The Simple Normative Systems Language

Thomas Ågotnes[1], Wiebe van der Hoek[2], Juan A. Rodríguez-Aguilar[3], Carles Sierra[3], and Michael Wooldridge[2]

[1] Department of Computer Engineering, Bergen University College, Norway
[2] Department of Computer Science, University of Liverpool, UK
[3] Artificial Intelligence Research Institute IIIA, Spanish Council for Scientific Research CSIC, Spain

Abstract. Although many formalisms have been developed for reasoning about normative behaviour, most of these have been somewhat divorced from real systems. That is, while they allow the specification of what is obligatory or permissible, there is often no direct mapping to concrete computational systems, and so no way of really being able to tell (for example) whether a system respects or violates some system of norms. In this paper, we introduce the Simple Normative Systems Language, a concrete computational language for defining normative systems that has a concrete interpretation with respect to Reactive Modules, a well-known and widely used language for defining multi-agent systems. We introduce and formally define the language, briefly investigate some of its properties, and illustrate its use by means of an example.

1 Introduction

Normative systems, or social laws, have been widely promoted as an approach to coordinating multi-agent systems [9]. Crudely, a normative system defines a set of constraints on the behaviour of agents, corresponding to obligations, which may or may not be observed by agents. A number of formalisms have been proposed for reasoning about normative behaviour in multi-agent systems, typically based on deontic logic [10, 5, 8]. However the computational properties of such formalisms – in particular, their use in the practical design and synthesis of normative systems – has received relatively little attention. In this paper, we seek to rectify this omission. We introduce the *Simple Normative Systems Language* (SNL), a computational language for defining normative systems that has a concrete interpretation with respect to Reactive Modules, a well-known and widely used language for defining multi-agent systems [2, 7]. We motivate the language, introduce and formally define it, briefly investigate some of its properties, and illustrate its use by means of an example.

2 Kripke Structures and Normative Systems

We will model our multi-agent systems using *Kripke structures*, a widely used model of concurrent, distributed, and reactive systems [6, 4]. Kripke structures

describe the possible behaviours of agents in a system as transitions between possible states of the system, and abstracts away details such as the internal structure of agents. Thus, our approach is general and not specific to any particular agent or system architecture. Furthermore, this abstraction allows us to reason about normative systems in a general and abstract way, independent of the specifics of a particular normative systems architecture.

Abstractly, a Kripke structure can be thought of as a directed graph, in which nodes in the graph correspond to states of a system, and transitions in the graph correspond to actions, which cause state transitions. (As we will see later, in fact a transition corresponds to a tuple of actions, one for each agent in the system.) We label states with a description of the primitive propositions that are true in that state; these propositions provide a "description" of the state. Formally, let $\Phi = \{p, q, \ldots\}$ be a finite set of atomic *propositional variables*. Then *Kripke structure* (over Φ) is a quad

$$\mathcal{K} = \langle S, S^0, R, V \rangle,$$

where:

- S is a finite, non-empty set of *states*, with $S^0 \subseteq S$ ($S^0 \neq \emptyset$) being the *initial states* of $\mathcal{K}$;
- $R \subseteq S \times S$ is a total binary relation on S, which we refer to as the *transition relation*[4]; and
- $V : S \rightarrow 2^{\Phi}$ labels each state with the set of propositional variables true in that state.

A *path* over R is an infinite sequence of states $\pi = s_0, s_1, \ldots$ which must satisfy the property that $\forall u \in \mathbb{N}$: $(s_u, s_{u+1}) \in R$. If $u \in \mathbb{N}$, then we denote by $\pi[u]$ the component indexed by u in π (thus $\pi[0]$ denotes the first element, $\pi[1]$ the second, and so on). A path π such that $\pi[0] = s$ is an *s-path*.

