Model Checking with HyLoMoC

Maja M. D. Jaakson

August 29, 2012

1 Introduction

This paper is a literate Haskell program in which we present an implementation of a hybrid logic–model checker, HyLoMoC, in Haskell. We begin by introducing the models and hybrid language which are of interest and then show how the implementation proceeds. The paper concludes with a discussion concerning HyLoMoC's limitations.

2 Hybrid logics

Hybrid logics extend modal logics by allowing direct reference to worlds in a model. A basic hybrid logic typically includes the language of propositional polymodal logic along with a set of *nominals*, which are atomic formulae naming worlds, and a *satisfiability operator* $@_i$ for each nominal i. To this basic language of hybrid logic, one may add a converse operator $\Diamond_i^{\smile}$, which makes it possible to speak of what holds at an i-predecessor of the current state; an existential or universal operator $\mathbf{E}$ or $\mathbf{A}$ (alternatively $\mathbf{E}x$ or $\mathbf{A}x$), which allows us to claim a formula holds at some/every state in the model; and the binder operator $\downarrow$, which allows us to bind a world variable to the current state. In what follows, we deal with a rather rich hybrid logic which contains all of these operators, save the converse operator. We will use the non-binding $\mathbf{E}$ as our existential operator.

Let us now give the syntax and semantics of our language, which we shall call $\mathcal{H}(@, \mathbf{E}, \downarrow)$.

Definition 2.1 Let $\mathsf{REL} = \{\Diamond_1, \Diamond_2, \Diamond_3, \dots\}$ (accessibility relations), $\mathsf{PROP} = \{p_1, p_2, p_3, \dots\}$ (proposition letters), $\mathsf{CONS} = \{i_1, i_2, i_3, \dots\}$ (constant nominals) and $\mathsf{VAR} = \{i_1, i_2, i_3, \dots\}$ (variable nominals) be disjoint and countable sets of symbols. Well-formed formulae of the hybrid language $\mathcal{H}(@, \mathsf{E}, \downarrow)$ in the signature $\langle \mathsf{REL}, \mathsf{PROP}, \mathsf{CONS}, \mathsf{VAR} \rangle$ are defined recursively as follows:

$$\mathsf{FORM} ::= p \mid i \mid x \mid \neg\varphi \mid \varphi \wedge \psi \mid \Diamond_j \varphi \mid \mathsf{E}\varphi \mid \downarrow x.\varphi$$

where $p \in \mathsf{PROP}$, $i \in \mathsf{CONS} \cup \mathsf{VAR}$, $x \in \mathsf{VAR}$, $\Diamond_j \in \mathsf{REL}$ and $\varphi, \psi \in \mathsf{FORM}$.

A hybrid model $\mathbb{M}$ is a quintuple $\mathbb{M} = \langle W, R, V, A, G \rangle$, where W is a non-empty set of worlds or states and R is a subset of REL consisting in a set of functions $\Diamond_j : W \to \wp(W)$, which map worlds to their successors. Note that the valuation function for hybrid logics usually maps members of $\mathsf{PROP} \cup \mathsf{CONS}$ to sets of worlds, with the added condition that the nominals map to singleton sets of worlds; but here, we have split up this function into V and A. The new valuation function V now maps proposition letters to the worlds in which they hold ($V : \mathsf{PROP} \to \wp(W)$) and the assignment function A maps each constant nominal to the single world for which it stands ($A : \mathsf{CONS} \to W$). Similarly, G assigns worlds to variable nominals ($G : \mathsf{VAR} \to W$). Given this function, we also define the x-variant function G^x_w as follows: $G^x_w(y) = w$ when $x = y$; otherwise, $G^x_w(y) = G(y)$.

