Proceedings, The Thirteenth AAAI Conference on
Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)

Practical Specification of Belief Manipulation in Games

Markus Eger, Chris Martens
meger@ncsu.edu, crmarten@ncsu.edu
Principles of Expressive Machines Lab
NC State University
Raleigh, NC, USA

Abstract

Actions that affect knowledge asymmetrically between agents occur in numerous domains, from card games such as poker to the secure transmission of information. Applications in such domains often depend on reflection over knowledge, including what an agent knows about what other agents know. We are interested in enabling formal specification of these systems which may be used for executable prototyping as well as verification and other formal reasoning. Dynamic Epistemic Logic provides a formal basis for such reasoning, but is often prohibitively cumbersome to use in practice. We present an implementation and macro system called Ostari, backed by a particular flavor of Dynamic Epistemic Logic, which allows us to scale the ideas to more realistic problems. We demonstrate how actions that manipulate agents' beliefs can be written concisely and how this capability can be applied to modeling a popular card game by utilizing our system's ability to execute action sequences, answer queries about knowledge states, and find action sequences to satisfy a particular goal.

Introduction

Epistemic logic, the logic of knowledge and belief (Hintikka 1962), is useful in distributed reasoning to be able to represent and reason about disparities in the knowledge of different agents. For example, in many card games, players have a hand of cards that is only visible to themselves. Agents playing these games can benefit from having a model of what their opponents know, as illustrated in figure 1, by using this information to plan their actions. Several games are even built around the notion of manipulating and reasoning about other player's beliefs. For example, Hanabi (Bauza 2011) is a cooperative card game in which players cannot see the cards in their own hand and have to give hints to their cooperators to tell them about the cards in their hands. Playing this game well requires reasoning about what the cooperators already know and how they will interpret information that is given to them. There are also social deduction games, such as The Resistance (Eskridge 2009), One Night Ultimate Werewolf (Alspach and Okui 2014) and many others, in which players are secretly assigned to different factions and have to deduce which other players are in the opposing faction or factions.

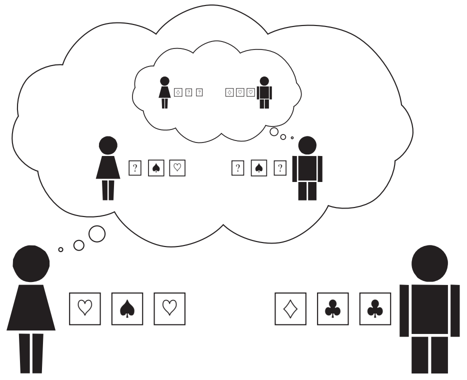

Figure 1: Theory of mind: Having a mental model of other agents' mental models

For computational agents to perform this reasoning, they need to be able to reason about other agents' knowledge (also known as *theory of mind*) including how it changes over time. This kind of reasoning, captured formally by *Dynamic Epistemic Logic* (DEL) (Van Ditmarsch, van Der Hoek, and Kooi 2007), is arguably a fundamental challenge of AI: cognitive scientists have linked it to social intelligence in humans (Baron-Cohen 1997; Wimmer and Perner 1983), a phenomenon that AI researchers have seen as critical for building collaborative systems from autonomous agents (Castelfranchi 1998; Dignum, Prada, and Hofstede 2014). While DEL provides an expressive theoretical foundation, using it for any non-trivial problem can be quite cumbersome, because the basic unit of reasoning is a single literal. For example, in a card game, expressing that a player gains knowledge of a particular card in their hand requires enumerating all possible values that card can have and joining them with the appropriate operator.

In this paper, we present our system Ostari, which provides an implementation of a particular flavor of DEL, presented by Baltag (2002). Our contribution is this implementation and the included macro system that can be used to describe epistemic actions in a concise way and a complete run-time environment that allows users to define and execute actions, as well as query the epistemic state, or define a goal for which the system will find an action sequence. We will also demonstrate how this implementation can be used to express actions needed for Hanabi and briefly discuss other applications.

We will now survey related work, describe Ostari and how the macros are compiled to Baltag, and then describe our Hanabi case study that demonstrates our system's ability to execute actions and answer queries about epistemic actions and knowledge propagation.

