NorMC User Guide

Piotr Kaźmierczak, Truls Pedersen and Thomas Ågortnes

This file is a Literate Haskell program.

1 Installation and Usage

NorMC is written in Haskell, and it needs a Glasgow Haskell Compiler interpreter to run. You can download GHC from its homepage at `http://www.haskell.org/ghc/`, but a simpler and better way is to download the whole Haskell Platform: `http://hackage.haskell.org/platform/`. Both Haskell and GHC are free to use and available for Windows, Mac, Linux, and other platforms.

In order to run NorMC, you have to load its files into the GHC interpreter:

```
$ ghci NCCTL.hs
```

You can also load the examples mentioned in the paper, either from the shell (as above) or from within the GHCi command line:

```
Prelude> :l Ex02.hs
```

Current version of NorMC cannot be compiled as a standalone Haskell program, and it always needs GHC to run. Also, NorMC is not compatible with Hugs, and has not been tested with any other Haskell interpreters.

2 Simple Example

In order to demonstrate the use of the model checker, we now present a simple example that introduces basic concepts and constructs. It is a toy example that proves a rather well known fact that two people can pass each other on the sidewalk. We begin with importing the model checker files along with a standard Haskell library for handling lists.

```
module Ex01 where

import NCCTL hiding (owner)
import FODBR (FODBR, build, find1, nubisect)
import Data.List (sort, (\\))
```

In this section we demonstrate how NorMC is used, via a simple example. The core of the model checker is a function called `check`. By importing the NorMC code into a Haskell interpreter, this function can be used as follows:

```
*Example> check myModel myNS myFormula
[s1, s3, s7]
```

Here, the arguments `myModel`, `myNS` and `myFormula`, represent the model, the normative system, and the formula respectively, and the function returns the set of (in this case three) states in which the formula holds in the given model and normative system. We now describe the types of these three arguments expected by `check`.

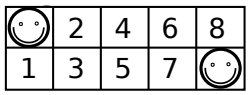
Kripke models are represented by the `Kripke` data structure. To represent sets, such as the set of states in the `Kripke` structure, sorted lists with no duplicate occurrences are used. In NorMC `Kripke` is defined as follows:

```
data (Ord s, Eq p) ⇒ Kripke p s = Kripke {
  agents :: [Int],        states :: [s],
  tr :: FODBR s s,        owner :: (s, s) → Int,
  valuation :: p → [s]  }
```

Also, we provide an abbreviation for the 'backward' and 'forward' state:

```
forwards,backwards :: (Ord s, Eq p) ⇒ Kripke p s → SBMT s s
forwards = fst ∘ tr
backwards  = snd ∘ tr
```

As an example, we describe a simple model of a pavement, divided into 10 *positions*, with two agents on opposite sides of it. The agents can move (or choose to stand still) alternatingly. The model, in its initial state, can be visualised as follows:



The model is defined by the following declaration.

```
exampleModel :: Kripke Prop State
exampleModel = Kripke [0,1] statespace transition owner val
```

The `transition` relation over the `statespace` describes the possible transitions resulting from agents' actions. In addition to these components we also need to assign the `owner` of each transition step. The proposition symbols and their valuation map complete the definition of the model.

Regarding the type of the propositions, `Prop`, we would like to talk about whether a particular agent is found on the east/west edge of the pavement, or in the north/south "lane". The type of the states, `State`, must be able to code the current agent and the positions of both the agents. We can represent this by three integers.

```
data Property = N | S | E | W | T deriving Eq

type Prop  = (Property, Int)
type State = (Int, Int, Int)
```

A state is thus a tuple consisting of three integers. The first indicates the position of agent `0`, the second the position of agent 1, and the last indicates which agent's turn it is. The sorted list of possible states of the model can be described by Haskell's list comprehension syntax succinctly:

```
statespace :: [State]
statespace = [ (p0, p1, i) | p0 ← [0..9], p1 ← [0..9], p0 /= p1, i ← [0,1] ]
```

