Verified Synthesis of Knowledge-Based Programs in Finite Synchronous Environments

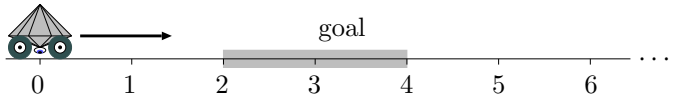
Peter Gammie[1,2]

[1] The Australian National University, Canberra ACT 0200, Australia
Peter.Gammie@anu.edu.au
WWW home page: http://peteg.org/
[2] National ICT Australia

Abstract. Knowledge-based programs (KBPs) are a formalism for directly relating agents' 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.

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

where "$\mathbf{K}_{\text{robot}}$ goal" intuitively denotes "the robot knows it is in the goal region" [8, Example 7.2.2]. We will make this precise in §2, 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 this 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 [4,7,8] that this is a useful level of abstraction at which to reason about distributed systems, and some kinds of multi-agent systems [21]. The downside 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 [10] or pencil-and-paper refinement [7]. 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 [18] practical.

The contributions of this work are as follows: §2 develops enough of the theory of KBPs in Isabelle/HOL [19] to support a formal proof of the possibility of their implementation by finite-state automata (§3). The later sections extend this development with a full top-down derivation of an original algorithm that constructs these implementations (§4) and two instances of it (§5 and §6), culminating in the mechanical synthesis of two standard examples from the literature: the aforementioned robot (§5.1) and the muddy children (§6.1).

We make judicious use of parametric polymorphism and Isabelle's locale mechanism [2] to establish and instantiate this theory in a top-down style. Isabelle's code generator [12] allows the algorithm developed here to be directly executed on the two examples. The complete development, available from the Archive of Formal Proofs [9], includes the full formal details of all claims made here.

In the following we adopt the Isabelle convention of prefixing fixed but arbitrary types with an apostrophe, such as $'a$, and suffixing type constructors as in $'a$ list. Other non-standard syntax will be explained as it arises.

2 Semantics of Knowledge-Based Programs

We use what is now a standard account of the multi-agent (multi-modal) propositional logic of knowledge [5,8]. The language of the guards is propositional, augmented by one knowledge modality per agent and parameterised by a type $'p$ of propositions and $'a$ of agents:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{K}_a\ \varphi$$

Formulas are interpreted with respect to a *Kripke structure*, which consists of a set of worlds of type $'w$, an equivalence relation $\sim_a$ for each agent a over these worlds, and a way of evaluating propositions at each world; these are collected in a record of type $('a, 'p, 'w)$ KripkeStructure. We define satisfaction of a formula φ at a world w in structure M as follows:

$$\begin{array}{lll}
M, w \models p & \text{iff} & p \text{ is true at } w \text{ in } M \\
M, w \models \neg\varphi & \text{iff} & M, w \models \varphi \text{ is false} \\
M, w \models \varphi \wedge \psi & \text{iff} & M, w \models \varphi \text{ and } M, w \models \psi \\
M, w \models \mathbf{K}_a\ \varphi & \text{iff} & M, w' \models \varphi \text{ for all worlds } w' \text{ where } w \sim_a w' \text{ in } M
\end{array}$$

Intuitively $w \sim_a w'$ if a cannot distinguish between worlds w and w'; the final clause expresses the idea that an agent knows ψ iff ψ is true at all worlds she considers possible (relative to world w)[3]. This semantics supports nested modal operators, so, for example, "the sender does not know that the receiver knows the bit that was sent" can be expressed.

We represent a *knowledge-based program* (KBP) of type $('a, 'p, 'aAct)$ KBP as a list of records with fields guard and action, where the guards are knowledge formulas and the actions elements of the $'aAct$ type, and expect there to be one per agent. Lists are used here and elsewhere to ease the generation of code (see §5 and §7). The function set maps a list to the set of its elements.

Note that the robot of §1 cannot directly determine its exact position because of the noise in its sensor, which means that we cannot allow arbitrary formulas as guards. However an agent a *can* evaluate formulas of the form $\mathbf{K}_a\psi$ that depend only on the equivalence class of worlds a considers possible. That φ is a boolean combination of such formulas is denoted by subjective a φ.