Related Work

Modeling agent knowledge and how it changes over time has been of interest to researchers for a number of applications, such as teamwork in networks without central coordination (Roth, Simmons, and Veloso 2005), consensus in distributed systems (Olfati-Saber, Fax, and Murray 2007), and security of information flow in programs (Sabelfeld and Myers 2003) or between agents (Van Eijck and Gattinger 2015).

Different features of knowledge are required for different applications: In the simplest case, we need only to tag knowledge with an agent or set of agents that knows it. *Reflective* knowledge (Sosa 2009) allows agents to reason about what is not known and what others know (or do not know). Without this stratification, agents could not reason about the effects of their actions on the epistemic states of other agents.

The two features of *dynamism* (changing knowledge) and *reflection* characterize the space that we investigate, since this reasoning is what must be carried out to analyze play in partial information games while also generalizing to other domains.

There are more approaches to how to model agent's knowledge than we can enumerate here, so we will only briefly talk about general approaches, including an example for each. Many applications use specialized data structures that only capture the granularity of knowledge relevant to that application, as e.g. in the work by Ryan et al. (2015), who model characters that can misremember and lie by tracking where they obtained each piece of information. Another approach is to represent agents' uncertainty as multi-valued variables, which typically range over true/-false/unknown, but can also be extended to a range of values representing a probability of certainty. The work by Brenner and Nebel (2009) is an example for this approach in a planning setting, but they do not have a representation for beliefs about other agents' beliefs. Finally, some researchers, such as Liau (2003), use modal logic to represent beliefs of agents, including beliefs about beliefs. Our approach is also based on such a logic, and we will therefore describe it in more detail now.

Epistemic Modal Logic

Our work primarily draws from research on Epistemic Modal Logic, using the classical model for knowledge based on *possible worlds*, where something is considered to be known by an agent if it holds in all worlds the agent considers possible. The standard semantical model for such logics is given by Kripke structures (Kripke 1963). Since this has been a topic of research for decades, in the interest of brevity we will refer to Halpern (1986) for a survey of early research on the topic. For our work, we are particularly interested in how to model the change of agents' knowledge by actions. Van Ditmarsch et al. (2007) describe the history of *Dynamic Epistemic Logics*[1], which form the basis for our work. Much of the research on Dynamic Epistemic Logic has been about expressivity and provability, but our focus lies with how to write actions concisely. We are not the first ones to care about brevity, though. One of the claims of Belardinelli et al. (2016) is that formulas in their logic are exponentially more succinct than a previously developed logic. In contrast to their work, our work focuses more on developing human readable encodings of real world problems in the style of imperative programming languages.

Baltag's Action Language

The logical language we use as the basis of our work was developed by Alexandru Baltag to describe games with belief updates and suspicious players (Baltag 2002). This language has very few primitives, which makes it very suitable for reasoning and implementation, but is capable of expressing a wide range of actions, including *deception* and agents merely *suspecting* that an action is happening, even if that is not actually the case. Actions include $\mathrm{flip}\, p$, which "flips" the truth value of a single literal p; the test operator $?p$, which only executes subsequent actions if p holds; a sequencing operator $a \cdot b$ (product); and non-deterministic choice between two actions $a + b$ (sum). Baltag also includes notation for aggregate sums and products $\Sigma_i a_i$ and $\Pi_i a_i$.

Actions in Baltag are applied to an *epistemic state*, which consists of a set of worlds, with one world marked as the actual world, and an accessibility relation between worlds for each agent that defines which worlds an agent considers possible in any particular world. A *belief* of an agent d, written $\Box_a p$, means that p holds in all worlds that are accessible from the actual world.

Epistemic actions describe how an action *appears* to agents. The suspicion operator $(a)^d$ represents that an agent d suspects that the action a is happening, but by itself this does not cause a to actually happen. The mutual, introspective learning operator $(a)*^D$, on the other hand, causes a group of agents D to truthfully and mutually learns that a is happening, i.e. a happens and every agent in D is aware of it happening, and is aware that every other agent in D is aware of it happening, etc.