Now that we have a model of systems, we can define normative systems. In this paper, a normative system should be understood simply as *a set of constraints on the behaviour of agents in a system*. More precisely, a normative system defines, for every possible system transition, whether or not that transition is considered to be legal or not, in the context of the normative system. Different normative systems may differ on whether or not a particular transition is considered legal. Formally, a normative system η (w.r.t. a Kripke structure $\mathcal{K} = \langle S, S^0, R, V \rangle$) is simply a subset of R, such that $R \setminus \eta$ is a total relation. We refer to the requirement that $R \setminus \eta$ is total as a *reasonableness* requirement: it prevents social laws which lead to states with no successor. Let $N(R) = \{\eta \mid \eta \subseteq R \ \& \ R \setminus \eta \text{ is total}\}$ be the set of normative systems over R. The intended interpretation of a normative system η is that the presence of an arc (s, s') in η means that the transition (s, s') is forbidden in the context of η, hence, $R \setminus \eta$ denotes the allowed transitions. Since it is assumed that η is reasonable, we are guaranteed that such a transition always exists for every state.

[4] Following standard usage in the branching time literature, we say a relation $R \subseteq S \times S$ is total iff $\forall s \exists s'(s, s') \in R$.

If π is a path over R and η is a normative system over R, then we say that π is η-*conformant* if it satisfies the property that $\forall u \in \mathbb{N}$, $(\pi[u], \pi[u+1]) \notin \eta$. We denote the set of η-conformant s-paths (w.r.t. some assumed R) by $\mathcal{C}_\eta(s)$.

Since normative systems in our view are just *sets* (of disallowed transitions), we can *compare* them, to determine, for example, whether one is *more liberal* (less restrictive) than another: if $\eta \subset \eta'$, then η places fewer constraints on a system than η', and hence η is more liberal. Notice that, assuming an *explicit* representation of normative systems, (i.e., representing a normative system η directly as a subset of R), checking such properties can be done in polynomial time. We can also operate on them with the standard set theoretic operations of union, intersection, etc. Taking the union of two normative systems η_1 and η_2 may yield (depending on whether $R \setminus (\eta_1 \cup \eta_2)$ is total) a normative system that is *more restrictive* (less liberal) than either of its parent systems, while taking the *intersection* of two normative systems may yield a normative system which is *less restrictive* (more liberal). The $\cup$ operation is intuitively the act of superposition, or composition of normative systems: imposing one law on top of another: care must be taken when operating on normative systems to ensure the resulting normative system is reasonable.

3 The Simple Normative Systems Language

When we program, we do not talk in terms of Kripke structures. Rather, we talk in terms of programs and programming languages. A program (or more precisely, a collection of such programs, interacting with one-another) can be thought of as defining a Kripke structure, by way of the semantics of the language. In this section, we present a language for defining systems, and (somewhat informally) its semantics in terms of Kripke structures.

The REACTIVE MODULES LANGUAGE (RML) was introduced by Alur and Henzinger as a simple but expressive formalism for specifying game-like distributed system models [2], and this language is used as the model specification language for several model checkers [3]. In this section, we consider a "stripped down" version of RML called SIMPLE REACTIVE MODULES LANGUAGE (SRML), introduced in [7]; this language represents the core of RML, with some "syntactic sugar" removed to keep the presentation (and semantics) simple.

Here is an example of an agent in SRML (note that agents are referred to as "modules" in SRML):

```
module toggle controls x
  init
  ℓ1 : ⊤ ⇝ x′ := ⊤
  ℓ2 : ⊤ ⇝ x′ := ⊥
  update
  ℓ3 : x ⇝ x′ := ⊥
  ℓ4 : (¬x) ⇝ x′ := ⊤
```

This module, named *toggle*, controls a single Boolean variable, x. The *choices* available to the agent at any given time are defined by the `init` and `update` rules[5]. The `init` rules define the choices available to the agent with respect to the initialisation of its variables, while the `update` rules define the agent's choices subsequently. In this example, there are two `init` rules and two `update` rules. The `init` rules define two choices for the initialisation of this variable: assign it the value $\top$ or the value $\bot$. The "prime" notation for variables, e.g., x', means "the value of x afterwards". Both of these rules can fire initially, as their conditions ($\top$) are always satisfied; in fact, only one of the available rules will ever *actually* fire, corresponding to the "choice made" by the agent on that decision round. On the left hand side of the rules are *labels* (ℓ_i) which are used to identify the rules. Note that labels do not form part of the original RML language, and in fact play no part in the semantics of RML – their role will become clear below. We reserve a distinguished label "`[]`", which we later will use for rules that must never be disabled. With respect to `update` rules, the first rule says that if x has the value $\top$, then the corresponding choice is to assign it the value $\bot$, while the second rules says that if x has the value $\bot$, then it can subsequently be assigned the value $\top$. In other words, the module non-deterministically chooses a value for x initially, and then on subsequent rounds toggles this value. Notice that in this example, the `init` rules of this module are non-deterministic, while the `update` rules are deterministic. An SRML *system* is a set of such modules.