We model the agents' interactions using a *finite environment*, following van der Meyden [18], which consist of a finite type $'s$ of states, a set *envInit* of initial states, a function *envVal* that evaluates propositions at each state, and a projection *envObs* that captures how each agent instantaneously observes these states. The system evolves using the transition function *envTrans*, which incorporates the environment's non-deterministic choice of action *envAction* and those of the agents' KBPs into a global state change. We collect these into an Isabelle locale:

```
locale Environment =
  fixes jkbp :: 'a ⇒ ('a, 'p, 'aAct) KBP
    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
  assumes subj: ∀ a gc. gc ∈ set (jkbp a) ⟶ subjective a (guard gc)
```

A locale defines a scope where the desired types, variables and assumptions are fixed and can be freely appealed to. Later we can instantiate these in various ways (see §4) and also extend the locale (see §3.1).

In the Environment locale we compute the actions enabled at world w in an arbitrary Kripke structure M for each agent using a list comprehension:

```
definition jAction :: ('a, 'p, 'w) KripkeStructure ⇒ 'w ⇒ 'a ⇒ 'aAct list where
  jAction M w a ≡ [ action gc. gc ← jkbp a, (M, w ⊨ guard gc) ]
```

This function composes with *envTrans* provided we can find a suitable Kripke structure and world. With the notional mutual dependency between knowledge and action of §1 in mind, this structure should be based on the set of traces

[3] As one would expect there has been extensive debate over the properties of knowledge; the reader is encouraged to consult [8, Chapter 2]. Also their Chapter 7 presents a more general (but non-algorithmic) account of KBPs at a less harried pace.

generated by *jkbp* in this particular environment, i.e., the very thing we are in the process of defining. As with all fixpoints there may be zero, one or many solutions; the following construction considers a broadly-applicable special case for which unique solutions exist.

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 $t \rightsquigarrow s$, equipped with tFirst, tLast, tLength and tMap functions.

Our construction begins by deriving a Kripke structure from an arbitrary set of traces T. The equivalence relation on these traces can be defined in a variety of ways [8,18]; here we derive the relation from the *synchronous perfect-recall (SPR) view*, which records all observations made by an agent:

```
definition spr-jview :: 'a ⇒ 's Trace ⇒ 'obs Trace where
  spr-jview a ≡ tMap (envObs a)
```

The Kripke structure mkM T relates all traces that have the same SPR view, and evaluates propositions at the final state of the trace, i.e., *envVal* ∘ tLast. In general we apply the adjective "synchronous" to relations that "tell the time" by distinguishing all traces of distinct lengths.

Using this structure we construct the sequence of *temporal slices* that arises from interpreting *jkbp* with respect to T by recursion over the time:

```
fun jkbpTn :: nat ⇒ 's Trace set ⇒ 's Trace set where
  jkbpT0 T       = { tInit s |s. s ∈ set envInit }
| jkbpTSuc n T = { t ⇝ envTrans eact aact (tLast t) |t eact aact.
                     t ∈ jkbpTn T ∧ eact ∈ set (envAction (tLast t))
                   ∧ (∀ a. aact a ∈ set (jAction (mkM T) t a)) }
```

We define jkbpT T to be $\bigcup n$ jkbpT$_n$ T. This gives us a closure condition on sets of traces T: we say that T *represents jkbp* if it is equal to jkbpT T. Exploiting the synchrony of the SPR view, we can inductively construct traces of length $n+1$ by interpreting *jkbp* with respect to all those of length n:

```
fun jkbpCn :: nat ⇒ 's Trace set where
  jkbpC0       = { tInit s |s. s ∈ set envInit }
| jkbpCSuc n = { t ⇝ envTrans eact aact (tLast t) |t eact aact.
                     t ∈ jkbpCn ∧ eact ∈ set (envAction (tLast t))
                   ∧ (∀ a. aact a ∈ set (jAction (mkM jkbpCn) t a)) }
```

We define mkMC$_n$ to be mkM jkbpC$_n$, and jkbpC to be $\bigcup n$. jkbpC$_n$ with corresponding Kripke structure mkMC.

We show that jAction mkMC t = jAction mkMC$_n$ t for $t \in$ jkbpC$_n$, i.e., that the relevant temporal slice suffices for computing jAction, by appealing to a multi-modal generalisation of the *generated model property* [5, §3.4]. This asserts that the truth of a formula at a world w depends only on the worlds reachable from w in zero or more steps, using any of the agents' accessibility relations at each step. We then establish that jkbpT$_n$ jkbpC = jkbpC$_n$ by induction on n, implying that jkbpC represents *jkbp* in the environment of interest. Uniqueness follows by a similar argument, and so:

Theorem 1. *The set* jkbpC *canonically represents jkbp.*

This is a specialisation of [8, Theorem 7.2.4].