Actions are represented by *event structures*, which are similar to Kripke structures, consisting of a set of possible actions (as opposed to possible worlds in a Kripke structure), with one of them marked as the *actual action* and an accessibility relation $\rightarrow_d$ for each agent d between the possible actions. The interpretation of this accessibility relation is that when an action a happens, an agent believes that any action a' that can be accessed via $a \rightarrow_d a'$ could have happened.

[1]Many authors distinguish between *Public Announcement Logic* (Plaza 1989) and Dynamic Epistemic Logic, but we will follow Van Ditmarsch et al.'s terminology and refer to all of these as Dynamic Epistemic Logics.

In contrast to Kripke structures, event structures additionally have preconditions associated with each possible action that define in which worlds they are applicable. For example, Baltag defines a classical `if` statement, e.g. `if(`p`) then` ϕ `else` ψ, as $(?p\cdot\phi + ?\neg p\cdot\psi)$. When applying such a non-deterministic action to an epistemic state, each deterministic option is applied to the state, if its precondition holds in that state. In this case, exactly one of the two options will hold, since either p or $\neg p$ will hold, but in the general case, the result of an action application can be a set of worlds. To apply the appearance of an action $(a)^d$, it is applied to all worlds that are accessible by d from the actual world in which its preconditions hold. If an action appears to an agent as a non-deterministic choice between multiple actions that change the world in different ways, applying this action to a world they consider possible will result in multiple new worlds they now consider possible, thus increasing their uncertainty about the world. Conversely, if an action that appears to an agent as having preconditions, that action will only be applicable in worlds in which these preconditions hold, and the agent will no longer consider other worlds possible, decreasing their uncertainty.

Practical Specifications

Our system, Ostari was designed around two key ideas: The first is that instead of manipulating single bits/literals, users can define *properties*, which are partial functions with *signatures* defined in the execution environment or *context*. For example, the context for a card game may define a property `at` with a signature of $Players \times HandIndex \mapsto Cards$. The second idea is the use of a more procedurally oriented syntax with statement types that represent common epistemic operations. These statements are used to compose *actions* that manipulate the values assigned to properties as well as players' knowledge about them. Our system then translates these statements and property definitions that provide a compact representation of the actions into the corresponding, typically much larger, Baltag expressions. In this section we will not only discuss how our system performs this translation, but also how we ensure that we do not actually have to generate and evaluate the full, compiled formula.

Syntax

The basic building block of Ostari is an action, which is a block of code that has public and secret parameters and conditionally sets property values and/or manipulates agents' knowledge. Secret parameters are only known to a subset of the agents. Listing 1 shows an action for a made-up game in which the objective is to find Aces in the players' hands, and hands are hidden from all players (including the holder). It can be read as: `findaos` is an action with two parameters: p, a $Player$, and i, a $HandIndex$. When the action is executed, if p knows (the symbols `[]` represent $\Box$) that the ith card in their hand is the Ace of Spades, all players are told that p has it (but not where!). Otherwise, p (but not the other players) learns truthfully which card they actually have at their ith hand position. We will describe how the individual statement types map to Baltag in more detail below.

```
findaos(p: Players, i: HandIndex)
 if ([] p at(p,i,AceOfSpades))
   public (Players) holder(AceOfSpades) = p
 else
   learn (p) Which c in Cards: at(p,i) == c
```

Listing 1: An example for an action written in Ostari

More generally, the BNF definition of what an action definition looks like can be seen in figure 2.

⟨*action*⟩ ::= ⟨*identifier*⟩(⟨*parameter*⟩*) ⟨*statement*⟩

⟨*parameter*⟩ ::= ⟨*identifier*⟩ : ⟨*identifier*⟩
  | ⟨*identifier*⟩(⟨*player*⟩+) : ⟨*identifier*⟩

⟨*statement*⟩ ::= ⟨*property*⟩ = ⟨*term*⟩
  | learn (⟨*player*⟩+) ⟨*fact*⟩
  | suspect (⟨*player*⟩+) ⟨*fact*⟩
  | public (⟨*player*⟩+) ⟨*statement*⟩
  | if (⟨*condition*⟩) ⟨*statement*⟩ else ⟨*statement*⟩
  | { ⟨*statement*⟩* }

