Hexa in LCCI

René Mellema

Compiled on: 6th July 2019

1 The Hexa Game

In this example, we will walk through the modeling of the Hexa game in LCCI, using the implementation to aid us in answering questions about the model. The Hexa game was first presented in [3], and is used more often as a testbed for Dynamic Epistemic Logics. We decided to use this as an example instead of Citadels since the models are smaller and easier to understand.

Since the implementation is written in Haskell, the example will also be given in Haskell. While we will try to keep the example easy to follow for those who have no experience with Haskell, if you want to use `LCCI.hs` yourself, it might be useful to read up on using the language. For this we would recommend using 'The Haskell Road to Logic, Maths and Programming, Second Edition' [1]. If you are also interested in using Haskell as a general programming language, then *Learn you a Haskell* [2] is also a good introduction.

1.1 The Game

Before we can start to model the game in LCCI, we will first have to understand the rules of the game. In the Hexa game, there are three players, which we will call Ann (a), Bill (b), and Carol (c). Each of these players will get a unique card out of a deck of three. We will number the cards 0, 1, and 2. The goal of the game is to know the full deal of the cards by asking questions and getting answers.

The only types of questions that we will allow are asking for a specific card. To make the example more interesting and informative, we will also allow for different actions, in particular the swapping of cards between agents.

1.2 Setting up

Before we can start to model the example, we will first have to set up some Haskell specific things. In Haskell code is normally organized in modules, which are named after where they are in the file system. Since this files lives in the `LCCI/Examples` folder and is named `Hexa.lhs`, the module name will be `LCCI.Examples.Hexa`.

```
module LCCI.Examples.Hexa where
```

Since we did not want to reinvent the wheel, we will also need to import some other packages, in particular packages for sets and maps. These come from the collections module from Hackage.

```
import qualified Data.Map as Map
import qualified Data.Set as Set
```

Now we can import the LCCI specific packages that we need. The first of these is the `LCCI.Issue` package, which contains code for representing information states, issues, and state maps.

```
import LCCI.Issue
```

We will also need to represent the models. The `LCCI.Model` package contains code for working with and defining both the static models from IE-PDL and the update models from LCCI.

```
import LCCI.Model
```

Besides needing to represent the models, we will also need to represent formulas of the language, so we can ask questions about them to the implementation. For this, we will need to import the `LCCI.Syntax` package.

```
import LCCI.Syntax
```

Because `LCCI.Syntax` defines the syntax in a way that is not as easy to read or write as one might want, we will import the `LCCI.Syntax.Pretty` package to use functions that are a bit easier on the eyes.

```
import LCCI.Syntax.Pretty
```

We will also want to be able to evaluate formulas in models, which we can do using the `supports` function from the `LCCI.Evaluation` module.

```
import LCCI.Evaluation
```

There is also a module that has functions for the creation of action models for common scenarios, which we will use in our discussion of action models.

```
import LCCI.Announcements
```

When we want to define our own action models, we will also need to define the substitutions for events, for which we will also need to import a package.

```
import LCCI.Substitution as Substitution
```

This are all the packages that we have to import, meaning we can now start to define the actual model.

1.3 The Model

1.3.1 The Worlds and Valuation

The first thing that we need to define for the model itself, are the worlds. While it is usual to number worlds in implementations, LCCI.hs is a bit more flexible. Because of this, we can also use a different representation in the implementation. The only requirement that the implementation has it that the worlds can be ordered. Thanks to this, we can use the representation that is usual for the Hexa model, where each world is represented by the deal of cards.[1]

We will represent a deal of cards by a triple of numbers, where the number is the card, so one of 0, 1, or 2, and the position in the triple signifies the player, so a 0 being in position one means that Ann has card 0, etc. Since triples can be lexicographically ordered, all we have to do is tell Haskell that we want to use this to represent worlds. We do this by declaring a triple as an instance of the type World.

```haskell
instance (Ord a, Ord b, Ord c) => World (a, b, c)
```

We will also want to be able to show the worlds in a readable manner. This means we will have to make our triples an instance of the class PrettyShow, which is what is used in LCCI.hs to prettyprint objects.

```haskell
instance (Show a, Show b, Show c) => PrettyShow (a, b, c) where
    prettyShow (a, b, c) = show a ++ show b ++ show c
```

