NORMC: a Norm Compliance Temporal Logic Model Checker

Piotr Kaźmierczak [a], Truls Pedersen [b] and Thomas Ågotnes [b,a]
[a] *Dept. of Computing, Mathematics and Physics, Bergen University College*
phk@hib.no
[b] *Dept. of Information Science and Media Studies, University of Bergen*
{truls.pedersen,thomas.agotnes}@infomedia.uib.no

Abstract. We describe NORMC, a model checker for *Norm Compliance* CTL, a temporal logic for reasoning about compliance in normative systems, implemented in the Haskell programming language. NORMC is intended as a tool for students, researchers, and practitioners to learn about and understand normative systems, and as an exploratory tool for researchers in multi-agent systems. The objectives of the paper are twofold. First, to give a system description of NORMC. Second, to argue and demonstrate that the Haskell programming language is a natural and useful alternative for model checking multi-agent systems; in particular that the full power of Haskell makes it easy to describe arbitrary multi-agent state-transition models in a natural way.

Keywords. multi-agent systems, normative systems, model checking, haskell

1. Introduction

Normative systems, or *social laws*, have emerged as a promising and powerful framework for coordinating multi-agent systems [13,14,9,10,1,3,7,2,4]. The starting point is a state-transition model of a multi-agent system, typically a *legacy system*, and the goal is to constrain the behaviour of the agents in the system in such a way that the global behaviour of the system exhibits some desirable properties. Such a restriction on agents' behaviour is called a normative system, or a social law. The desirable global properties are typically represented using a (modal) logical formula; the *objective* (typically not satisfied in the initial system).

A key issue in normative systems is the question of *compliance*. Even if a normative system is *effective*, i.e., will ensure that the objective holds, under the assumption that all agents comply with it – how do we know that they will actually comply? And what happens if they do not? There are several possible reasons for non-compliance [4]: an autonomous and rational agent might choose not to comply because it is not in her best interest; a rational agent might not comply by accident (a failure); an irrational agent might choose not to comply without any particular reason. *Norm compliance* CTL (NCCTL) [4] was developed to reason about normative systems and in particular about (non-)compliance. It can be used, e.g., to model check compliance properties such as "which agents *have to comply* for the objective to hold". NCCTL extends the branching-

time temporal logic *Computation-Tree Logic* (CTL) [8] with a modality $[P]$, where P is a *coalition predicate*, i.e., a possible property of groups of agents (*coalitions*). The meaning of the expression $[P]\varphi$ is that if any coalition that satisfies P complies with the normative system, then φ will hold. Examples of NCCTL expressions include the following, which are evaluated in the context of a model and a normative system:

- $[supseteq(C)]\varphi$: if any superset of C comply, φ will hold (C is *sufficient*[1])
- $[\neg geq(k)]\neg\varphi$: at least k agents have to comply for φ to hold (the normative system is k-*necessary*)
- $[geq(n-k)]\varphi \wedge \langle ceq(n-k-1)\rangle\neg\varphi$, where n is the total number of agents: k is the largest number of non-compliant agents the normative system can tolerate whilst still being effective for φ (the *resilience* of the normative system is k)[2]

Formally, $[P]$ has a non-standard *update semantics*, which makes it difficult to use standard branching-time model checkers directly to verify normative system properties specified in NCCTL.

In this paper we describe NORMC, a prototype model checker for NCCTL implemented in Haskell [12]. The intended use of NORMC is as a tool to learn about and understand normative systems, and as an exploratory tool for researchers. Even small examples can be difficult to understand properly without a computational tool, because the number of possible model updates resulting from different groups of agents complying is typically exponential in the size of the model. As a prototype tool intended for academic rather than industrial use, the focus in the implementation of NORMC has not been on efficiency, but rather on clarity, extensibility and ease of use. In particular, standard symbolic model checking optimization techniques have not been implemented. However, the implementation is still efficient enough for interesting and non-trivial examples.

The objectives of the paper are twofold. First, the paper is a system description of NORMC that demonstrates how it can be used to model check normative systems. Second, we want to argue for and demonstrate the usefulness of the Haskell programming language for model checking modal logics, as already suggested by model checkers such as the epistemic logic model checker DEMO [15]. We argue that Haskell's native support for discrete structures and lazy evaluation mechanism makes it well suited for programming model checking algorithms. Furthermore, a Haskell implementation means that it is possible to have the full power of the Haskell language available for the *user* to *describe models* – in contrast with the restricted model description languages available in popular temporal logic model checkers such as SPIN [11] or NUSMV [5].