⟨*term*⟩ ::= ⟨*property*⟩
  | ⟨*constant*⟩
  | Null

⟨*property*⟩ ::= ⟨*identifier*⟩ (⟨*term*⟩*)

⟨*fact*⟩ ::= ⟨*condition*⟩
  | Each ⟨*identifier*⟩ in ⟨*identifier*⟩ : ⟨*condition*⟩
  | Which ⟨*identifier*⟩ in ⟨*identifier*⟩ : ⟨*condition*⟩

⟨*condition*⟩ ::= ⟨*property*⟩ == ⟨*term*⟩
  | ⟨*property*⟩ != ⟨*term*⟩
  | Forall ⟨*identifier*⟩ in ⟨*identifier*⟩ : ⟨*condition*⟩
  | Exists ⟨*identifier*⟩ in ⟨*identifier*⟩ : ⟨*condition*⟩
  | [] (⟨*player*⟩): ⟨*condition*⟩
  | not [] (⟨*player*⟩): ⟨*condition*⟩
  | ⟨*condition*⟩ or ⟨*condition*⟩
  | ⟨*condition*⟩ and ⟨*condition*⟩

Figure 2: The syntax for action definitions in Ostari

Compilation to Baltag

When an action is executed by providing values for its parameters, the macros have to be compiled to Baltag. First, the public parameters are substituted throughout the action body for the values provided for them. For secret parameters, there will be two versions of the action: One, ϕ, with the actual value substituted for it, called the *actual* action, and one, $\sum_i \psi_i$ consisting of the sum (non-deterministic choice) of actions with all possible assignments for the parameter values, called the *appearance*. The actual action is the one that is executed, and visible to the agents a aware of the value of the secret parameters, whereas the appearance describes all possible assignments, which is how the action appears to all other agents B. We encode the whole action as $(\phi) *^a .(\sum_i \psi_i)^B$. We will now describe in detail how properties are compiled to predicates and then briefly outline how the individual statements are compiled.

Property compilation During compilation, a property $p(x) = y$ is mapped to a predicate $P(x, y)$. By definition, among all possible values of y at most one of the $P(x, y)$ holds at any time. If none holds, it means the property is not set, otherwise it is set to the value of y for which $P(x, y)$ holds. Property expressions can be nested, and special care has to be taken when compiling a property term that appears inside another expression. Consider, for example `f(g(x)) = y`. Another way to express this would be `Exists x': g(x) = x' and f(x') = y`, or in predicates $\exists x'\ G(x, x') \wedge F(x', y)$. However, formulas in Baltag cannot have quantified variables, so to compile these expressions we need to resolve the quantifier by inserting all values the variable can take, and connecting the resulting formulas with the choice operator $+$, resulting in $\sum_{x'} ?G(x, x') \cdot ?F(x', y)$. Since every property can only be set to at most one value, for each such sum, at most one term will be applicable in any particular world. Now consider an expression that has multiple layers of nesting. From the inside out, we take the property, prepend it to the formula and add an existential quantifier. To resolve the quantifiers, we then construct a formula for every combination of values of these temporary variables and join all of these formulas with the choice operator. The resulting formula will thus have the form $\sum_{\vec{v}} ?F_1(x, \vec{v}_1) \cdot \prod_i ?F_i(v_i, v_{i+1}) \cdot ?F_n(v_n, y)$, where $\vec{v}$ ranges over all combinations of assignments of values to the temporary variables.