Now we can define our worlds. Since all the cards are unique and there are only three cards, there is a total of 6 worlds.

```haskell
-- | The worlds in the Hex model
w012, w021, w102, w120, w201, w210 :: (Int, Int, Int)
w012 = (0, 1, 2)
w021 = (0, 2, 1)
w102 = (1, 0, 2)
w120 = (1, 2, 0)
w201 = (2, 0, 1)
w210 = (2, 1, 0)
```

For the definition of the model later on, we will also need to have a set of all the worlds, which we will already define here.

```haskell
-- | The set of worlds
ws :: Set.Set (Int, Int, Int)
ws = Set.fromList [w012, w021, w102, w120, w201, w210]
```

Now we have our worlds defined we can start to define our valuation function. In LCCI.hs the valuation is a function from a proposition to a world to a boolean.

[1]If we wanted to, we could have also used the IntWorld data type that is already defined by LCCI.hs. Then we wouldn't have to define an instance of World or PrettyShow but we would give up the flexibility we now have.

This function can be defined directly, or built up from a `Map` from worlds to the propositions that are true in that world, using the function `valuationFromMap`. Since we have a large amount of structure in our definition of the worlds, we will go with the first option. This means that we will first have to define our propositional atoms.

We will keep our propositional atoms simple, and add one for each combination of player and card. In `LCCI.hs` propositional atoms start with a capital letter, in order to easily differentiate them from agents.

```
-- | The propositions in the Hex model
a0, a1, a2, b0, b1, b2, c0, c1, c2 :: Proposition
a0 = proposition "A0" -- Ann has card 0
a1 = proposition "A1" -- Ann has card 1
a2 = proposition "A2" -- Ann has card 2
b0 = proposition "B0" -- Bill has card 0
b1 = proposition "B1" -- Bill has card 1
b2 = proposition "B2" -- Bill has card 2
c0 = proposition "C0" -- Carol has card 0
c1 = proposition "C1" -- Carol has card 1
c2 = proposition "C2" -- Carol has card 2
```

For the valuation, it is also very useful to split up this player card pair so we can compare it with the card in the triple again. For this we will define two functions that extract the card or player name from a proposition. Both of these depend upon the pretty string representation of a proposition, which is just the string we gave it before.

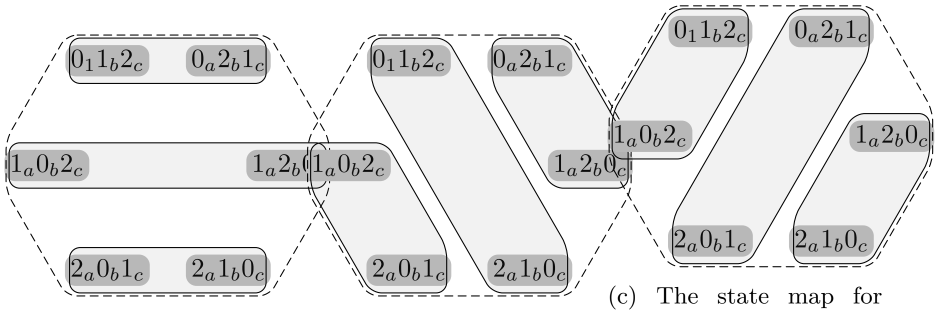
```
-- | Get the number of the card for the given proposition.
card :: Proposition -> String
card = tail . prettyShow

-- | Get the identifier for the player in the given proposition.
player :: Proposition -> Char
player = head . prettyShow
```

So now we can define our valuation. The valuation compares the (string representation of) the card that the player of a proposition `p` has in the world to the card that they have according to the proposition. If they are equal, it returns `True`, and otherwise it returns `False`.

```
-- | The valuation of the propositional atoms
v :: Valuation (Int, Int, Int)
v p (ca, cb, cc)
    | player p == 'A' = show ca == card p
    | player p == 'B' = show cb == card p
    | player p == 'C' = show cc == card p
    | otherwise       = False
```

This ends the definition of the valuation, which means that we can now focus on the last part of the static model, which is the state map.