We define truth in a pointed model in $\mathcal{H}(@, \mathbf{E}, \downarrow)$ as follows:

$\mathbb{M}, G, w \models p$	iff $w \in V(p)$	for $p \in \mathsf{PROP}$
$\mathbb{M}, G, w \models i$	iff $A(i) = w$	for $i \in \mathsf{CONS}$
$\mathbb{M}, G, w \models x$	iff $G(x) = w$	for $x \in \mathsf{VAR}$
$\mathbb{M}, G, w \models \neg\varphi$	iff $\mathbb{M}, G, w \not\models \varphi$	
$\mathbb{M}, G, w \models \varphi \wedge \psi$	iff $\mathbb{M}, G, w \models \varphi$ and $\mathbb{M}, G, w \models \psi$	
$\mathbb{M}, G, w \models \Diamond_j \varphi$	iff $w' \in \Diamond_j(w)$ and $\mathbb{M}, G, w' \models \varphi$	for some $w' \in \mathsf{W}$
$\mathbb{M}, G, w \models @_i \varphi$	iff $\mathbb{M}, G, A(i) \models \varphi$	for $i \in \mathsf{CONS}$
$\mathbb{M}, G, w \models @_x \varphi$	iff $\mathbb{M}, G, G(i) \models \varphi$	for $x \in \mathsf{VAR}$
$\mathbb{M}, G, w \models \mathbf{E}\varphi$	iff $\mathbb{M}, G, w' \models \varphi$	for some $w' \in \mathsf{W}$
$\mathbb{M}, G, w \models \downarrow x.\varphi$	iff $\mathbb{M}', G^x_w, w \models \varphi$	where $\mathbb{M}, \mathbb{M}'$ agree on $\mathsf{W}, \mathsf{R}, \mathsf{V}, \mathsf{A}$

As usual, one may use $\Box_j \varphi$ and $\mathbf{A}\varphi$ as shorthand for $\neg\Diamond_j\neg\varphi$ and $\neg\mathbf{E}\neg\varphi$ respectively.

So much for the semantics; in the next section, we will show how models of $\mathcal{H}(@, \mathbf{E}, \downarrow)$ can be represented and how one can implement a model checker for this logic.

3 HyLoMoC

Let's begin by importing the two modules we'll be needing, Data.List and Data.Map. As we shall see later on, Data.Map will be particularly useful for model-building purposes.

```
module HyLoMoC where

import Data.List
import Data.Map
```

3.1 Language

We create a `Form` datatype for our language in a similar style to that used in the Haskell Road project treatment of propositional logic.

```
type Name = Int
data Form = Prop Name
          | Cons Name
          | Var Name
          | Neg Form
          | Cnj Form Form
          | Dia Int Form
          | At Form Form
          | Exists Form
          | Binder Form Form
          deriving (Eq, Ord)
```

Thus, in addition to having propositional atoms of the form `Prop x`, we now have constant nominals `Cons x` and variable nominals `Var x`, where `x` is a `Name`. Formulae with the possibility operator as a main connective take an `Int` n which corresponds to the nth member of the set of accessibility relations in a model. We run into a typing problem with `At` and `Binder`, however; notice that both of these can take *any* formula as a first argument, while `At` should only take nominals and `Binder` should only take variable nominals there. To get around this, then, we introduce way of checking whether a formula is, indeed, a well-formed formula:

```
wff :: Form -> Bool
wff (Prop x)             = True
wff (Cons x)             = True
wff (Var x)              = True
wff (Neg f)              = wff f
wff (Cnj f1 f2)          = wff f1 && wff f2
wff (Dia n f)            = wff f
wff (At (Cons x) f)      = wff f
wff (At (Var x) f)       = wff f
wff (At x f)             = False
wff (Exists f)           = wff f
wff (Binder (Var x) f)   = wff f
wff (Binder x f)         = False
```

We can now make our formulae instances of the `Show` class, using `!` to represent ↓ in binder formulae.

```
instance Show Form where
  show (Prop p)      = "p_" ++ show p
  show (Cons i)      = "i_" ++ show i
  show (Var x)       = "x_" ++ show x
  show (Neg f)       = '~' : show f
  show (Cnj f1 f2)   = show f1 ++ " & " ++ show f2
  show (Dia n f)     = "<" ++ show n ++ ">" ++ show f
  show (At i f)      = "@_" ++ show i ++
                               " (" ++ show f ++ ")"
  show (Exists f)    = "E (" ++ show f ++ ")"
  show (Binder x f) = "!_" ++ show x ++
                               " (" ++ show f ++ ")"
```