Statement Compilation While it is possible to write terms directly in Baltag in our system, by embedding them in angle brackets, it is usually more convenient to make use of our macro system. It compiles the different statement types in the following way:

- **Assignment** statements (`f(x) = y`) are compiled to two parts, that are executed in order: First, $F(x, y')$ is flipped if it currently holds for any y', i.e. if `f(x)` was previously set to some y', it will be unset, then $F(x, y)$ is flipped.
- **Suspect** and **Learn** (`suspect/learn (a) p`) statements are simply wrappers for Baltag's (suspicious) learn and mutual, truthful learn operators, i.e. they are compiled to $(p)^a$ or $(p)*^a$, respectively. Since p can only be a condition, it will never modify the actual world, and thus the only difference between `suspect` and `learn` is that the former does not necessarily provide truthful information.
- **precondition** statements (`precondition p`) are compiled to Baltag test statements, i.e. $?p$.
- **If/Else** statements (`if p then a else b`) are compiled to a non-deterministic choice, where one choice has p as a precondition, while the other has $\neg p$, i.e. $?p \cdot a + ?\neg p \cdot b$.
- The **Public** modifier (`public (a) s`) can be used on any statement, and causes that statement to be wrapped in a mutual, truthful learning operator, i.e. $(s)*^a$. This is used to make changes in the world known to the agents. An `if` statement modified by a `public` modifier will have both choices wrapped in the mutual learning operator individually, i.e. the agents will know which branch was taken.

Note that the condition used in the `suspect` and `learn` statements can have two additional, special quantifiers:

- **learn/suspect Which** behaves similar to `learn/suspect Exists`, in that it tells the agent whether there is at least one item satisfying the condition, but additionally it tells the agent which item that is. In terms of Baltag, `learn (d): Exists x in Xs: p(x)==y` will be mapped to $(?P(x_1, y) + ?P(x_2, y) + \cdots + ?P(x_n, y))*^d$, i.e. an action that applies whenever any of the $p(x_i)$ holds, whereas `learn (d): Which x in Xs: p(x)==y` will be mapped to $(?P(x_1, y)) *^d + (?P(x_2, y)) *^d + \cdots + (?P(x_n, y))*^d$, with the mutual learning operator applied to each term in the sum individually. The main use of `learn/suspect Which` is to inform an agent about the value of a property. Since a property can only have one value, only one of the non-deterministic choices will apply. For example, the action of a player looking at the top card of the deck may be encoded as `learn (d): Which c in Cards: at(deck, 1) == c`.
- **learn/suspect Each** is used in another common scenario, in which an agent should learn about all values that satisfy a given condition. For example, to tell a player which cards in their hand are spades, one would write `learn (d): Each i in Index: suit(at(hand, i)) == spades`. The `Each` quantifier will result in an action, that consists of a sequence with one item for each element of the set that is quantified over. Each of these items is a non-deterministic choice between a test of the condition and a test of the negation of the condition for the given element, and these are individually wrapped in the learning operator. In other words, for each element of the set, the agents learn whether or not the condition holds for it, which ultimately tells them exactly for which subset of elements it holds. For example, if the player has 2 cards in hand, i will range over the elements 1 and 2, and for each of these values the formula will be $(?(\text{suit}(\text{at}(hand, i)) == spades)) *^d + (?\neg(\text{suit}(\text{at}(hand, i)) == spades))*^d$, with i substituted accordingly. The resulting terms are then joined using the composition operator. Note that $\text{suit}(\text{at}(hand, i)) == spades$ will be expanded as described above to use predicates. We omitted this translation here for brevity.

Implementation Details

We have implemented our system Ostari in Haskell, which evaluates expressions lazily. Our implementation makes use of this fact in several places. States are represented as tuples containing the facts that hold in the actual world, and, instead of having an explicit list of all possible worlds, the appearance map as a function. For a given agent d, this function returns the set of all worlds that are reachable from the actual world. However, these worlds, in turn, have facts that hold in them and their own appearance to agents, and thus their own appearance map, i.e. they are also tuples consisting of facts and an appearance map function, using the same representation as we use for the actual world. This allows us to recursively apply the appearances of actions to the appearance of a world in exactly the same way as we apply the actual action to the actual world, and results in an arbitrarily nestable theory of mind. But consider what would happen if we did not have lazy evaluation at this point: When we apply an action to a world, we apply the appearance of that action to the appearance of that world. However, the appearance of each action in turn contains an appearance map, that would have to be applied to the appearance of the appearance of the world, and so on. In other words, because action and state appearances are defined recursively, applying an action would also recursively apply its appearances to infinitely many levels. In practice, the number of worlds will be finite, avoiding infinite appearance application, but would still require applying actions and their appearances to a potentially very large number of states and appearances or states. Using lazy evaluation, however, we will only ever apply actions to the appearances of states that are actually eventually observed using a query or print operation.