(a) The state map for Ann (b) The state map for Bill (c) The state map for Carol

Figure 1: The three different state maps

1.3.2 The State Map

For the state map we will first have to consider exactly which scenario we want to model. Since this is most interesting for our purposes, we will pick a situation where all of the agents have been dealt a card, but where they have not looked at it yet. This means that in our initial situation the agents have no information yet, beside the information on how the game works. For simplicity, we will at first assume that the agents are only interested in knowing their own card. This gives the state maps in Figure 1.

Before we can define the state maps in `LCCI.hs`, we will also have to define the agents. This is done by using the function `atom`, that defines an atomic program. In `LCCI.hs`, agents start with a lower case letter, in order to differentiate them from propositional atoms.

```
-- | The "actions" in the Hex model (The knowledge relations)
a, b, c :: Atomic
a = atom "a"
b = atom "b"
c = atom "c"
```

In `LCCI.hs`, a state map for one agent is defined as a `Map` from worlds to issues, just as in LCCI. We can define issues using the function `issue`, which takes a list of states and returns the issue represented by those states. Similarly, we can create a state using the function `state`, that takes a list of worlds, and returns the information state that contains only those worlds. This means that we can define the state map for Ann as follows:

```
ann_map :: StateMap (Int, Int, Int)
ann_map = Map.fromList
    [ (w012, issue [state [w012, w021], state [w102, w120], state [w201, w210]])
    , (w021, issue [state [w012, w021], state [w102, w120], state [w201, w210]])
    , (w102, issue [state [w012, w021], state [w102, w120], state [w201, w210]])
    , (w120, issue [state [w012, w021], state [w102, w120], state [w201, w210]])
    , (w201, issue [state [w012, w021], state [w102, w120], state [w201, w210]])
    , (w210, issue [state [w012, w021], state [w102, w120], state [w201, w210]])
    ]
```

The full state map then becomes a mapping from agents to state maps like `ann_map`, as follows:

```
-- | The state maps for all the various agents.
s :: Map.Map Atomic (StateMap (Int, Int, Int))
s = Map.fromList
    [ (a, ann_map)
    , (b, Map.fromList
        [ (w012, issue [state [w012, w210], state [w102, w201], state [w021, w120]])
        , (w021, issue [state [w012, w210], state [w102, w201], state [w021, w120]])
        , (w102, issue [state [w012, w210], state [w102, w201], state [w021, w120]])
        , (w120, issue [state [w012, w210], state [w102, w201], state [w021, w120]])
        , (w201, issue [state [w012, w210], state [w102, w201], state [w021, w120]])
        , (w210, issue [state [w012, w210], state [w102, w201], state [w021, w120]])
        ])
    , (c, Map.fromList
        [ (w012, issue [state [w012, w102], state [w021, w201], state [w120, w210]])
        , (w021, issue [state [w012, w102], state [w021, w201], state [w120, w210]])
        , (w102, issue [state [w012, w102], state [w021, w201], state [w120, w210]])
        , (w120, issue [state [w012, w102], state [w021, w201], state [w120, w210]])
        , (w201, issue [state [w012, w102], state [w021, w201], state [w120, w210]])
        , (w210, issue [state [w012, w102], state [w021, w201], state [w120, w210]])
        ])
    ]
```

Now we have all the ingredients to define the static model, which we will call `hexa`.

```
-- | The actual model
hexa :: StaticModel (Int, Int, Int)
hexa = StaticModel ws v s
```

Now we can use the model to test some formulas. Formulas are made with the constructors from the `LCCI.Syntax` package, or the functions from the `LCCI.Syntax.Pretty` package. First we will check if in the entire model, so each world in the model, Ann wonders what her card is. We can test this by loading up `ghci` with the current file and executing the following line:

```
supports hexa ws (wonder a (a0 \\/ a1 \\/ a2))
```

This line returns `True`, meaning that Ann wonders what her card is. We will now spent some time explaining what this line does.