3 Automata for KBPs

We now shift our attention to the problem of synthesising standard finite-state automata that *implement jkbp*. This section summarises the work of van der Meyden [18]. In §4 we will see how these are computed.

The essence of these constructions is to represent an agent's state of knowledge by the state of an automaton (of type *'ps*), also termed a *protocol*. This state evolves in response to the agent's observations of the system using *envObs*, and is deterministic as it must encompass the maximal uncertainty she has about the system. Our implementations take the form of Moore machines, which we represent using a record:

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

Transitions are labelled by observations, and states with the set of actions enabled by *jkbp*. The initialising function pInit maps an initial observation to an initial protocol state. A joint protocol *jp* is a mapping from agents to protocols. The term runJP *jp t* runs *jp* on a trace *t* in the standard manner, yielding a function from agents to protocol states. Similarly actJP *jp t* denotes the joint action of *jp* on trace *t*, i.e., $\lambda a.$ pAct (*jp a*) (runJP *jp t a*).

That a joint protocol *jp implements jkbp* is to say that *jp* and *jkbp* yield identical joint actions when run on any canonical trace $t \in$ jkbpC. To garner some intuition about the structure of such implementations, our first automata construction explicitly represents the partition of jkbpC induced by spr-jview, yielding an infinite-state joint protocol:

definition mkAuto :: *'a* ⇒ (*'obs*, *'aAct*, *'s* Trace set) Protocol **where**
  mkAuto *a* ≡ (| pInit = λ*obs*. { *t* ∈ jkbpC . spr-jview *a t* = tInit *obs* },
    pTrans = λ*obs ps*. { *t* |*t t'*. *t* ∈ jkbpC ∧ *t'* ∈ ps
      ∧ spr-jview *a t* = spr-jview *a t'* ⇝ *obs* },
    pAct = λ*ps*. jAction mkMC (SOME *t*. *t* ∈ ps) *a* |)

abbreviation equiv-class *a tobs* ≡ { *t* ∈ jkbpC . spr-jview *a t* = *tobs* }

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

Running mkAuto on a trace $t \in$ jkbpC yields the equivalence class of *t* for agent *a*, equiv-class *a* (spr-jview *a t*), and as pAct clearly prescribes the expected actions for subjective formulas, we have:

Theorem 2. mkAuto *implements jkbp in the given environment.*

3.1 A sufficient condition for finite-state implementations

van der Meyden showed that the existence of a *simulation* from mkMC to a finite structure is sufficient for there to be a finite-state implementation of *jkbp* [18, Theorem 2]. We say that a function f, mapping the worlds of Kripke structure M to those of M' is a simulation if it has the following properties:

- Propositions evaluate identically at $u \in \mathsf{worlds}\ M$ and $f\ u \in \mathsf{worlds}\ M'$;
- If two worlds u and v are related in M for agent a, then $f\ u$ and $f\ v$ are also related in M' for agent a; and
- If two worlds $f\ u$ and v' are related in M' for agent a, then there exists a world $v \in \mathsf{worlds}\ M$ such that $f\ v = v'$ and u and v are related in M for a.

From these we have $M,\ u \models \varphi$ iff $M',\ f\ u \models \varphi$ by straightforward structural induction on φ [5, §3.4, Ex. 3.60]. This result lifts through jAction and hence jkbpC_n. The promised finite-state protocol simulates the states of mkAuto.

4 An effective construction

The remaining algorithmic obstruction in mkAuto is the appeal to the infinite set of canonical traces jkbpC. While we could incrementally maintain the temporal slices of traces jkbpC_n, ideally the simulated equivalence classes would directly support the necessary operations. We therefore optimistically extend van der Meyden's construction by axiomatising these functions in the SimEnvironment locale of Figure 1, and making the following definition:

```
definition mkAutoSim :: 'a ⇒ ('obs, 'aAct, 'rep) Protocol where
  mkAutoSim a ≡
    (| pInit = simInit a,
       pTrans = λobs ec. (SOME ec'. ec' ∈ set (simTrans a ec) ∧ simObs a ec' = obs),
       pAct = λec. simAction ec a |)
```

The specification of these functions is complicated by the use of *simAbs* to incorporate some data refinement [20], which allows the type *'rep* of representations of simulated equivalence classes (with type *'ss* set) to depend on the entire context. This is necessary because finite-state implementations do not always exist with respect to the SPR view [18, Theorem 5], and so we must treat special cases that may use quite different representations. If we want a once-and-for-all-time proof of correctness for the algorithm, we need to make this allowance here.