The paper is organized as follows. We first review the formal normative systems framework and NCCTL. The details of NORMC implementation are discussed in Section 3. In Section 4 we illustrate usage with an example, and show how NORMC was used to find an error in a case study in the literature. We conclude in Section 5.

NORMC, with source code, can be downloaded from `http://pkazmierczak.github.com/NorMC/`. Although not strictly necessary, familiarity with the Haskell programming language is helpful to fully understand the source code snippets in this paper. We provide some explanation, but due to lack of space and a different focus in this paper, we further refer the reader to a short manual (which comes with a simple and illustrative example) that can be found on the NORMC's website.

[1] Notions of sufficiency, k-necessity and resilience are formally defined in [4].

[2] $ceq(n)$ is an abbreviation of $(geq(n) \wedge \neg geq(n+1))$.

2. Background

We give a, necessarily terse due to lack of space, overview of the background; see [4] for more details on NCCTL. Assume a set Φ of propositional variables. As the semantic model we use an *agent-labelled Kripke structure*; a tuple $K = \langle S, s^0, R, A, \alpha, V\rangle$ where S is a finite, non-empty set of *states*; $s^0 \in S$ is the *initial state*; $R \subseteq S \times S$ is a serial binary relation (i.e., $\forall s \exists t\ (s,t) \in R$) on S, which we refer to as the *transition relation*; A is a set of *agents*; $\alpha : R \to A$ labels each transition in R with an agent; and $V : S \to \wp(\Phi)$ labels each state with a set of propositional variables.

A *path* π over a relation R is an infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $\forall u \in \mathbb{N} : (s_u, s_{u+1}) \in R$. $\pi[0]$ denotes the first element of the sequence, $\pi[1]$ the second, and so on. An *s-path* is a path π such that $\pi[0] = s$. $\Pi_R(s)$ is the set of s-paths over R, and we write $\Pi(s)$, if R is clear from the context.

Objectives are specified using CTL formulas. We use an adequate fragment of the language defined by the following grammar:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{E}\bigcirc \varphi \mid \mathsf{E}(\varphi \mathcal{U} \varphi) \mid \mathsf{A}(\varphi \mathcal{U} \varphi)$$

where $p \in \Phi$. The standard derived propositional connectives are used, in addition to standard derived CTL connectives such as $\mathsf{A}\bigcirc \varphi$ for $\neg\mathsf{E}\bigcirc \neg\varphi$ (see [8] for details). Satisfaction of a formula φ in a state s of a structure K, $K, s \models \varphi$, is defined as follows:

$K, s \models \top$;

$K, s \models p$ iff $p \in V(s)$;

$K, s \models \neg\varphi$ iff not $K, s \models \varphi$;

$K, s \models \varphi \vee \psi$ iff $K, s \models \varphi$ or $K, s \models \psi$;

$K, s \models \mathsf{E}\bigcirc \varphi$ iff $\exists \pi \in \Pi(s) : K, \pi[1] \models \varphi$;

$K, s \models \mathsf{E}(\varphi \mathcal{U} \psi)$ iff $\exists \pi \in \Pi(s), \exists u \in \mathbb{N}$, s.t. $K, \pi[u] \models \psi$ and $\forall v, (0 \leq v < u) : K, \pi[v] \models \varphi$;

$K, s \models \mathsf{A}(\varphi \mathcal{U} \psi)$ iff $\forall \pi \in \Pi(s), \exists u \in \mathbb{N}$, s.t. $K, \pi[u] \models \psi$ and $\forall v, (0 \leq v < u) : K, \pi[v] \models \varphi$.