Formally, a rule γ over a set of propositional variables Φ and a set of labels $\mathcal{L}$ is an expression

$$\ell : \varphi \rightsquigarrow v_1' := \psi_1; \ldots; v_k' := \psi_k$$

where $\ell \in \mathcal{L}$ is a label, φ (the guard) is a propositional logic formula over Φ, each v_i is a member of Φ and ψ_i is a propositional logic formula over Φ. We require that no variable v_i appears on the l.h.s. of two assignment statements in the same rule (hence no issue on the ordering of the updates arises). The intended interpretation is that if the formula φ evaluates to true against the interpretation corresponding to the current state of the system, then the rule is *enabled* for execution; executing the statement means evaluating each ψ_i against the current state of the system, and setting the corresponding variable v_i to the truth value obtained from evaluating ψ_i. We say that $v_1, \ldots, v_k$ are the *controlled variables* of γ, and denote this set by $ctr(\gamma)$. A set of rules is said to be *disjoint* if their controlled variables are mutually disjoint.

When dealing with the SRML representation of models, a state is simply equated with a propositional valuation (i.e., the set of states of an SRML system is exactly the set of possible valuations to variables within it: $S = 2^{|\Phi|}$). Given a state $s \subseteq \Phi$ and a rule $\gamma = \ell : \varphi \rightsquigarrow v_1' := \psi_1; \ldots; v_k' := \psi_k$ such that s enables γ (i.e., $s \models \varphi$) we denote the result of *executing* γ on s by $s \oplus \gamma$. For example, if $s = \{p, r\}$, and $\gamma = \ell : p \rightsquigarrow q' := \top; r' := p \wedge \neg r$, then $s \oplus \gamma = \{p, q\}$. Given a state $s \subseteq \Phi$, and a set Γ of disjoint rules over Φ such that every member of Γ is enabled by s, then the interpretation s' resulting from Γ on s is denoted by

[5] To be more precise, the rules are in fact *guarded commands*.

$s' = s \oplus \Gamma$ (since the members of Γ are disjoint, we can pick them in any order to execute on s).

As described above, there are two classes of rules that may be declared in an atom: `init` and `update`. An `init` rule is only used to initialise the values of variables, when the system begins execution. We will assume that the guards to `init` command are "$\top$", i.e., every `init` command is enabled for execution in the initialisation round of the system.

An SRML *module*, m, is a triple:

$$m = \langle ctr, init, update \rangle \text{ where:}$$

- $ctr \subseteq \Phi$ is the (finite) set of variables controlled by m;
- *init* is a (finite) set of *initialisation* rules, such that for all $\gamma \in init$, we have $ctr(\gamma) \subseteq ctr$; and
- *update* is a (finite) set of *update* rules, such that for all $\gamma \in update$, we have $ctr(\gamma) \subseteq ctr$.

Given a module m, we denote the controlled variables of m by $ctr(m)$, the initialisation guarded commands of m by $init(m)$, and the update guarded commands of m by $update(m)$. An SRML system ρ is then an $(n+2)$-tuple

$$\rho = \langle Ag, \Phi, m_1, \ldots, m_n \rangle$$

where $Ag = \{1, \ldots, n\}$ is a set of agents, Φ is a vocabulary of propositional variables, and for each $i \in Ag$, m_i is the corresponding module defining i's choices; we require that $\{ctr(m_1), \ldots, ctr(m_n)\}$ forms a partition of Φ (i.e., every variable in Φ is controlled by some agent, and no variable is controlled by more than one agent).