When compiling nested property terms to predicates, the number of terms is exponential in the number of levels of nesting, i.e. $\vec{v}$ will range over an exponential number of values in the resulting formula $\sum_{\vec{v}} ?f_1(x, v_1) \cdot \prod_i ?f_i(v_i, v_{i+1}) \cdot ?f_n(v_n, y)$. Our implementation addresses this problem in two ways. First, note that for any fixed value for v_1, we have all possible assignments of values to v_2 through v_n. An alternative compilation would therefore be $\sum_{v_1} ?f_1(x, v_1) \cdot (\sum_{\vec{v}} ?f_2(x, v_2) \cdot \prod_i ?f_i(v_i, v_{i+1}) \cdot ?f_n(v_n, y))$, where $\vec{v}$ now only ranges over the variables v_2 through v_n. Of course, in total we still end up with the same number of choices, but the actual execution is faster, because for every v_1 for which $f_1(x, v_1)$ does not hold in the state the action is executed in, the whole inner sum does not need to be evaluated at all, and because of Haskell's lazy evaluation we never even generate those terms in that case. We can then recursively repeat this process for the inner sum. It is also not necessary to start with variable v_1, and in fact our approach instead uses the variables in descending order of how many values they can take, eliminating more branches early.

Another optimization comes from an observation made in practice. When using Ostari, some property values are never going to change, and others only have a limited range that depends on their arguments. For example, a property `color` of a card never changes in most games, while a property `top` that maps card stacks to cards may only have a limited range, for example when only red cards may be put onto the red stack. Our system allows users to designate properties as *static*, meaning they never change, or *restricted*, meaning they can only hold for a subset of their domain. These restrictions cause the compilation process to only include terms in which all properties have valid values. Furthermore, because static properties always have the same value, we can also drop the actual check in the resulting formula.

Applications

To demonstrate the capabilities of Ostari we will discuss in detail how it can be used to describe game actions of the cooperative card game Hanabi, and how these actions can then be used to design agents for the game. This demonstration is intended to show how concisely actions can be written using our macros compared to how verbose a direct encoding in Baltag would be, but also how our system can answer queries about the epistemic state of agents and demonstrate its other capabilities. We will also briefly describe other applications of our system.

Hanabi

Hanabi (Bauza 2011) is a cooperative card game, in which players build fireworks, represented by stacks of cards with ranks from 1 to 5 in five different colors. Contrary to most other card games, players hold cards with the faces pointing away from themselves, so that everyone sees everyone else's cards, but not their own. The goal of the game is to place cards in ascending rank on the stack corresponding to their color. To provide information to players about their cards, hints may be given to a player. Every such hint tells a player all cards in their hand that have a specific color or rank, for example which of their cards are red. Hanabi is an interesting game for AI researchers because it requires communication to overcome the limitations of partial observability, but only provides very limited options for this communication. Osawa (2015) developed several AI agents for the game, noting that having a model of what the other agent knows improves the score that is obtained by the agents.

Listing 2 shows how the action representing giving a player the hint that tells them about all their cards of a particular color could be encoded in Ostari. For comparison, for five cards in a player's hand, this operation alone compiles to 125 terms in the resulting Baltag formula, when using the optimizations of term reordering and eliminating static properties described above. This formula would already be cumbersome to write by hand, but also not intuitive to come up with. A naive approach involving enumerating all possible hands would be completely infeasible, as it would require at least $\binom{25}{5} = 53130$ terms. Our macros take the burden of coming up with clever encodings to shorten these expressions from the user and perform them automatically.

```
hintcolor(p: Players, c: Colors)
  learn (p): Each i in HandIndex:
          color(at(p, i)) == c
```

Listing 2: The "hint about a color" action for Hanabi

In the initial state of the game, every player knows every other player's cards, as well as that they know the respective other player's cards, etc. Additionally, every player would consider every world possible in which the cards in their own hand are among those that they don't see. Executing a hint action will then eliminate all worlds that would conflict with the information that was given from the player's set of worlds they consider possible.