The function `supports` is used to check if a certain model and state support a given formula. Here we want to know something about the `hexa` model, so we pass that as the model. Since we want to know if the formula is supported in the whole model, we then pass the set of all worlds (`ws`) as the second parameter. If the formula is supported in the set of all worlds, then, by persistence, it is also supported in all subsets of the set of all worlds, and therefore in the whole model. Then we need to pass the formula that we want to check to the function. For this we use functions from the `LCCI.Syntax.Pretty` package. We use the function `wonder` to create a formula akin to the wonder modality. The function takes two arguments, the first of which is a program representing the agents that wonder something, in our case just `a`, and secondly a formula that we want to know if the agent wants to know. For creating this formula we can use the `\\/` operator, which is inquisitive disjunction. Conjunction and classical disjunction can similarly be written out using slashes. We have to make an inquisitive disjunction with all the cards Ann might have, so we have to use all the propositional atoms we defined for Ann before.

We can now do something similar for Bill:

```
supports hexa ws (wonder b (a0 \\/ a1 \\/ a2))
```

As one would expect, since Bill is not yet interested in which card Ann has, this call returns `False`. However, Bill should know that Ann does not yet know what her card is. We can check for higher order knowledge like this by making more "complex" formulas.

```
supports hexa ws (knows b (neg (knows a (a0 \\/ a1 \\/ a2))))
```

As we would expect, this returns `True`, since it is common knowledge that no-one has looked at their card yet. The function `knows` in LCCI's knowledge operator, and `neg` creates the negation of a formula.

We can also check if some of the rules of the game are common knowledge. For example, we can check if it is common knowledge that when Ann has card 0, Bill or Carol cannot have card 0:

```
supports hexa ws (knows (iter (a .+ b .+ c)) (a0 --> neg (b0 \/ c0)))
```

Which, as one would expect, returns `True`. Here we use the function `iter` to create an unbounded iteration version of a program, and use `.+`, the choice operator, to combine the various agents into one group. `-->` is implication.

1.4 It's Time for Action

Having the static model is nice, but we should also try to introduce some actions into our model. For this we can create action models in a similar manner as we did for defining the `hexa` model. Before we will dive into defining action models however, we will first show how to use them, using the `LCCI.Announcements` package.

1.4.1 Announcements

The `LCCI.Announcements` package exports functions for the creation of public, private, and secret announcements, based on a list of formulas and (one or more) lists of agents. We will use the function `privateInform` to model the action of Ann looking at her card. This function takes 5 arguments. The first is the name of the model as a string, which is used when pretty printing the model in a formula. The second is the group of agents that learn the information, the third is the group of agents that see the information being shared and that are interested in what is being shared, and the fourth is the group of agents that see the information being shared, but don't care. The last argument is a list of possibilities for the material that is being shared.

We will call the model read_a, since Ann reads her card. She is the only one with which the information is being shared, so the first list is just Ann. Bill and Carol are both interested in which card Ann saw, so they want to know what she learned. Since these are all the agents, the next list is empty. The last list contains the possible formulas that Ann might have learned, but now wrapped in the function `Prop`. The reason for this is that `LCCI.Syntax.Pretty` is a bit more liberal in the types of arguments that it accepts then that `LCCI.Announcements` is. When we defined the propositions, we did not define them as formulas, but as propositions, and here we use the function `Prop` to cast them to formulas.

```
ann_look = privateInform "read_a" [a] [b, c] [] [Prop a0, Prop a1, Prop a2]
ann_events = Set.toList $ events $ snd ann_look
```

Besides defining the model, we also extract the events from the model for later use.

Now we can ask the system about the properties that the model has after updating. There are two ways in which we can do this. We can use formulas with an update operator, or we can update `hexa` with `ann_look` to get a new model. For now, we will go with the former, but later on we will show the latter.

Lets say that we want to express that after Ann looks at her card, that she knows which card she has. For this, we will have to use the constructor `Update`, which takes an action model, a list of events, and a formula to test in the new model.

```
supports hexa ws (Update ann_look ann_events (knows a (a0 \\/ a1 \\/ a2)))
```

As one would expect, this returns `True`. We can also test if this is common knowledge among all the agents. Since this line is a little long, we will split it across multiple lines. In ghci we can do this by eclosing the line with `:{` and `:}` on their own separate lines.

```
:{
supports hexa ws (knows (iter (a .+ b .+ c))
                        (Update ann_look ann_events (knows a (a0 \\/ a1 \\/ a2))))
:}
```

Which yet again returns `True`. Now, it is also important to check that Bill does not learn Ann's card from the announcement.

```
supports hexa ws (Update ann_look ann_events (knows b (a0 \\/ a1 \\/ a2)))
```