A *joint* update rule (init rule) is an indexed tuple $\langle \gamma_1, \ldots, \gamma_k \rangle$ of rules, with a rule $\gamma_i \in update(m_i)$ ($\gamma_i \in init(m_i)$) for each $i \in Ag$. A joint (update/init) rule $\langle \gamma_1, \ldots, \gamma_k \rangle$ is enabled by a propositional valuation s iff all its members are enabled by s.

It is straightforward to extract the Kripke structure $\mathcal{K}_\rho = \langle S_\rho, S^0_\rho, R_\rho, V_\rho \rangle$ corresponding to an SRML system ρ:

- the state set S_ρ and valuation function V_ρ correspond to the possible valuations of variables Φ, with initial states S^0_ρ corresponding to the valuations that could be generated by the joint init rules of ρ against the empty valuation[6]; and
- the transition relation R_ρ is defined by $(s, s') \in R_\rho$ iff there exists some joint update rule $\langle \gamma_1, \ldots, \gamma_n \rangle$ such that this joint rule is enabled in s and $s' = s \oplus \{\gamma_1, \ldots, \gamma_n\}$.

[6] More precisely, the state space corresponds to the valuations that are *reachable* from the initial states of the system.

3.1 A Concrete Language for Normative Systems

We now introduce the SRML *Norm Language*, or *Simple Normative Systems Language* (SNL), for representing normative systems, which corresponds to the SRML language for models. The general form of a normative system definition in SNL is as follows:

$$\begin{array}{l}\texttt{normative-system} \quad id \\ \quad \chi_1 \texttt{ disables } \ell_{1_1}, \ldots, \ell_{1_k} \\ \quad \cdots \\ \quad \chi_m \texttt{ disables } \ell_{m_1}, \ldots, \ell_{m_k}\end{array}$$

Here, $id \in \Sigma$ is the name of the normative system, where we assume a set Σ of identifiers. The body of the normative system is defined by a set of *constraint rules*. A constraint rule

$$\chi \texttt{ disables } \ell_1, \ldots, \ell_k$$

consists of a condition part χ, which is a propositional logic formula over the propositional variables Φ of the system, and a set of rule labels $\{\ell_1, \ldots, \ell_k\} \subseteq \mathcal{L}$. We require that none of the ℓ_i labels is the distinguished label `[]`, which is used for rules that never should be disabled. The intuition is that if χ is satisfied in a particular state, then any SRML rule with a label that appears on the r.h.s. of the constraint rule will be disabled in that state, according to this normative system. Consider the following simple example.

$$\begin{array}{l}\texttt{normative-system} \quad \textit{forceTrue} \\ \quad \top \texttt{ disables } \ell_3\end{array}$$

We here define a normative system *forceTrue*, which consists of a single rule. The condition part of the rule is $\top$, and hence always fires; in this case, the effect is to disable the rule with label ℓ_3. Since the condition part of this rule is always enabled, in the *forceTrue* normative system, rule ℓ_3 can never fire.

Formally, an SNL constraint rule is a pair

$$c = \langle \varphi, L \rangle$$

where φ is a propositional formula over Φ, and $L \subseteq \mathcal{L}$ is a set of rule labels. An SNL normative system is then a pair

$$\eta = \langle id, C \rangle$$

where $id \in \Sigma$ is a unique identifier for the normative system and C is a set of SNL constraint rules. In any given state s, the set of SRML rules that are disabled by a normative system $\langle id, C \rangle$ will be the set of rules whose labels appear on the right hand side of constraint rules in C whose condition part is satisfied in s. Let $lgl(s, \eta)$ denote the set of labels of rules that are both legal in s according to the normative system η, and that are enabled in s. A SNL *interpretation* is then simply a set of SNL normative systems, each with a distinct name.

In order to ensure that a SNL normative system $\eta = \langle id, C \rangle$ is reasonable with respect to a SRML system $\rho = \langle Ag, \Phi, m_1, \ldots, m_n \rangle$, we must impose a further restriction on η: for every state $s \in S_\rho$ and agent i, there must exist a rule in $update(m_i)$ with a label ℓ which is enabled in s and not disabled by η.