Listing 3 shows how the action of playing a card could be encoded using our macros. If the card is the next one in sequence, it will be put on top of the corresponding stack, otherwise the number of mistakes the players made will increase by one. When compiled to Baltag, the condition in the `if` statement alone results in 888 terms. Ostari, in comparison, allows a very concise description of game rules.

```
play(p: Players, i: HandIndex)
{
  public if (rank(at(p,i)) ==
    succ(rank(top(stack(at(p,i))))))
      top(stack(at(p,i))) = at(p,i)
  else
      mistakes() = succ(mistakes())
  public at(p,i) = Null
}
```

Listing 3: The "play a card" action for Hanabi

Epistemic Queries

Executing actions is only useful if we can also examine the results. Our system provides several capabilities for this purpose. On a basic level, it is possible to print the actual world after the execution of any action, or to print the actual world and what that world looks like to all agents. However, since this contains every property assignment and there may be a large number of worlds that an agent considers possible this can quickly become unwieldy to read. We therefore also provide a way to perform queries on the state. The syntax for queries is equivalent to the syntax for conditions, so it is possible to, for example, query if an agent knows that a property has a certain value, which would otherwise require manually verifying that this is the case in all worlds in the appearance map. Listing 4 shows how to define what to execute and query in our implementation. The query is to be read as "Does a know that the card at position 1 in a's hand is the Red 1?". While this is indeed the case, a does not know anything about their hand initially, so the query is answered with "False". After the first action, which tells a where all their red cards are, a still does not know that the card is a 1, so the query is again answered with "False". After the second hint action, however, a knows that the card is red and a 1, so it is a red 1 in all worlds they deem possible, and the query is answered with "True".

```
query: [] (a): at(a,1) == Red1
hintcolor(a, red)
query: [] (a): at(a,1) == Red1
hintrank(a, 1)
query: [] (a): at(a,1) == Red1
```

Listing 4: An execution sequence for Hanabi

Another key feature of Baltag our system exposes is that of suspecting actions that don't actually happen. Instead of executing e.g. `hintcolor(a, red)` it is possible to execute `b suspects hintcolor(a, red)`, which will then update the suspecting agent's mental state accordingly. This capability may not seem very useful at first, but it represents an agent's planning of actions and imagining what their outcome would be. To complement this planning process, it is also possible to specify a goal condition, which will cause the system to use breadth-first search (Bundy and Wallen 1984) for a sequence of actions, both executed and suspected ones that, when applied to the current state, will cause the goal to be satisfied. For example, instead of executing the two hint actions in listing 4, we could have written `goal: [] (a): at(a,1) == Red1` which would have caused the system to find a sequence of actions such as the one we executed.

Other applications

Hanabi is not the only game in which agents benefit from having a mental model of other players' beliefs. Our system also allows concise definitions of game rules commonly found in social deduction games. In particular, One Night Ultimate Werewolf combines actions that have both epistemic- and non-epistemic effects, as in the troublemaker role, that can secretly exchange the roles of two other players without any other player being aware of the exchange. In contrast to Hanabi, One Night Ultimate Werewolf is also not cooperative, and therefore players have an incentive to lie or mislead. Baltag provides the means for players to merely suspect an action happening or a fact being true, and our macro system makes it easy to encode such actions.

Our system also has applications beyond games, like narrative generation. In many stories it may be desirable for actors to have a mental model of the beliefs of other actors. Consider, for example, a detective story, in which the actions of the criminal depend on the beliefs of the detective, or even just their beliefs about the beliefs of the detective. Using our goal-directed search capabilities it is possible to generate stories in which the murderer tries to mislead the detective into suspecting another character of the crime.