In the initial state, it is 0's turn, agent 0 is in position 0, and agent 1 is in position 9.

```
initState :: State
initState = (0, 9, 0)
```

The valuation function selects the states from `statespace` in which the indicated agent is in one of the appropriate positions.

```
project :: Int → State → Int
project 0 (p0, _, _) = p0
project 1 (_, p1, _) = p1

val :: Prop → [State]
val (N, ag) = filter (even ∘ (project ag)) statespace
val (S, ag) = filter (odd ∘ (project ag)) statespace
val (W, ag) = filter ((`elem` [0,1]) ∘ (project ag)) statespace
val (E, ag) = filter ((`elem` [8,9]) ∘ (project ag)) statespace
val (T, ag) = filter (λ(_,_,i) → i == ag) statespace
```

The `transition` relation indicates which transitions are available to the agents, or simply what 'moves' agents can perform.

```
transition = build [((p0,p1,i), (p0',p1',1-i)) |
                    (p0,p1,i) ← statespace,
                    p0'       ← possibleSteps (i == 0) p0,
                    p1'       ← possibleSteps (i == 1) p1,
                    (p0',p1',1-i) `elem` statespace]
```

`possibleSteps` is a helper function for `transition`, and says that agents can 'stand', 'move' west, east, and so on. If it's not the agent's turn (first argument is `False`) the only possible move is standing.

```
possibleSteps :: Bool → Int → [Int]
possibleSteps False p = [p]
possibleSteps True p = [p-2, p, p+2]++(if even p then [p+1] else [p-1])
```

Normative systems are represented in the same way as the transition relation, through the `FODBR` type. The following relation `illegal` specifies a normative system for the example: any agent must not place himself directly in front of the other agent, and if an agent is able to move in the general direction required for the satisfaction of the goal, he must do so.

```
illegal :: FODBR State State
illegal = build [ ((p0,p1,i),(p0',p1',1-i)) |
                  (p0,p1,i)   ← statespace,
                  (p0',p1',_) ← (find1 (fst transition) (p0,p1,i)),
                  p1' == p0' + 2 || if i == 0 then (p0' < 8 && p0 ≥ p0')
                                              else (p1' > 1 &&  p1 ≤ p1')]

owner :: (State,State) → Int
owner ((_,_,i),_) = i
```

Finally, formulas are represented by the structures `Formula` and `Coalition`:

```
data (Eq p) => Formula p = Prop p           | Neg  (Formula p)
             | Disj (Formula p) (Formula p) | Conj (Formula p) (Formula p)
             | EX   (Formula p)             | EF   (Formula p)
```

```
              | EG   (Formula p)             | EU   (Formula p) (Formula p)
              | CD   Coalition   (Formula p) | CS   Coalition   (Formula p)
                deriving (Show)

data Coalition = Subseteq [Int]              | Supseteq [Int]
               | Eq [Int]                    | GEQ Int
               | CNeg Coalition              | CDisj Coalition Coalition
                 deriving (Show)

ag,af :: (Eq p) => (Formula p) -> (Formula p)
ag f = Neg (EF (Neg f))
af f = Neg (EG (Neg f))
```

A formula that should be true only in the initial state in the example:

```
initF :: Formula Prop
initF = Conj (Conj (Prop (T, 0)) (Conj (Prop (N, 0)) (Prop (W, 0))))
             (Conj (Prop (S, 1)) (Prop (E, 1)))
```

And now an *objective* formula: agent `0` is east, and `1` is west:

```
goalF :: Formula Prop
goalF = Conj (Prop (E, 0)) (Prop (W, 1))
```

We now have all three arguments for the `check` function:

```
*Ex01> check exampleModel illegal (Conj initF (af goalF))
[]
*Ex01> check exampleModel illegal (CD (GEQ 0) (Conj initF (af goalF)))
[(0,9,0)]
```

We can see that there is no state satisfying both `initF` and `af goalF`, but there exists a coalition which can ensure that we eventually end up satisfying the goal.