Given a SRML system ρ and a (reasonable w.r.t. ρ) SNL normative system η, it is straightforward to induce a normative system η' with respect to the Kripke structure $\mathcal{K}_\rho$:

- $(s, s') \in \eta'$ iff for every joint update rule $\langle \gamma_1, \ldots, \gamma_n \rangle$ enabled in s where $s' = s \oplus \{\gamma_1, \ldots, \gamma_n\}$, there exists a SNL constraint rule $c = \langle \varphi, L \rangle \in C$ such that $s \models \varphi$ and there is some γ_i with label ℓ such that $\ell \in L$.

Given SNL normative systems η_1 and η_2, for some SRML system ρ, we say: η_1 is *at least as liberal* as η_2 in system ρ if for every state $s \in S_\rho$, every rule that is legal according to η_2 is legal according to η_1; and they are *equivalent* if for every state $s \in S_\rho$, the set of rules legal according to η_1 and η_2 are the same.

Theorem 1. *The problem of testing whether* SNL *normative system η_1 is at least as liberal as* SNL *normative system η_2 is* PSPACE*-complete, as is the problem of testing equivalence of such systems.*

Proof. We do the proof for checking equivalence; the liberality case is similar. For membership of PSPACE, consider the complement problem: guess a state s, check that $s \in S_\rho$, (reachability of states in RML is in PSPACE) and check that there is some rule legal in s according to η_2 is not legal in s according to η_1, or vice versa. Hence the complement problem is in NPSPACE, and so the problem is in PSPACE. For PSPACE-hardness, we reduce the problem of propositional invariant checking over (S)RML modules. Given an SRML system ρ and propositional formula φ, define normative systems η_1 and η_2 as follows (where ℓ does not occur in ρ):

`normative-system` η_1	`normative-system` η_2
$\neg\varphi$ `disables` ℓ	$\bot$ `disables` ℓ

According to η_2, ℓ is always enabled; thus η_1 will be equivalent to η_2 iff φ holds across all reachable states of the system.

4 Case Study: A Traffic Controller

Consider a circular road with two parallel lanes. Vehicles circulate on the two lanes clockwise. We consider three types of vehicles: cars, taxis, and ambulances. We consider the road discretised in a finite number of positions, each position occupied by a vehicle represented as an instance of a predicate *at(lane-number,lane-position,vehicle-id)*. For instance, *at(2,5,car-23)* stands for agent *car-23* staying on lane 2 at position 5. Notice that lane 2 stands for the outer lane while lane 1 stands for the inner lane. We will refer to lane 1 as the right lane and to lane 2 as the left lane considering the direction of the vehicles. At each time step, cars and taxis can either stand still or change their position by one unit. Cars and taxis can either move straight or change lane, for instance a car can go from *at(2,5,car-23)*

to either *at(2,6,car-23)* or *at(1,6,car-23)*. Ambulances can stand still or change their position by one or two units, either straight or changing lanes at will.

To avoid crashes and permit that ambulances get faster to hospitals and that taxis have priority over private cars, we can imagine a number of norms that regulate the behaviour of the vehicles:

- η_1: Ambulances have priority over all other vehicles
- η_2: Cars cannot use the rightmost (priority) lane
- η_3: Vehicles have "right" priority.

By right priority we mean that a vehicle on the left lane has to give way to any vehicle running in parallel on the right lane intending to change to the left lane. These norms act on the decisions that agents can make by constraining them. For instance, η_1 will force cars to stop to allow ambulances to overtake them. Also, norms have relative priorities and for instance η_1 takes priority over η_2 and η_3, that is, cars might occupy the priority lane to facilitate the progress of an ambulance, or a car might give left priority to another car for the same reason.

We model each vehicle as a module containing the rules that determine their physically legal movements, and the global traffic control as a set of norms that constraint the application of certain rules. Although each module should be encoded using propositions, we employ predicates with variables over finite domains that can be easily translated into propositions. We do so to facilitate the reading of the modules.