Luckily, this returns `False`. It is important that Bill now wonders which card Ann has.

```
supports hexa ws (Update ann_look ann_events (wonder b (a0 \\/ a1 \\/ a2)))
```

Which returns `True`, as expected. If fact, this is even a public issue between Bill and Carol:

```
supports hexa ws (Update ann_look ann_events (wonder (iter (b .+ c)) (a0 \\/ a1 \\/ a2)))
```

As mentioned before, we can also create a new model using an action model and a static model. For this we can use the `productUpdate` function from the `LCCI.Evaluation` module. There is similar function for making an updated state, which is the `updatedState` function. For example, if we wanted to transform our model into a model in which every agent knew their card, we could create that model by updating it with action models created by the `privateInform` function. Here we would need to use the function `snd` to extract the action model and throw away the name.

```
hexa1 = productUpdate hexa $ snd ann_look
ws1 = worlds hexa1
hexa2 = productUpdate hexa1 $
            snd $ privateInform "read_b" [b] [a, c] [] [Prop b0, Prop b1, Prop b2]
ws2 = worlds hexa2
hexa3 = productUpdate hexa2 $
            snd $ privateInform "read_c" [c] [a, b] [] [Prop c0, Prop c1, Prop c2]
ws3 = worlds hexa3
```

Now we can use this new model to for example check if Bill knows his card.

```
supports hexa3 ws3 (knows b (b0 \\/ b1 \\/ b2))
```

Which, as one would expect, gives us back `True`.

Now we have seen how the update operator works in `LCCI.hs`, we will take a look at defining action models.

1.4.2 Swapping cards

Bill and Carol are a bit mischievous and decided to swap their cards. This action can also be modelled within `LCCI.hs`, since it also implements the substitutions from LCCI as well. Defining an action model is similar to defining a static model, so here we will have to define the events first.

The swapping of cards sounds like it would only need one event, but it actually will need a bit more. The method we will use will require 3 events, one for Bill and Carol swapping cards 0 and 1, one for them swapping 0 and 2, and one for them swapping 1 and 2. Who holds which card is not important for this model. We will assume that all agents are disinterested in which event actually happens. The reasoning for this is that, if the agents were interested in which card one of the two had before the action, they will be interested after the action has been executed, but if they were not interested, the swapping cards will have no effect on their state maps. This gives us the following events and state maps.

```
e1, e2, e3 :: Event
e1 = Event 1
e2 = Event 2
e3 = Event 3

ei :: StateMap Event
ei = Map.fromList [(e1, issue [state [e1, e2, e3]]),
                   (e2, issue [state [e1, e2, e3]]),
                   (e3, issue [state [e1, e2, e3]])]

em :: Map.Map Atomic (StateMap Event)
em = Map.fromList [(a, ei), (b, ei), (c, ei)]
```

Besides needing state maps for the event, we will also need preconditions for the events. The precondition for swapping card 0 and card 1 is that the one agent has card 0 and the other has card 1. Since the cards are all unique, we won't have to worry about the agents swapping the same card. This gives us the following preconditions.

```
pre :: Map.Map Event Formula
pre = Map.fromList
        [ (e1, (b0 /\ c1) \/ (b1 /\ c0))
        , (e2, (b0 /\ c2) \/ (b2 /\ c0))
        , (e3, (b1 /\ c2) \/ (b2 /\ c1))
        ]
```

That only leaves the substitutions, which work similarly to the preconditions, but have a more complex structure. Instead of mapping every event to a formula, they map every event to a substitution, which maps some propositional atoms to a formula. This is also why `LCCI.hs` has such a clear divide between propositional atoms and formulas. We can make substitutions using the `Substitution.fromList` function in a natural manner.

```
sub :: Map.Map Event Substitution
sub = Map.fromList
        [ (e1, Substitution.fromList [(b0, Prop c0), (b1, Prop c1),
                                      (c0, Prop b0), (c1, Prop b1)])
        , (e2, Substitution.fromList [(b0, Prop c0), (b2, Prop c2),
                                      (c0, Prop b0), (c2, Prop b2)])
        , (e3, Substitution.fromList [(b1, Prop c1), (b2, Prop c2),
                                      (c1, Prop b1), (c2, Prop b2)])
        ]
```