A routine induction on $t \in \mathsf{jkbpC}$ shows that mkAutoSim faithfully maintains a representation of the simulated equivalence class of t, which in combination with the locale assumption *simAction* gives us:

Theorem 3. mkAutoSim *implements jkbp in the given environment.*

Note that we are effectively asking *simTrans* to compute the actions of *jkbp* for all agents using only a representation of a simulated equivalence class for the

```
locale SimEnvironment =
      Environment jkbp envInit envAction envTrans envVal envObs
   for jkbp :: 'a ⇒ ('a, 'p, 'aAct) KBP
   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

   — Simulation operations
+ fixes simf :: 's Trace ⇒ 'ss :: finite
   and simRels :: 'a ⇒ ('ss × 'ss) set
   and simVal :: 'ss ⇒ 'p ⇒ bool

   — Adequacy of representations
   and simAbs :: 'rep ⇒ 'ss set

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

 assumes simf: sim mkMC (mkKripke (simf ‘ jkbpC) simRels simVal) simf
   and simInit:
       ∀ a obs. obs ∈ envObs a ‘ set envInit
           ⟶ simAbs (simInit a obs) = simf ‘ equiv-class a (tInit obs)
   and simObs:
       ∀ a ec t. t ∈ jkbpC ∧ simAbs ec = simf ‘ equiv-class a (spr-jview a t)
           ⟶ simObs a ec = envObs a (tLast t)
   and simAction:
       ∀ a ec t. t ∈ jkbpC ∧ simAbs ec = simf ‘ equiv-class a (spr-jview a t)
           ⟶ set (simAction ec a) = set (jAction mkMC t a)
   and simTrans:
       ∀ a ec t. t ∈ jkbpC ∧ simAbs ec = simf ‘ equiv-class a (spr-jview a t)
           ⟶ simAbs ‘ set (simTrans a ec)
             = { simf ‘ equiv-class a (spr-jview a (t' ⇝ s)) |t' s.
                 t' ⇝ s ∈ jkbpC ∧ spr-jview a t' = spr-jview a t}
```

Fig. 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. The function mkKripke constructs a Kripke structure from its three components. By *sim M M′ f* we assert that f is a simulation from M to M'.

particular agent a. This contrasts with our initial automata construction mkAuto (§3) that appealed to jkbpC for this purpose. We will see in §5 and §6 that our concrete simulations do retain sufficient information.

4.1 A synthesis algorithm

We now show how automata that implement *jkbp* can be constructed using the operations specified in SimEnvironment. Taking care with the definitions allows us to extract an executable version via Isabelle/HOL's code generator [12].

We represent the automaton under construction by a pair of maps, one for actions, mapping representations to lists of agent actions, and the other for the transition function, mapping representations and observations to representations. These maps are represented by the types $'ma$ and $'mt$ respectively, with operations collected in *aOps* and *tOps*. These MapOps records contain *empty*, *lookup* and *update* functions, specified in the standard way with the extra condition that they respect *simAbs* on the domains of interest.

abbreviation jkbpSEC $\equiv \bigcup a.\ \{\ simf\ `\ \text{equiv-class}\ a\ (\text{spr-jview}\ a\ t)\ |t.\ t \in \text{jkbpC}\ \}$

locale Algorithm =
  SimEnvironment *jkbp envInit envAction envTrans envVal envObs*
                 simf simRels simVal simAbs simObs simInit simTrans simAction
    for *jkbp* :: $'a \Rightarrow ('a, 'p, 'aAct)$ KBP
— ... as for SimEnvironment ...