4.1 Vehicle Modules

We define two modules standing for two types of vehicles: those with 2-unit speed (vehicle-2) and those with 1-unit speed (vehicle-1). Either vehicle module, after initialising its position, nondeterministically chooses the next position the vehicle will occupy. Note that the operations on vehicles' positions, addition and subtraction, are modulo-n operations, where n is the number of positions in the road.

```
module vehicle-1
   interface at(l,p,v):bool
   atom controls at(l,p,v)
     awaits init-at(l,p,v)
     init
     [] init-at(l,p,v) -> at(l,p,v) := T
     [] ~init-at(l,p,v) -> at(l,p,v) := T
   update
     v-straight:
      at(l,p,v) & not(at(l,p+1,z)) ->
        at(l,p+1,z) := T,
        at(l,p,v)  := F
     v-right:
```

```
      at(2,p,v) & not(at(1,p+1,z)) ->
        at(1,p+1,z) := T,
        at(2,p,v) := F
     v-left:
      at(1,p,v) & not(at(2,p+1,z)) ->
        at(2,p+1,z) := T,
        at(1,p,v) := F
    endatom
endmodule

module vehicle-2
   interface at(l,p,v):bool
   atom controls at(l,p,v)
     awaits init-at(l,p,v)
     init
     [] init-at(l,p,v) -> at(l,p,v) := T
     [] ~init-at(l,p,v) -> at(l,p,v) := T
   update
     v-straight:
      at(l,p,v) & not(at(l,p+1,z)) ->
        at(l,p+1,z) := T,
        at(l,p,v)  := F
     v-right:
      at(2,p,v) & not(at(1,p+1,z)) ->
        at(1,p+1,z) := T,
        at(2,p,v) := F
     v-left:
      at(1,p,v) & not(at(2,p+1,z)) ->
        at(2,p+1,z) := T,
        at(1,p,v) := F
     v-right-straight:
      at(2,p,v) & ~at(1,p+1,z1) & ~at(1,p+2,z2) ->
      at(l-1,p+2,v) := T , at(l,p,v) := F
     v-straight-right:
      at(2,p,v) & ~at(2,p+1,z1) and ~at(1,p+2,z2) ->
      at(l-1,p+2,v) := T , at(l,p,v) := F
     v-right-left:
      at(2,p,v) & ~at(2,p+2,z1) & ~at(1,p+1,z2) ->
       at(l,p+2,v) := T , at(l,p,v) := F
     v-straight-straight:
      at(l,p,v) & ~at(l,p+1,z1) & ~at(l,p+2,z2) ->
      at(l,p+2,v) := T , at(l,p,v) := F
     v-straight-left:
      at(1,p,v) & ~at(1,p+1,z1) & ~at(2,p+2,z2) ->
      at(l+1,p+2,v) := T , at(l,p,v) := F
    v-left-straight:
```

```
      at(1,p,v) & ~at(2,p+1,z1) & ~at(2,p+2,z2) ->
      at(l+1,p+2,z) := T , at(l,p,z) := F
    v-left-right:
      at(1,p,v) & ~at(2,p+1,z1) & ~at(1,p+2,z2) ->
      at(l,p+2,v) := T , at(l,p,v) := F
    endatom
endmodule
```

4.2 Cars, Taxis and Ambulances

Cars and taxis are vehicles of type vehicle-1. Ambulances are vehicles of type vehicle-2. Thus, we define cars and taxis by renaming the vehicle-1 module, whereas we define ambulances by renaming the vehicle-2 module. For instance, for the i-th car, for the j-th taxi and for the k-th ambulance, we would have the following definitions based on the renaming mechanism provided by MOCHA [1]:

```
c-i := vehicle-1 [v, v-right, v-left, v-straight :=
            car-i, car-i-right, car-i-left, car-i-straight]

t-j := vehicle-1 [v, v-right, v-left, v-straight :=
            taxi-j, taxi-j-right, taxi-j-left, taxi-j-straight]

a-k := vehicle-2[v, v-right, v-left, v-straight, v-right-straight,
             v-straight-right, v-right-left, v-straight-straight,
             v-straight-left, v-left-straight, v-left-right :=
             ambulance-k, ambulance-k-right, ambulance-k-left,
             ambulance-k-straight, ambulance-k-right-straight,
             ambulance-k-straight-right, ambulance-k-right-left,
             ambulance-k-straight-straight, ambulance-k-straight-left,
             ambulance-k-left-straight, ambulance-k-left-right]
```

Ambulances are vehicles of type vehicle-2 and are defined in a similar way to cars and taxis.