A *normative system* η over K is a set of *constraints* on the behaviour of the agents. Formally, $\eta \subseteq R$, such that $R \setminus \eta$ is a serial relation, represents the *forbidden* transitions. $N(R)$ denotes the set of all normative systems over K. We say that normative systems are *implemented* on Kripke structures, which means that after the implementation all the transitions that are forbidden according to a normative system are removed from the structure. Formally, if η is a normative system over K, then $K \dagger \eta$ stands for the Kripke structure obtained from K by removing the transitions forbidden by η, i.e., if $K = \langle S, s^0, R, A, \alpha, V\rangle$ and $\eta \in N(R)$, then $K \dagger \eta = \langle S, s^0, R', A, \alpha', V\rangle$ where $R' = R \setminus \eta$, and α' is the restriction of α to R': if $(s, s') \in R'$ then $\alpha'(s, s') = \alpha(s, s')$. A set $C \subseteq A$ is called a *coalition*. Let η be a normative system over K, then $\eta \restriction C$ is the normative system restricted to the actions of agents in C: $\eta \restriction C = \{(s, s') : (s, s') \in \eta \ \&\ \alpha(s, s') \in C\}$.

The language of NCCTL extends the language of CTL with an operator $\langle P \rangle$ where P is a *coalition predicate*. The coalition predicates P are defined by the following grammar:

$$P ::= subseteq(C) \mid supseteq(C) \mid geq(n) \mid \neg P \mid P \vee P$$

where $C \subseteq A$ and $n \in \mathbb{N}$. Satisfaction of a predicate P by a coalition C_0, $C_0 \models_{cp} P$, is defined straightforwardly: $C_0 \models_{cp} subseteq(C)$ iff $C_0 \subseteq C$; $C_0 \models_{cp} geq(n)$ iff $|C_0| \geq n$; $C_0 \models_{cp} supseteq(C)$ iff $C_0 \supseteq C$; $C_0 \models_{cp} \neg P$ iff not $C_0 \models_{cp} P$; and $C_0 \models_{cp} P_1 \vee P_2$ iff $C_0 \models_{cp} P_1$ or $C_0 \models_{cp} P_2$. Other coalition predicates can be introduced as abbreviations, such as $eq(C) = subseteq(C) \wedge supseteq(C)$, etc.

Formally, the language of the NCCTL is generated as follows:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{E}\bigcirc \varphi \mid \mathsf{E}(\varphi \mathcal{U} \varphi) \mid \mathsf{A}(\varphi \mathcal{U} \varphi) \mid \langle P \rangle \varphi$$

We also write $[P]\varphi$ to denote the dual coalition predicate: $\neg\langle P\rangle\neg\varphi$.

Formulas of NCCTL are interpreted in a triple (K, η, s) where K is a Kripke structure, η a normative system over K and s is a state of K. The clauses for coalition predicates are as follows:

$$K, \eta, s \models \langle P \rangle \varphi \text{ iff } \exists C \subseteq A \, (C \models_{cp} P \text{ and } K \dagger (\eta \restriction C), \eta, s \models \varphi)$$

All other formulas are defined as for CTL, but carrying the normative system in the context: e.g. $K, \eta, s \models \mathsf{E}\bigcirc \varphi$ iff $\exists \pi \in \Pi(s) : K, \eta, \pi[1] \models \varphi$.

3. Implementation

We now discuss the design and implementation of NORMC. The core of the model checker is the function `check`. It implements an extension of a standard model checking algorithm for CTL formulas, as described in [6], with additional clauses for coalition predicates. `check` takes a model, normative system and formula as arguments and returns the set of states in which the given formula is satisfied. It is used in the following way:

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

where `myModel` is the Kripke structure, `myNS` is the normative system, and `myFormula` is the objective.

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 the data structure `Kripke` is defined as follows:[3]

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

[3] One mysterious element in this code snippet is the type of `tr` (transition relation), which is `FODBR`. It is a type provided by our binary relations library, and it is explained in Section 3.1.

The requirements `Ord s` and `Eq p` state that the type of states `s` should support ordering, and the type of propositional symbols `p` should support equality testing, respectively.

Normative systems are naturally represented in the same way as transition relations, so an example implementation of a function representing a normative system could look as follows:

```
normativeSystem :: FODBR s s
normativeSystem = build [ ((p0,p1,i),(p0',p1',1-i)) |
                          (p0,01,i)   <- statespace,
                          (p0',p1',_) <- ... ]
```

We discuss the `build` function in more detail in a subsection below.