\+ **fixes** *aOps* :: $('ma, 'rep, 'aAct\ \text{list})$ MapOps
    and *tOps* :: $('mt, 'rep \times 'obs, 'rep)$ MapOps
  assumes *aOps*: MapOps *simAbs* jkbpSEC *aOps*
      and *tOps*: MapOps $(\lambda k.\ (simAbs\ (\text{fst}\ k), \text{snd}\ k))$ (jkbpSEC × UNIV) *tOps*

UNIV is the set of all elements of a type. The repetition of type signatures in these extended locales is tiresome but necessary to bring the type variables into scope. As we construct one automaton per agent, we introduce another locale:

locale AlgorithmForAgent = Algorithm — ... + **fixes** $a :: 'a$

The algorithm traverses the representations of simulated equivalence classes of jkbpC reachable via *simTrans*. We use the executable depth-first search (DFS) theory due to Berghofer and Krauss [3], mildly generalised to support data refinement. The DFS locale requires the following definitions, shown in Figure 2:

- an initial automaton *k-empt*;
- the initial frontier *frontier-init* is the partition of the set of initial states under *envObs a*;
- the successor function *k-succs* is exactly *simTrans a*;
- for each reachable state the action and transition maps are updated with *k-ins*; and
- the visited predicate *k-memb* uses the domain of the *aOps* map.

```
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))

definition alg-dfs aOps tOps frontier-init simObs simTrans simAction ≡
    let k-empt = (empty aOps, empty tOps);
        k-memb = (λs A. isSome (lookup aOps (fst A) s));
        k-succs = simTrans;
        acts-update = (λec A. update aOps ec (simAction ec) (fst A));
        trans-update = (λec ec' at. update tOps (ec, simObs ec') ec' at);
       k-ins = (λec A. (acts-update ec A, foldr (trans-update ec) (k-succs ec) (snd A)))
     in gen-dfs k-succs k-ins k-memb k-empt frontier-init
```

Fig. 2. The algorithm. The symbol @ denotes list concatenation.

Instantiating the DFS locale is straightforward:

```
sublocale AlgorithmForAgent
      < KBPAlg!: DFS k-succs k-is-node k-invariant k-ins k-memb k-empt simAbs
```

This *conditional interpretation* is a common pattern in these proofs: it says that we can discharge the requirements of the DFS locale while appealing to the AlgorithmForAgent context, i.e., the constraints in the SimEnvironment locale and those for our two maps. The resulting definitions and lemmas appear in the AlgorithmForAgent context with prefix KBPAlg.

Our invariant over the reachable state space is that the automaton under construction is well-defined with respect to the *simAction* and *simTrans* functions. The DFS theory shows that the traversal visits all states reachable from the initial frontier, and we show that the set of reachable equivalence classes coincides with the partition of jkbpC under spr-jview a, modulo simulation and representation. Thus the algorithm produces an implementation of *jkbp* for agent a.

We trivially generalise the fixed-agent lemmas to the multi-agent locale:

```
sublocale Algorithm < KBP!: AlgorithmForAgent — ... a for a
```

The output of the DFS is converted into a protocol using *simInit* and *lookup* on the maps; call this mkAutoAlg. We show in the Algorithm context that mkAutoAlg prescribes the same actions as mkAutoSim for all $t \in$ jkbpC, and therefore:

Theorem 4. mkAutoAlg *is a finite-state implementation of jkbp in the given environment.*

The following sections show that this theory is sound and effective by fulfilling the promises made in the SimEnvironment locale of Figure 1: §5 demonstrates a simulation and representation for the single-agent case, which suffices for finding an implementation of the robot's KBP from §1; §6 treats a multi-agent scenario general enough to handle the classic muddy children puzzle.

5 Perfect Recall for a Single Agent

Our first simulation treats the simple case of a single agent executing an arbitrary KBP in an arbitrary environment, such as the robot of §1. We work in the SingleAgentEnvironment locale, which is the Environment locale augmented with a variable *agent* denoting the element of the $'a$ type. We seek a finite space that simulates mkMC; as we later show, satisfaction at $t \in$ jkbpC is a function of the set of final states of the traces that *agent* considers possible, i.e., of:

definition spr-jview-abs :: $'s$ Trace $\Rightarrow$ $'s$ set **where**
  spr-jview-abs $t \equiv$ tLast ‘ equiv-class *agent* (spr-jview *agent* t)

To evaluate propositions we include the final state of t in our simulation:

definition *spr-sim-single* :: $'s$ Trace $\Rightarrow$ $'s$ set $\times$ $'s$ **where**
  spr-sim-single $t \equiv$ (spr-jview-abs t, tLast t)

In the structure mkMCS, $(U,\ u) \sim_a (V,\ v)$ iff $U = V$ and *envObs agent* $u =$ *envObs agent* v, and propositions are evaluated with *envVal* $\circ$ snd. Then:

Theorem 5. mkMCS *simulates* mkMC.