4.3 Controller Module

The controller module solely initialises the positions of cars, taxis, and ambulances. These vehicles wait for this initialisation to happen in order to start moving along the road.

```
module traffic_control
   interface init-at(l,p,v):bool
   atom controls init-at(l,p,v)
   init
     [] init-at(2,4,car-1) := T,
       init-at(1,1,car-1) := F,
```

```
      init-at(1,2,car-1) := F, ...
    ...
    [] init-at(2,6,taxi-1) := T,
      init-at(1,1,taxi-1) := F,
      init-at(1,2,taxi-1) := F, ...
    ...
    [] init-at(1,3,ambulance-1) := T,
      init-at(1,1,ambulance-1) := F,
      init-at(1,2,ambulance-1) := F, ...
  update
  endatom
endmodule
```

4.4 Normative Systems

The norms detailed in the beginning of Section 4 are translated into three separate normative systems as defined below. Notice that our implementation of normative system η_3 is rather conservative in the sense that we enforce a car to give priority to another car by forcing it to stand still. We adopt this view for the sake of simplicity. Otherwise, in order for a car to give way to another car with right priority, a signalling system should be used.

```
normative-system N1
  at(l,p,car-i) and at(l,p-1,ambulance-j)  disables
       car-i-straight,car-i-left,car-i-right
  at(l,p,taxi-i) and at(l,p-1,ambulance-j) disables
       taxi-i-straight,taxi-i-left,taxi-i-right

normative-system N2
  at(1,p,car-i) disables car-i-straight
  at(2,p,car-i) disables car-i-right

normative-system N3
 at(2,p,z1) and at(1,p,z2) and z1 != z2 disables
 z1-straight,z1-right
```

Properties that can be proven in this system are:

- Without norms, there might be crashes.
- If there is only one ambulance then under norm systems η_1, η_2, and η_3 there are no crashes.
- If there is more than one ambulance and there is at least one free position, then under (only) norm systems η_1 and η_2 it is permitted that two different vehicles can be at the same position on the same lane.
- If there are no ambulances, then norm system η_3 ensures that two different vehicles cannot be at the same position on the same lane.

5 Conclusions

We have introduced the Simple Normative Systems Language (SNL) as a concrete computational language in order to be able to add norms to multi-agent systems defined via the well-known Reactive Modules language. An advantage over other formalisms for normative behaviour in multi-agent systems is that the SNL language can be executed directly, as a part of a Reactive Modules specification.

The Reactive Modules is most popular as a specification language for *model checking* [4], as implemented in the MOCHA system [3]. Model checking is the problem of checking whether a given system – specified in the Reactive Modules language – has a given property – specified by a formula in a logical language. We are currently working on using logics with deontic operators for "obligation" and "permission" for model checking our normative systems, and studying the properties of such logics.

References

1. R. Alur, L. de Alfaro, T. A. Henzinger, S. C. Krishnan, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Taşiran. MOCHA user manual. University of Berkeley Report, 2000.
2. R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(11):7–48, July 1999.
3. R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Taşiran. Mocha: Modularity in model checking. In *CAV 1998: Tenth International Conference on Computer-aided Verification, (LNCS Volume 1427)*, pages 521–525. Springer-Verlag: Berlin, Germany, 1998.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.
5. F. Dignum. Autonomous agents with norms. *Artificial Intelligence and Law*, 7:69–79, 1999.
6. E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.
7. W. van der Hoek, A. Lomuscio, and M. Wooldridge. On the complexity of practical ATL model checking. In *Proc. of the Fifth Inter. Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2006)*, Hakodate, Japan, 2005.
8. A. Lomuscio and M. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.
9. Y. Shoham and M. Tennenholtz. On social laws for artificial agent societies: Offline design. In P. E. Agre and S. J. Rosenschein, editors, *Computational Theories of Interaction and Agency*, pages 597–618. The MIT Press: Cambridge, MA, 1996.
10. R. J. Wieringa and J.-J. Ch. Meyer. Deontic logic in computer science. In J.-J. Ch. Meyer and R. J. Wieringa, editors, *Deontic Logic in Computer Science — Normative System Specification*, pages 17–40. John Wiley & Sons, 1993.