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)
```

Should we need more CTL connectives, it is very easy to implement appropriate abbreviations:

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

The `check` function is our counterpart of $\models$, but rather than a relation $\models \subseteq \mathcal{M} \times \mathcal{N} \times S \times \varphi$, we have `check` : $\mathcal{M} \times \mathcal{N} \times \varphi \to \wp(S)$. The function is defined recursively, and the clauses for propositional variables and path-state-quantifiers are defined as in standard CTL algorithms (see [6] for more details).

The propositional clauses are fairly obvious:

```
check :: (Ord s, Eq p) ⇒ (Kripke p s) → (FODBR s s) → (Formula p) → [s]
check mod sys (Prop p)    = sort $ (valuation mod) p
check mod sys (Neg f)     = (states mod) `nubminus` (check mod sys f)
check mod sys (Disj f f') = (check mod sys f) `nubunion` (check mod sys f')
check mod sys (Conj f f') = (check mod sys f) `nubisect` (check mod sys f')
```

The path-state-quantifiers are checked as follows: for the `EX f` case, `(check mod sys f)` is the set where `f` is satisfied and hence `(EX f)` is satisfied in the set of states

which has a successor in `(check mod sys f).EF f` is satisfied in the set of states `s0` where `f` is satisfied and the image of these states. We repeat this process until it settles on a fixed point. `EG f` is satisfied in the set of states `s0` which satisfy `f` and also in the states which satisfy `f` *and* has an edge into this set. Finally `EU f f'` is satisfied in the set of states where `f'` is satisfied, and (as long as the preimage of a given state is in the set) the intersection of the preimage and the set containing the states in which `f` is satisfied.

```
check mod sys (EX f)      =
    find (backwards mod) (check mod sys f)
check mod sys (EF f)      = fix ff (check mod sys (EX f)) where
    ff ss = ss `nubunion` (find $ backwards mod) ss
check mod sys (EG f)      = fix ff (check mod sys f) where
    ff ss = ss `nubisect` (find $ backwards mod) ss
check mod sys (EU f f')   = fix ff (check mod sys f') where
    ff ss = ss `nubunion` ((find $ backwards mod) ss `nubisect` (check
                                                        mod sys f))
```

For the coalition predicates we need some semantic update functions: `nsimplement` implements a normative system on a Kripke structure (the † operator), `nsrestrict` implements the ↾ operator, and `ir` combines the two, implementing a given normative system restricted to the given coalition predicate:

```
nsimplement :: (Ord s, Eq p) ⇒ (Kripke p s) → (FODBR s s) → (Kripke p s)
nsimplement mod sys = mod { tr = (tr mod) `minus` sys }

nsrestrict :: (Ord s, Eq p) ⇒ (Kripke p s) → (FODBR s s) → [Int] → (FODBR s s)
nsrestrict mod sys coa = restrict sys (λs s' → ((owner mod (s,s')) `elem` coa))

ir :: (Ord s, Eq p) ⇒ (Kripke p s) → (FODBR s s) → [Int] → (Kripke p s)
ir mod sys coa = nsimplement mod (nsrestrict mod sys coa)
```

The function `checkCoaPred` returns `True` iff coalition predicates are satisfied by its arguments: a coalition predicate and a set of agents.

```
checkCoaPred :: Coalition → [Int] → Bool
checkCoaPred (Subseteq set) coa = coa `subset` set
checkCoaPred (Supseteq set) coa = set `supset` coa
checkCoaPred (Eq set)       coa = set == coa
checkCoaPred (GEQ n)        coa = n ≤ length coa
checkCoaPred (CNeg c)       coa = not (checkCoaPred c coa)
checkCoaPred (CDisj c c')   coa = (checkCoaPred c coa) || (checkCoaPred c' coa)

