.

C. Sternagel and R. Thiemann. Executable transitive closures of finite relations. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://isa-afp.org/entries/KBPs.shtml, March 2011. Formal proof development.

R. van der Meyden. Finite state implementations of knowledge-based programs. In *FTTCS*. Springer, 1996.

R. van der Meyden. Constructing Finite State Implementations of Knowledge-Based Programs with Perfect Recall. In *PRICAI '96: Proceedings from the Workshop on Intelligent Agent Systems, Theoretical and Practical Issues*, pages 135–151. Springer-Verlag, 1997.

D. J. N. van Eijck and S. M. Orzan. Modelling the epistemics of communication with functional programming. In *TFP*. Tallinn University, 2005.