Finally, we define `box` and `forAll` duals:

```
box :: Int -> Form -> Form
box n f = Neg (Dia n  (Neg f))

forAll :: Form -> Form
forAll f = Neg (Exists (Neg f))
```

3.2 Models

We now move on to providing a datatype for our models.

```
data Model a = Model { worlds :: [a],
                       rels :: [a -> [a]],
                       val :: Form -> a -> Bool,
                       assn :: Form -> a,
                       g :: Form -> a }
```

Why do we represent models this way? For one, W is quite naturally represented as `worlds`, a (possibly infinite) list of `a`s, for instance of type `Int` or `String`. The set of accessibility relations R is represented in an equally natural manner by `rels`, a set of functions from worlds to lists of their successors. V is cast as `val`, which takes a (`Prop x`) formula and a world and tells us whether the first is true in the latter. This departs somewhat from the semantics given for $\mathcal{H}(@, \mathbf{E}, \downarrow)$—which would suggest that `val` should be of type `[Form -> [a]]`—but the departure is a welcome one: it allows us to use characteristic functions to give valuations on infinite models which would otherwise be less computationally tractable. (More specifically, consider a case in which we want `(Prop x)` to hold exactly at odd worlds. If we use a valuation which says `(Prop x)` holds exactly at each world in [1,3..] and try to check whether `Prop x` holds at `2`, the checker will never halt; but it will if we use a function which can tell us whether or not `(Prop x)` is true at any given world.) Finally, `assn` assigns a single world to a (`Cons x`) formula; and `g` assigns a single world to each (`Var x`) formula.

It will be handy to have some models around to use as examples, so we'll give two of them: a finite `babymodel` and an infinite `infmodel`. Creating HyLoMoC models is made easier by first defining `functioner`, which takes a list of pairs and a default value as input and outputs a function from the first pair-elements to the second.

```
-- mlookup is just shorthand for Data.Map.lookup.
mlookup :: (Ord k) => k -> Map k a -> Maybe a
mlookup = Data.Map.lookup

functioner :: (Ord a) => [(a,b)] -> b -> (a -> b)
functioner ls defaultvalue =
                   let listmap = (Data.Map.fromList ls) in
                       (\y -> case (mlookup y listmap) of
                               (Just x) -> x
                               Nothing  -> defaultvalue)
```

It's here that the `Data.Map` module becomes useful. When we use `functioner` on a list of pairs `(a,b)`, it first generates a map `listmap` where the `a`s become keys for the `b` values. When the output function is applied to an argument of type `a`, that value is looked up in `listmap`: if it has a return value, the output function gives that value and otherwise, it outputs whatever was chosen as `functioner`'s default value. Thus, `functioner` gives us an easy way to generate total functions from lists.

We now have what we need to take a look at our models. The `babymodel` has three worlds and a single accessibility relation whereby each world sees every odd world. `Prop 1` is true at worlds `1` and `3`; `Prop 2` is true at world `2`; and `Prop 3` is true at worlds `2` and `3`. We use `functioner` to make `Cons 1`, `2` and `3` true at worlds `1`, `2` and `3` respectively; all other `Cons x` are true at world `1`. We give a similar definition for `g`.

```
babymodel = Model [1,2,3]
         [(\y -> [1,3..])]
         (\y -> case y of
                 (Prop 1) -> (\w -> elem w [1,3])
                 (Prop 2) -> (\w -> w == 2)
                 (Prop 3) -> (\w -> elem w [2,3])
                 _        -> (\w -> False) )
         (functioner [(Cons 1,1),(Cons 2,2),(Cons 3,3)] 1)
         (functioner [(Var 1,2), (Var 2,1), (Var 3,3)] 1)
```

The `infmodel` has infinitely many worlds. It too has a single accessibility relation, whereby each world sees (only) its successor. `Prop 1` is true exactly at the odd worlds, `Prop 2` is true exactly at the even ones, and all other `Prop x` fail to hold at any world. Each `Cons x` and `Var x` maps to its respective world `x`. (We add error messages for applications of `assn` and `g` to non-nominals, but this is not needed for the model checker to run properly.)

```
infmodel = Model [1..]
    [(\y -> [y + 1])]
    (functioner [(Prop 1, (\y -> odd y)),
    (Prop 2, (\y -> even y))] (\y -> False))
    (\y -> case y of
           (Cons x) -> x
           _        -> error "Not a cons-nominal." )
    (\y -> case y of
           (Var x) -> x
           _        -> error "Not a var-nominal." )
```