coasGivenCP :: (Ord s, Eq p) ⇒ (Kripke p s) → Coalition → [[Int]]
coasGivenCP mod cp = filter (checkCoaPred cp) (spowerlist $ agents mod)
```

Model checking NCCTL involves quantification over coalitions. As discussed, for each subformula φ the algorithm computes the set of satisfying states (call it $sat(\varphi)$). By the semantics of NCCTL, we have that $sat([P]\varphi) = \{s \in S : \forall C \subseteq A(C \models_{cp} P \Rightarrow K \dagger (\eta \restriction C), \eta, s \models \varphi)$. A naïve implementation of this involves testing every coalition against the predicate, once for each state. However, it is easy to see that the quantifier can be moved out (and similarly for $\langle P \rangle \varphi$):

Lemma 1 $sat([P]\varphi) = \bigcup_{C \models_{cp} P} \{s \in S : K \dagger (\eta \restriction C), \eta, s \models \varphi)\}$ *and* $sat(\langle P \rangle \varphi) = \bigcap_{C \models_{cp} P} \{s \in S : K \dagger (\eta \restriction C), \eta, s \models \varphi)\}$.

Thus we only have to test the predicate once for each coalition, and save a considerable amount of time in practice. This is made use of as follows in the clauses checking the "coalition predicate diamond" (`CD`) and "coalition predicate square" (`CS`):

```
check mod sys (CD c f) = foldl' nubunion [] $
                         map (λmod → (check mod sys f)) $
                             map (ir mod sys) $ coasGivenCP mod c
check mod sys (CS c f) = foldl' nubisect (states mod) $
                         map (λmod → (check mod sys f)) $
                             map (ir mod sys) $ coasGivenCP mod c
```

This concludes the description of the implementation of the model checking algorithm. As the reader familiar with programming model checkers might notice, the code of the whole model checking algorithm is very succinct and compact. The model checker code alone is only 88 lines long, and the library for handling binary relations has 140 lines. That gives only 228 lines in total, compared to thousands of lines of what could constitute a similar tool written in C, C++ or Java.

We would also argue that Haskell's syntax is pretty close to mathematical notation, hence fairly easy to understand for a logician.

3.1. Binary relations

Much of the core functionality of NORMC is found in the handling of the transition relation, and performing a model update operation is performing an appropriate restriction on this relation. We only mention the implementation of relation handling briefly here, due to lack of space and to keep focus on more high-level considerations.

Our relation handling implementation, `FODBR` (Finite Ordered Domain Binary Relation), is a sufficiently efficient representation of a binary relation by two binary search trees. We require that the domain is finite and consists of elements with an ordering, and we represent the binary relation by two multifunctions: $src : \mathcal{D} \to \wp(\mathcal{D}')$ and $trg : \mathcal{D}' \to \wp(\mathcal{D})$. Both these functions are stored as a binary search tree where each node contains a value of the type $(\mathcal{D}, [\mathcal{D}'])$ ('key' and 'value set', respectively).

The above functionality is provided by a collection of functions stored in a separate library that is loaded to the model checker. The library contains a number of useful definitions, most importantly the data type `FODBR`, which is a tuple of two binary trees, and a function `build`, which given a list of pairs `[..., (source,target),...]` constructs a pair of binary search trees consisting of tuples of source and targets, and target and sources, as shown in Figure 1.

The functionality provided by `FODBR` requires that argument lists are sorted and contain no duplicate elements, and guarantees that any returned lists also satisfy these properties. This allows us to implement the usual set theoretic operations of union, intersection and difference efficiently.

The implementation of `FODBR` library is where we make use of Haskell's lazy evaluation mechanisms. For the purpose of our model checking algorithm, only the backwards image is generated – lazy evaluation of the structure ensures that the forward direction

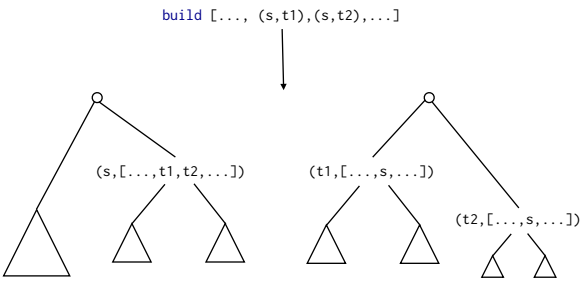

Figure 1. `build` function constructing binary trees, the forward tree on the left, and the backward on the right.

is not calculated. However, should we need to generate counter examples or extend the model checker functionality in some way, Haskell will generate the forward direction as well.

4. Specifying and model checking a complex system

We will now present parts of a specification of an elaborate example taken from [4]. The example describes four researchers attending a conference who need to share a limited amount of resources. The model has 62500 states and 470596 transitions. We present the essential parts of this example and use it to demonstrate the flexibility in describing elaborate models in Haskell/NORMC, and that NORMC can model check pretty large models. Due to lack of space some details must be left out, but we focus on the essential parts.

The states are tuples $s = \langle O_a, O_b, O_c, O_d, i\rangle$ where for each $i \in \{a, b, c, d\} = A$, O_i represents the resources owned by agent i. The resources they need to share are: (i) a printer (the singleton set R_1), (ii) two scanners (the set R_2), and (iii) three computers (the set R_3). The agents do not have the same needs. Agent a needs the printer and a computer, agent b needs a scanner and the printer, agent c needs a scanner and a computer, and agent d only needs the printer. Agents' actions are turn-based. There is no distribution of resources that allows all agents to own the resources they need simultaneously, but there are limitations on the permitted transitions which guarantee that each agent will eventually own all the resources needed.

The first restriction is the implementation of a normative system which sets some basic rules of behaviour. Since the example is rather elaborate, we have to omit much of the formal definition here and refer to [4] for details. Below is an informal description of the normative system η_0 [4]:

> [...] no agent (i) owns two resources of the same type at the same time, (ii) takes possession of a resource that he does not need, (iii) takes possession of two new resources simultaneously, and (iv) fails to take possession of some useful resource if it is available when it is his turn [...]

We describe this model in Haskell using seven integers to represent each state. The first six of these integers represent the owner of, respectively, (1) the printer, (2) scanner_1, (3) scanner_2, (4) computer_1, (5) computer_2 and (6) computer_3. Each of these can take on a

value $0..4$ where 0 represents a *free* resource and a non-zero number indicates the owner of this resource. The seventh integer represent whose turn it is and has a value $1..4$.

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

The set of states is defined as follows.