An optimisation is to identify related worlds, recognising that the agent behaves the same at all of these. This quotient is isomorphic to spr-jview-abs ‘ jkbpC, and so the algorithm effectively simplifies to the familiar subset construction for determinising finite-state automata.

We now address algorithmic issues. As the representations of equivalence classes 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 environment states $'s$ must belong to the class linorder of linearly-ordered types.

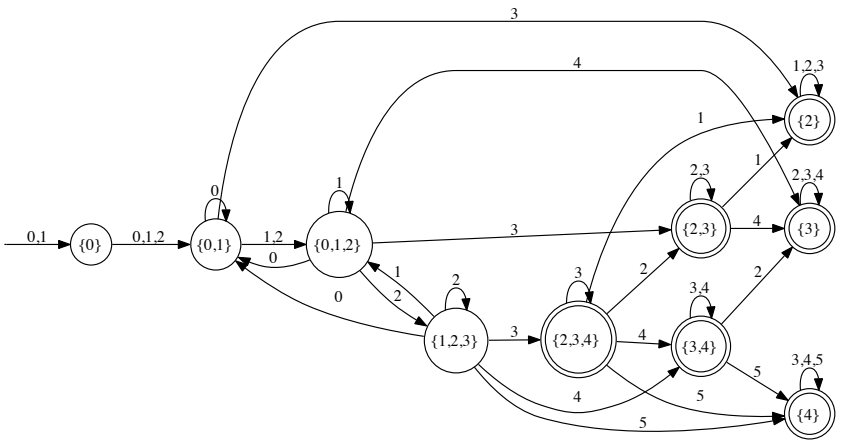
For a set of states X, we define a function eval $X\ \varphi$ that computes the subset of X where φ holds. The only interesting case is that for knowledge: eval X $(\mathbf{K}_a\ \psi)$ evaluates to X if eval $X\ \psi = X$, and $\emptyset$ otherwise. This corresponds to standard satisfaction when X represents spr-jview-abs t for some $t \in$ jkbpC. The requisite *simObs*, *simInit*, *simAction* and *simTrans* functions are routine, as is instantiating the Algorithm locale. Thus we have an algorithm for all single-agent scenarios that satisfy the Environment locale.

A similar simulation can be used to show that there always exist implementations with respect to the multi-agent *clock view* [18, Theorem 4], the weakest synchronous view that considers only the time and most-recent observation.

5.1 The Robot

We now feed the algorithm, the simulated operations of the previous section and a model of the autonomous robot of §1 to the Isabelle/HOL code generator. To obtain a finite environment we truncate the number line at 5. This is intuitively

sound for the purposes of determinining the robot's behaviour due to the synchronous view and the observation that if it reaches this rightmost position then it can never satisfy its objective. Running the resulting Haskell code yields this automaton, which we have minimised using Hopcroft's algorithm [11]:



The inessential labels on the states indicate the robot's knowledge about its position, and those on the transitions are the observations yielded by the sensor. Double-circled states are those in which the robot performs the Halt action, the others Nothing. We can see that 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. We can also see that traditional minimisation does not yield the smallest automaton we could hope for. This is because the algorithm does not specify what happens on invalid observations, which are modelled as errors instead of don't-cares.

6 Perfect Recall in Broadcast Environments with Deterministic Protocols

We now consider a more involved multi-agent case, where deterministic JKBPs operate in non-deterministic environments and communicate via *broadcast*. It is well known [8, Chapter 6] 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.

The broadcast is modelled as a *common observation* of the environment's state that is included in all agents' observations. We also allow the agents to maintain entirely disjoint private states of type *'as*. This is expressed in the locale in Figure 3, where the constraints on *envTrans* and *envObs* enforce the disjointness.

```
record ('a, 'es, 'as) BEState =
  es :: 'es
  ps :: ('a × 'as) odlist — Associates an agent with her private state.

locale DetBroadcastEnvironment =
  Environment jkbp envInit envAction envTrans envVal envObs
    for jkbp :: 'a ⇒ ('a :: {finite, linorder}, 'p, 'aAct) KBP
    and envInit :: ('a, 'es :: {finite, linorder}, 'as :: {finite, linorder}) BEState list
    and envAction :: ('a, 'es, 'as) BEState ⇒ 'eAct list
    and envTrans :: 'eAct ⇒ ('a ⇒ 'aAct)
                    ⇒ ('a, 'es, 'as) BEState ⇒ ('a, 'es, 'as) BEState
    and envVal :: ('a, 'es, 'as) BEState ⇒ 'p ⇒ bool
    and envObs :: 'a ⇒ ('a, 'es, 'as) BEState ⇒ ('cobs × 'as option)

+ fixes agents :: 'a odlist
    and envObsC :: 'es ⇒ 'cobs
  defines envObs a s ≡ (envObsC (es s), ODList.lookup (ps s) a)
  assumes agents: ODList.toSet agents = UNIV
      and envTrans: ∀ s s' a eact eact' aact aact'.
            ODList.lookup (ps s) a = ODList.lookup (ps s') a ∧ aact a = aact' a
             ⟶ ODList.lookup (ps (envTrans eact aact s)) a
               = ODList.lookup (ps (envTrans eact' aact' s')) a
      and jkbpDet: ∀ a. ∀ t ∈ jkbpC. length (jAction mkMC t a) ≤ 1
```