Conclusion and Future Work

We have presented Ostari, our implementation of Baltag's variant of Dynamic Epistemic Logic, which is available on github[2]. Our implementation provides several macros that make writing actions that manipulate agents' beliefs less cumbersome than writing them in the logic directly, while still exposing the expressivity of the logic. This includes the ability to have agents suspect actions or be uninformed about some details of an action, such as the value of some of the parameters. We have detailed how our macros are compiled to Baltag formulas, with few lines of Ostari code often resulting in hundreds of terms in the formula. Our system allows users to define such actions, an initial epistemic state and an execution sequence. This execution sequence can consist of actions that are applied to the epistemic state, queries on the state, as well as goal directives that cause the system to find a sequence of actions such that the goal is reached. We have shown in detail how our system applies to the cooperative card game and briefly provided an overview of other applications. For future work, we plan on applying our system to more games in which manipulating and reasoning about players' beliefs is an integral part of game play and use it as the basis for a framework for agents for such games.

[2]http://github.com/yawgmoth/Ostari

References

Alspach, T., and Okui, A. 2014. One night ultimate werewolf. https://beziergames.com/collections/all-uw-titles/products/one-night-ultimate-werewolf.

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

Baron-Cohen, S. 1997. *Mindblindness: An essay on autism and theory of mind*. MIT press.

Bauza, A. 2011. Hanabi. http://www.antoinebauza.fr/?tag=hanabi.

Belardinelli, F.; van Ditmarsch, H.; and van der Hoek, W. 2016. Second-order propositional announcement logic. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, 635–643. International Foundation for Autonomous Agents and Multiagent Systems.

Brenner, M., and Nebel, B. 2009. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems* 19(3):297–331.

Bundy, A., and Wallen, L. 1984. Breadth-first search. In *Catalogue of Artificial Intelligence Tools*. Springer. 13–13.

Castelfranchi, C. 1998. Modelling social action for ai agents. *Artificial Intelligence* 103(1):157–182.

Dignum, F.; Prada, R.; and Hofstede, G. J. 2014. From autistic to social agents. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, 1161–1164. International Foundation for Autonomous Agents and Multiagent Systems.

Eskridge, D. 2009. The resistance. http://www.indieboardsandcards.com/resistance.php.

Halpern, J. Y. 1986. Reasoning about knowledge: an overview. In *Proceedings of the 1986 Conference on Theoretical aspects of reasoning about knowledge*, 1–17. Morgan Kaufmann Publishers Inc.

Hintikka, J. 1962. *Knowledge and belief: an introduction to the logic of the two notions*, volume 4. Cornell University Press Ithaca.

Kripke, S. A. 1963. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly* 9(5-6):67–96.

Liau, C.-J. 2003. Belief, information acquisition, and trust in multi-agent systems–a modal logic formulation. *Artificial Intelligence* 149(1):31–60.

Olfati-Saber, R.; Fax, J. A.; and Murray, R. M. 2007. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE* 95(1):215–233.

Osawa, H. 2015. Solving Hanabi: Estimating hands by opponent's actions in cooperative game with incomplete information. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Plaza, J. 1989. Logics of public communications. In *Proceedings of the 4th ISMIS*, 201–216. Oak Ridge National Laboratory.

Roth, M.; Simmons, R.; and Veloso, M. 2005. Decentralized communication strategies for coordinated multi-agent policies. In *Multi-Robot Systems. From Swarms to Intelligent Automata Volume III*. Springer. 93–105.

Ryan, J. O.; Summerville, A.; Mateas, M.; and Wardrip-Fruin, N. 2015. Toward characters who observe, tell, misremember, and lie. *Proc. Experimental AI in Games* 2.

Sabelfeld, A., and Myers, A. C. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21(1):5–19.

Sosa, E. 2009. *Reflective knowledge: Apt belief and reflective knowledge*, volume 2. Oxford University Press.

Van Ditmarsch, H.; van Der Hoek, W.; and Kooi, B. 2007. *Dynamic epistemic logic*, volume 337. Springer Science & Business Media.

Van Eijck, J., and Gattinger, M. 2015. Elements of epistemic crypto logic. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, 1795–1796. International Foundation for Autonomous Agents and Multiagent Systems.

Wimmer, H., and Perner, J. 1983. Beliefs about beliefs: Representation and constraining function of wrong beliefs in young children's understanding of deception. *Cognition* 13(1):103–128.