```
statespace :: [State]
statespace = [(p, s1, s2, c1, c2, c3, a) |
                  p  ← [0..4], s1 ← [0..4], s2 ← [0..4],
                  c1 ← [0..4], c2 ← [0..4], c3 ← [0..4],  a  ← [1..4] ]
```

For the transition relation, if the current agent `a` owns a given resource r (`a` $==$ r), or nobody owns that resource (r $==$`0`), then the current agent can keep (resp. grab) or release (resp. ignore) it, otherwise that resource will not change owner.

```
transition :: FODBR State State
transition = build [((p , s1 , s2 , c1 , c2 , c3 , a),
                     (p', s1', s2', c1', c2', c3', 1 + (a `mod` 4))) |
                    (p,s1,s2,c1,c2,c3,a) ← statespace,
                    p'  ← if (p  == a || p  == 0) then [0, a] else [p ],
                    s1' ← if (s1 == a || s1 == 0) then [0, a] else [s1],
                    s2' ← if (s2 == a || s2 == 0) then [0, a] else [s2],
                    c1' ← if (c1 == a || c1 == 0) then [0, a] else [c1],
                    c2' ← if (c2 == a || c2 == 0) then [0, a] else [c2],
                    c3' ← if (c3 == a || c3 == 0) then [0, a] else [c3] ]
```

We label each transition with its owner. This is a simple projection.

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

We model a proposition symbol by an (agent) index and a keyword. We define the type `Resource` to denote the various resources and give a function which projects from a given state the owner of the given resource.

```
data Resource = Pr | S1 | S2 | C1 | C2 | C3 deriving Eq

project :: Resource → State → Int
project Pr (pr,_,_,_,_,_,_) = pr
project S1 (_,s1,_,_,_,_,_) = s1
-- and so on...
```

The valuation function is now simply defined by removing the states in which given agent does not own the resource in question.

```
type Proposition = (Resource, Int)

val :: Proposition → [State]
val (res, ag) = filter ((ag == ) ∘ (project res)) statespace
```

The normative system η_0 is implemented as the union of four separate relations (see the description of η_0).These are all constructed in a similar way, i.e. restricting the transition relation appropriately, and we show only `component3` which does not allow an agent to grab two resources at the same time.

```
component3 :: FODBR State State
component3 = restrict transition
          (λ(pr,s1,s2,c1,c2,c3,a) (pr', s1', s2', c1', c2', c3', a') →
           (sum $ zipWith (λx y → if (x /= a) && (y == a) then 1 else 0)
                          [pr , s1 , s2 , c1 , c2 , c3 ]
                          [pr', s1', s2', c1', c2', c3']) > 1)
eta_0 :: FODBR State State
eta_0 = component1 `union` component2 `union` component3 `union` component4
```

Next we define the normative system η_1: if an agent owns all his useful resources simultaneously (he is 'happy'), he will make them available in the next turn.