Fig. 3. The DetBroadcastEnvironment locale.

Similarly to §5, 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 $t \in \mathsf{jkbpC}$ 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)
```

Unlike the single-agent case of §5, 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 t ≡ {(tFirst t', tLast t') |t'. t' ∈ jkbpC ∧ tObsC t' = tObsC t}
```

We can predict an agent's final private state on $t' \in \mathsf{jkbpC}$ where $\mathsf{tObsC}\ t' = \mathsf{tObsC}\ t$ from the agent's private state in $\mathsf{tFirst}\ t'$ and $\mathsf{tObsC\text{-}abs}\ t$ due to the determinacy requirement *jkbpDet* and the constraint *envTrans*. Thus the agent's state of knowledge on t is captured by the following simulation:

```
record ('a, 'es, 'as) SPRstate =
  sprFst :: ('a, 'es, 'as) BEState
  sprLst :: ('a, 'es, 'as) BEState
  sprCRel :: (('a, 'es, 'as) BEState × ('a, 'es, 'as) BEState) set
```

definition spr-sim :: ('a, 'es, 'as) BEState Trace ⇒ ('a, 'es, 'as) SPRstate **where**
spr-sim $t \equiv (\!|$ sprFst = tFirst t, sprLst = tLast t, sprCRel = tObsC-abs t $|\!)$

We build a Kripke structure mkMCS of simulated traces by relating worlds U and V for agent a where *envObs* a (sprFst U) = *envObs* a (sprFst V) and *envObs* a (sprLst U) = *envObs* a (sprLst V), and sprCRel U = sprCRel V. Propositions are evaluated by *envVal* ∘ sprLst. We have:

Theorem 6. mkMCS *simulates* mkMC.

Establishing this is routine, where the final simulation property follows from our ability to predict agents' private states on canonical traces as mentioned above.

As in §5, we can factor out the common parts of these equivalence classes to yield a denser representation that uses a pair of relations and thus a four-level trie. We omit the tedious details of placating the SimEnvironment locale.

van der Meyden [18, §7] used this simulation to obtain finite-state implementations for non-deterministic KBPs under the extra assumptions that the parts of the agents' actions that influence *envAction* are broadcast and recorded in the system states, and that *envAction* be oblivious to the agents' private states. Therefore those results do not subsume the ones presented here, just as those of this section do not subsume those of §5.

6.1 The Muddy Children

The classic muddy children puzzle [8, §1.1, Example 7.2.5] is an example of a multi-agent broadcast scenario that exemplifies non-obvious reasoning about mutual states of knowledge. Briefly, there are $N > 2$ children playing together, some of whom get mud on their foreheads. Each can see the others' foreheads but not their own. A mother observes the situation and either says that everyone is clean, or says that someone is dirty. She then asks "Do 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?

Each agent child_i reasons with the following KBP:

do
[] $\hat{\mathbf{K}}_{\text{child}_i}\text{muddy}_i$ → Say "I know if my forehead is muddy"
[] $\neg\hat{\mathbf{K}}_{\text{child}_i}\text{muddy}_i$ → Say nothing
od

where $\hat{\mathbf{K}}_a\varphi$ abbreviates $\mathbf{K}_a\varphi \vee \mathbf{K}_a\neg\varphi$. As the mother has complete knowledge of the situation, we integrate her behaviour into the environment.

In general the determinism of a KBP is a function of the environment, and may be difficult to establish. In this case and many others, however, determinism is

syntactically manifest as the guards are logically disjoint, independently of the knowledge subformulas.