Now we have everything to put together our action model.

```
swap_bill_carol :: UpdateModel
swap_bill_carol = UpdateModel (Set.fromList [e1, e2, e3]) em pre sub
swap_events :: [Event]
swap_events = Set.toList $ events $ swap_bill_carol
```

We can use this action model to create an updated model, but if we want to use it in formulas, we will also need to give it a name. We can do this by wrapping it in a pair with a string.

```
swap_bill_carol' :: (String, UpdateModel)
swap_bill_carol' = ("swap(b,c)", swap_bill_carol)
```

Now we can use this version in a formula as we did before. We can for example say that, after Bill has looked at his card and then swapped, that he knows which card he does not have.

```
:{
supports hexa ws
        (Update (privateInform "read_b" [b] [a, c] [] [Prop b0, Prop b1, Prop b2])
               [Event 1, Event 2, Event 3]
               (Update swap_bill_carol' swap_events
                      (knows b (neg b0 \\/ neg b1 \\/ neg b2))))
:}
```

Here the system gives us `True`, as expected.

1.5 Other Uses of Formulas

Besides checking is a formula is supported, there are also more interesting things that `LCCI.hs` can do with formulas. The first of these is `isDeclarative`, which, as the name suggests, tests if a certain formula is declarative. It only takes the formula to test for. So for example, we can test if "Ann wonders what her card is" is declarative.

```
isDeclarative (wonder a (a0 \\/ a1 \\/ a2))
```

Since this formula is declarative, the system also correctly answers with `True`. We can also try it out on a question, such as "Does Ann, Bill, or Carol have card number 0?".

```
isDeclarative (a0 \\/ b0 \\/ c0)
```

To which the system correctly responds with `False`. If this example would have been more interesting, then one might have wondered what the resolutions for this formula were. Luckily, `LCCI.hs` can also calculate this for us, using the function `resolutions`. This function takes two arguments. The first is the maximum number of worlds for which we need to calculate the resolution, and the second is the formula that you want the resolutions for. Since this formula does not have a non-declarative iteration, the first argument is ignored, so we can pick any number.

```
resolutions 0 (a0 \\/ b0 \\/ c0)
```

This correctly gives us back the three propositional atoms in the formula. Besides being able to calculate the resolutions for a formula, it can also simplify (shorten) formulas using the function `simplify`, and convert them into a LaTeX version using the `toTex` function from the `LCCI.Util.Tex` module. More importantly, it can also calculate the IE-PDL equivalent of every LCCI formula using the functions in the `LCCI.Reduce` module.

The `LCCI.Reduce` module has as goal to implement the algorithm as laid out in **??**, including the T and K functions from **????**. This is also the module that was used in **??** to calculate these reductions. The main functions in this package are the `reduce` and `reduceStep` functions. Both of these reduce a formula, the first does it fully, and the latter does one step from the proof, but works from the outside in instead of the inside out. Both functions take two arguments, a number and a formula, where the number represents the maximum number of worlds that the formula needs to be reduced for, and is used in the calculation of the resolutions of a formula. The second argument is the formula that needs to be reduced.

As an example of this, this call is similar to the one used for the public announcement in **??**:

```
alpha = Prop $ proposition "alpha"
beta = Prop $ proposition "beta"
:{
reduceStep 100 (Update (publicRaise "" [a, b, c] [alpha]) [Event 1]
                       (knows (iter (a .+ b .+ c)) beta))
:}
```

We will probably want to save this result. For that, `ghci` has the special variable `it`, which we will reassign to a different variable.

```
reduced = it
prettyPrint reduced
```

This formula is a lot more complicated then the one given in **??**. This is because the formula in **??** is simplified, so we will also inspect the simplified formula.

```
prettyPrint $ simplify reduced
```

References

[1] H. C. Doets and D. J. N. vanEijck. 'The Haskell Road to Logic, Maths and Programming, Second Edition'. In: *Texts in Computing* (Dec. 2012).

[2] M. Lipovača. *Learn you a Haskell.* 2011. URL: `http://learnyouahaskell.com`.

[3] H. P. van Ditmarsch. 'Knowledge Games'. PhD thesis. Groningen: ILLC Dissertation Series 2000-06., Nov. 2000.