3.3 Checkers: isTruein and hyLoMoC

We are now in a position to see the model checkers themselves. They come in two flavours: `isTruein` checks whether a formula holds at a point in a given model, whereas `hyLoMoC` itself lists the worlds in a model which satisfy the formula provided. It is, of course, a good idea to run `wff` on a formula first if one is unsure whether it is well-formed.

```
hyLoMoC :: (Ord a) => Model a -> Form -> [a]
hyLoMoC m f = case (worlds m) of
  [] -> error "This model is empty!"
  xs -> Data.List.filter (\y -> isTruein m y f) (worlds m)
```

In the event that `hyLoMoC` is fed a proper model `m` (i.e., with non-empty `worlds`), it filters the worlds in `m` of which it's true that `f` holds in those worlds. In order to do this, `hyLoMoC` calls `isTruein`, which does most of the actual work.

```
isTruein :: (Ord a) => (Model a) -> a -> Form -> Bool
isTruein m w (Prop x)     = val m (Prop x) w
isTruein m w (Cons x)     = assn m (Cons x) == w
isTruein m w (Var x)      = g m (Var x) == w
isTruein m w (Neg f)      = not (isTruein m w f)
isTruein m w (Cnj f1 f2)  = (isTruein m w f1) &&
                            (isTruein m w f2)
isTruein m w (Dia n f)    = any (\y -> isTruein m y f)
                            (((rels m) !! n) w)
isTruein m w (At i f)     = isTruein m (assn m i) f
isTruein m w (Exists f)   = any (\y -> isTruein m y f)
                            (worlds m)
isTruein m w (Binder x f) = isTruein (Model (worlds m)
                                 (rels m)
                                 (val m)
                                 (assn m)
                                 (\y -> case (x == y) of
                                    True  -> w
                                    False -> g m y)) w f
```

Implementing `isTruein` follows rather straightforwardly from the semantics for $\mathcal{H}(@, \mathbf{E}, \downarrow)$. A proposition letter is true at a world in a model iff the valuation function outputs `True` when given that model, proposition letter, and world. A constant or variable nominal names a world in a model iff, according to `assn`/`g`, that is the world to which the nominal points. The negation and conjunction clauses are as usual. A formula `Dia n f` holds at a world `w` in a model `m` iff there is a `y` such that `y` makes `f` true and `y` is in the list of worlds accessible to `w` via the `n`th accessibility relation in `rels m`. `At i f` holds at a world in `m` iff `f` holds at the world picked out by `assn m i`; and `Exists f` holds at a world in `m` iff any world in `worlds m` satisfies `f`. Finally, we check whether `Binder x f` holds at `w` in `m` in the manner suggested by our truth definition for $\mathcal{H}(@, \mathbf{E}, \downarrow)$: we check whether `f` holds at `w` in a new model, which agrees on `worlds`, `rels`, `val` and `assn` but has a `g`, which is an `x`-variant of `g m`.

We encourage the reader to experiment with `hyLoMoC`, `isTruein` and our example models.

3.4 Limitations

In the preceding, we implemented model checkers for hybrid logic which can tell us, unproblematically, whether a formula is satisfied in a finite pointed model (`isTruein`) and which points in a finite model satisfy that formula (`hyLoMoC`). Both of these model checkers can deal with infinite models to some extent. Running `hyLoMoC infmodel (Prop 2)`, for instance, will produce continuous output of even worlds, since `Prop 2` holds at all even worlds in `infmodel`. Seeing as `isTruein infmodel w (Exists (Prop 3))` will never halt nor produce any output for any `w` (since it continues to go through the list of worlds `[1..]` looking for a `(Prop 3)`-world), it is not surprising that `hyLoMoC infmodel (Exists (Prop 3))` will never produce the `[]` we are looking for. This is one of the unfortunate limitations we face when it comes to evaluating formulae of hybrid logic on infinite models.[1]

[1]Thanks to the very helpful Erik Parmann for his advice and suggestions regarding this implementation.