```
stHappy :: Int → State → Bool
stHappy 1 (pr, _,  _,  c1, c2, c3, _) = (pr == 1 && (c1 == 1 || c2 == 1
                                                     || c3 == 1))
-- and similarly for stHappy 2, 3 and 4, defining the states where
-- agents are happy
ownsSomething :: Int → (State → Bool)
ownsSomething n = (pr, s1, s2, c1, c2, c3, a) → (pr == n ||
                                                 s1 == n || s2 == n ||
                                                 c1 == n || c2 == n ||
                                                 c3 == n)
eta_1 :: FODBR State State
eta_1 = restrict transition
        (λs s' → (stHappy (owner (s,s')) s) && (ownsSomething
                                                (owner (s,s')) s'))
```

For convenience, we define two example models, one with no normative system implemented on it (K_0), and the second one with η_0 implemented on it ($K_1 = K_0 \dagger \eta_0$):

```
exampleModel :: Kripke Proposition State
exampleModel = Kripke [1,2,3,4] statespace transition owner val
exampleModel' :: Kripke Proposition State
exampleModel' = ir exampleModel eta_0 [1,2,3,4]
```

Our first objective is that it is always the case that every agent will eventually become happy: $\varphi_1 = \mathsf{A}\Box(\bigwedge_{i \in Ag} \mathsf{A}\Diamond happy(i))$, where the proposition $happy(i)$ (code: `fHappy i`) is true iff i is 'happy'.

```
phi_1 :: Formula Proposition
phi_1 =  ag (Conj (af (fHappy 1)) (Conj(af (fHappy 2))
                                       (Conj(af(fHappy 3))
                                            (af (fHappy 4))))))
```

This concludes the implementation of the example model from [4]. We now proceed with model checking. First we check whether `eta_0` is effective.

```
*Ex02> check exampleModel' eta_1 phi_1
[]
```

The model checker outputs an empty set, indicating there are no states satisfying `phi_1` if no agents are required to comply with `eta_1`. In [4] it is claimed that compliance of coalition $\{1, 2, 3\}$ to `eta_1` is sufficient for the objective to hold (assuming that all agents comply with η_0). Let us check:

```
*Ex02> check exampleModel' eta_1 (CS (Supseteq [1,2,3]) phi_1)
[]
```

Surprisingly, the model checker tells us that the formula is not true in any state. Using the model checker we can produce a trace:

```
[(0,0,0,0,0,0,1),(0,0,0,0,0,1,2),(2,0,0,0,0,1,3),(2,0,0,0,3,1,4),
 (2,0,4,0,3,1,1),(2,0,4,0,3,0,2),(0,2,4,0,3,0,3),(0,2,4,0,0,0,4),
 (0,2,0,0,0,0,1),(0,2,0,0,0,1,2),(2,0,0,0,0,1,3)]
```

In steps 2 and 10 we end up in the same state. We have a loop where all agents comply with both η_0 and η_1. No agent is ever 'happy' in this loop. So, there is a path π in $\Pi(s_0)$ along which it is not the case that every agent will eventually be 'happy'. Thus, there is an error in the example in [4]. This error was not obvious, and this illustrates the benefit of software tools even for "toy" examples.

The problem with the example from [4] is that η_1 as stated is too weak; as seen above it allows agents to simultaneously grab some resources and release others in an endless loop without ever becoming 'happy'. We now introduce an additional condition that tells the agents to not release any resources they posses until they are 'happy':

```
dont_release :: FODBR State State
dont_release = restrict transition
               (λs@(pr,s1,s2,c1,c2,c3,a) (pr',s1',s2',c1',c2',c3',_) →
                 (not $ stHappy a s) &&
                 (((pr == a) && (pr' /= a)) || ((s1 == a) && (s1' /= a)) ||
                  ((s2 == a) && (s2' /= a)) || ((c1 == a) && (c1' /= a)) ||
                  ((c2 == a) && (c2' /= a)) || ((c3 == a) && (c3' /= a))))
```

Modify the normative system `eta_1` to include the new condition:

```
eta_1' :: FODBR State State
eta_1' = eta_1 `union` dont_release
```

Now $\{1, 2, 3\}$ is a sufficient[4] coalition (the only minimal sufficient coalition):

```
*Ex02> let is = ((0,0,0,0,0,0,1)::State)
*Ex02> is `elem` check exampleModel' eta_1'
                                        (CS (Supseteq [1,2,3])phi_1)
True
```

[4] C is sufficient for φ in the context of K and η iff $\forall C' \subseteq A : (C \subseteq C') \Rightarrow [K \dagger (\eta \restriction C') \models \varphi]$. Note that $K \dagger (\eta \restriction C) \models \varphi$ does not in general imply that $K \dagger (\eta \restriction C') \models \varphi$ when $C \subseteq C'$ [4].

5. Conclusions

In this paper we have described NORMC, a prototype model checker for Norm Compliance CTL (NCCTL). NCCTL extends CTL with a family of modalities with update semantics. We have also aimed to demonstrate that the Haskell programming language is a natural and useful alternative for model checking; in particular that the full power of Haskell makes it easy to describe arbitrary state-transition models in a natural way. While NORMC has not been optimised for industrial use, we have seen that it is efficient enough to be used on an interesting and non-trivial example. If higher efficiency is needed, there are two natural options. The first is to extend NORMC with standard symbolic model checking techniques such as binary decision diagrams (BDDs) and partial order reduction. The other is to change NORMC into a front-end for an efficient existing CTL model checker.

Acknowledgements Piotr Kaźmierczak and Thomas Ågotnes' work was supported by the Research Council of Norway project 194521 (FORMGRID). The authors also thank Paul Simon Svanberg and anonymous reviewers of STAIRS 2012 for helpful comments.

References

[1] Ågotnes, T., W. van der Hoek, C. S. Juan A. Rodriguez-Aguilar and M. Wooldridge, *On the logic of normative systems*, in: *Proceedings of IJCAI 2007*, pp. 1175–1180.

[2] Ågotnes, T., W. van der Hoek, M. Tennenholtz and M. Wooldridge, *Power in normative systems*, in: *Proceedings of AAMAS 2009*, pp. 145–152.

[3] Ågotnes, T., W. van der Hoek and M. Wooldridge, *Normative system games*, in: *Proceedings of AAMAS 2007*, pp. 876–883.

[4] Ågotnes, T., W. van der Hoek and M. Wooldridge, *Robust normative systems and a logic of norm compliance*, Logic Journal of the IGPL **18** (2009), pp. 4–30.

[5] Cimatti, A., E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, *NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking*, in: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, LNCS **2404** (2002).

[6] Clark, E. M., E. A. Emerson and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Trans. on Progr. Lang. and Systems **8** (1986), pp. 244–263.

[7] Dellunde, P., *On the multimodal logic of normative systems*, in: *Proc. of COIN'07* (2008), pp. 261–274.

[8] Emerson, E. A., *Temporal and modal logic*, in: J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, Elsevier, 1990.

[9] Fitoussi, D. and M. Tennenholtz, *Choosing social laws for multi-agent systems: Minimality and simplicity*, Artificial Intelligence **119** (2000), pp. 61–101.

[10] Hoek, W., M. Roberts and M. Wooldridge, *Social laws in alternating time: Effectiveness, feasibility, and synthesis*, Synthese **156** (2007), pp. 1–19.

[11] Holzmann, G., *The Spin model checker*, IEEE Trans. on Software Eng. **23** (1997), pp. 279–295.

[12] Jones, S. P., "Haskell 98 language and libraries: the Revised Report," Cambridge University Press, 2003.

[13] Shoham, Y. and M. Tennenholtz, *On the synthesis of useful social laws for artificial agent societies*, in: *Proceedings of AAAI 1992*.

[14] Shoham, Y. and M. Tennenholtz, *On social laws for artificial agent societies: Off-line design*, in: *Computational Theories of Interaction and Agency*, The MIT Press: Cambridge, MA, 1996 pp. 597–618.

[15] van Eijck, J., *Dynamic epistemic modelling*, Technical report, CWI/ILLC, Amsterdam, Uil-OTS, (2005).