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 preserved by *envTrans*. The recurring common observation is all of the children's public responses to the mother's questions. Being able to distinguish these types of observations is crucial to making this a broadcast scenario.

Running the algorithm for three children and minimising yields the automaton in Figure 4 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.

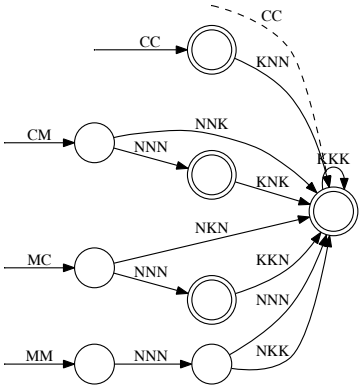


Fig. 4. The protocol of child_0.

To the best of our knowledge this is the first time that an implementation of the muddy children has been automatically synthesised.

7 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 §4, 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 Figure 2. Whether it is more convenient or even necessary to use a record and predicate or a locale presently requires experimentation and perhaps 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[4] does not readily support a choice operator, such as the one we used in §3. Even the heroics of the Isabelle

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

Collections Framework [15] 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 Figure 1; 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 [14] 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 [6,10,13,16], finding implementations of knowledge-based programs remains a substantially manual affair [1]. van der Meyden also proposed a complete semi-algorithm for KBP synthesis [17]. A refinement framework has been developed [4,7].

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 §5, and adding a common knowledge modality, useful for talking about consensus [8, Chapter 6], is routine. We hope that such an implementation will lead to more exploration of the KBP formalism.

Acknowledgments. 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, Bernie Pope and the anonymous reviewers for their helpful comments on a draft of this paper.

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

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

2. Ballarin, C.: Interpretation of locales in Isabelle: Theories and proof contexts. In: Borwein, J.M., Farmer, W.M. (eds.) MKM. LNCS, vol. 4108. Springer (2006)
3. Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. http://afp.sf.net/entries/Presburger-Automata.shtml (Dec 2009), Formal proof development
4. Bickford, M., Constable, R.L., Halpern, J.Y., Petride, S.: Knowledge-based synthesis of distributed systems using event structures. In: Baader, F., Voronkov, A. (eds.) LPAR. LNCS, vol. 3452. Springer (2004)
5. Chellas, B.: Modal Logic: an introduction. Cambridge University Press (1980)
6. van Eijck, D.J.N., Orzan, S.M.: Modelling the epistemics of communication with functional programming. In: TFP. Tallinn University (2005)
7. Engelhardt, K., van der Meyden, R., Moses, Y.: A program refinement framework supporting reasoning about knowledge and time. In: Tiuryn, J. (ed.) FOSSACS. LNCS, vol. 1784. Springer (Mar 2000)
8. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. The MIT Press (1995)
9. Gammie, P.: KBPs. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. http://afp.sf.net/entries/KBPs.shtml (May 2011), Formal proof development
10. Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Alur, R., Peled, D. (eds.) CAV. LNCS, vol. 3114. Springer (2004)
11. Gries, D.: Describing an Algorithm by Hopcroft. Acta Informatica 2 (1973)
12. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS. LNCS, vol. 6009. Springer (2010)
13. Kacprzak, M., Nabialek, W., Niewiadomski, A., Penczek, W., Pólrola, A., Szreter, M., Wozna, B., Zbrzezny, A.: VerICS 2007 - a model checker for knowledge and real-time. Fundamenta Informaticae 85(1-4) (2008)
14. Kaliszyk, C., Urban, C.: Quotients revisited for Isabelle/HOL. In: SAC. ACM (2011)
15. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Kaufmann, M., Paulson, L.C. (eds.) ITP. LNCS, vol. 6172. Springer (2010)
16. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multi-agent systems. In: Bouajjani, A., Maler, O. (eds.) CAV. LNCS, vol. 5643. Springer (2009)
17. van der Meyden, R.: Constructing finite state implementations of knowledge-based programs with perfect recall. In: Cavedon, L., Rao, A.S., Wobcke, W. (eds.) PRICAI Workshop on Intelligent Agent Systems. LNCS, vol. 1209. Springer (1996)
18. van der Meyden, R.: Finite state implementations of knowledge-based programs. In: Chandru, V., Vinay, V. (eds.) FSTTCS. LNCS, vol. 1180. Springer (1996)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
20. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge University Press (1998)
21. Shoham, Y., Leyton-Brown, K.: Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations. Cambridge University Press, New York, NY, USA (2008)