FoU-rapport

Nr 30

Magnus Hjelmblom (f. Blom)

Deontic Action-Logic Multi-Agent Systems in Prolog

HÖGSKOLAN I GÄVLE

Deontic Action-Logic Multi-Agent Systems in Prolog

Magnus Hjelmblom

mbm@hig.se

Thesis work for degree of Master of Science in Computing Science, 20 credits, 2007

Supervisor: Jan Odelstad, Division of Computer Science
University of Gävle, SE-801 76 Gävle, Sweden

Reviewer: Roland Bol, Department of Information Technology
Uppsala University, SE-751 05 Uppsala, Sweden

Abstract. This work describes the process of implementing a particular kind of multi-agent system called Deontic Action-Logic-based Multi-Agent Systems, DALMAS, in Prolog. The DALMAS is regulated by a normative system which is based on the Kanger-Lindahl theory of normative positions. The algebraic model for the DALMAS is inspected and instrumentalized through an executable logic program. In particular, important issues in the transition from a set-theoretical description to a Prolog implementation are discussed. Results include a general-level Prolog implementation, which may be freely used to implement specific systems. Two such systems have already been implemented and tested, and are described and discussed here.
Keywords: multi-agent system, norm-regulated system, implementation, Prolog, normative position, deontic structure

Sammanfattning

Inom datavetenskapen brukar ordet *agent* beteckna ett datorsystem eller ett program som självständigt kan utföra handlingar och samspela med sin omgivning för att lösa problem eller uppnå vissa mål. Agentens beteende styrs ofta av en nyttofunktion som bestämmer vilka handlingar som i en given situation är mest ”önskvärda” (det vill säga medför störst ”nytta”) för agenten. Ett *multiagentsystem*, ofta förkortat MAS, är ett system av sådana agenter som agerar i relation till varandra och till sin omgivning. På senare tid har mycket forskning inom detta område gått ut på att undersöka om det är möjligt att skapa effektivare multiagentsystem genom att reglera de enskilda agenternas beteende med *normer*. DALMAS, Deontic Action-Logic Multi-Agent System, är en abstrakt arkitektur för normreglerade multiagentsystem som har utvecklats av Jan Odelstad och Magnus Boman. Syftet med det arbete som beskrivs i denna uppsats har varit att med hjälp av programmeringsspråket Prolog skapa en datorimplementation på generell nivå av den abstrakta DALMAS-arkitekturen.

Namnet DALMAS kommer från två grenar inom logiken, *handlingslogik* och *deontisk logik*. De ligger till grund för sättet att formulera och representera de normer som reglerar agenternas beteende. Handlingslogiken är ett logiskt system som inkluderar sats- och predikatlogikens grundläggande regler, samt operatorn *Do* som är en binär ”handlingsoperator”. Den avsedda tolkningen av utsagan $Do(\omega,p)$ är ”ω ser till att p (är fallet)”, där ω kan beteckna en agent och p kan beteckna något sakförhållande (till exempel att mobiltelefonen är avstängd). Den deontiska logiken inkluderar de deontiska operatorerna *Shall* och *May*, där den avsedda tolkningen av *Shall* p är ”det skall vara fallet att p” medan den avsedda tolkningen av *May* p är ”det får vara fallet att p”. Den deontiska logiken kan användas för att formulera olika normativa utsagor, till exempel $q \rightarrow Shall\ p$ som kan utläsas ”om q gäller så skall det vara fallet att p”.

Genom att kombinera handlingslogiken och den deontiska logiken skapade den svenske logikern Stig Kanger ett kraftfullt språk för att formulera normativa utsagor. Till exempel kan utsagan $q \rightarrow Shall\ Do(\omega,p)$ utläsas ”om q gäller så skall det vara fallet att ω ser till att p”. Denna idé utvecklades vidare av Lars Lindahl till en teori om *normativa positioner*. Teorin innehåller tre system av typer av normativa positioner, varav det enklaste är ett system av sju en-agenttyper av normativa positioner. Ett exempel på en sådan en-agenttyp är $\mathbf{T}_2(\omega,p)$ som betecknar utsagan $May\ Do(\omega,p) \wedge May\ Pass(\omega,p) \wedge \neg May\ Do(\omega,\neg p)$. Uttrycket $Pass(\omega,p)$ är en förkortning av $\neg Do(\omega,p) \wedge \neg Do(\omega,\neg p)$, vilket kan tolkas som att ω förhåller sig passiv till förhållandet p. Med hjälp av typerna $\mathbf{T}_1$ – $\mathbf{T}_7$ kan konditionala normer uttryckas på ett kompakt sätt. För att underlätta formuleringen av sådana normer har Lindahl och Odelstad utvecklat en algebraisk notation för normer. Denna notation utgör grunden för representationen av normativa system som reglerar agenternas beteende i normreglerade DALMASar.

En deterministisk DALMAS definieras formellt som en 9-tupel bestående av en mängd agenter, en mängd handlingsfunktioner och diverse operatorer och funktioner som ger varje specifikt system sin särskilda karaktär. Av särskilt intresse är den så kallade deontiska struktur-operatorn, som för en given situation skapar den *deontiska strukturen* för den agent som står i tur att utföra en handling. Den deontiska strukturen avgör vilka av de möjliga handlingarna för den handlande agenten som är tillåtna. I en simpel DALMAS är definitionen av den deontiska struktur-operatorn särskilt enkel:

alla handlingar som inte uttryckligen förbjuds av det normativa systemet är tillåtna. Vilka handlingar *a* som är förbjudna för agent ω i tillståndet *s* bestäms av predikatet $Prohibited_{\omega,s}(a)$, som definieras i termer av det normativa systemet.

Arbetet har resulterat i Prolog-modulen *dnrDALMAS*, en Prolog-implementation på generell nivå av DALMAS-arkitekturen. Modulen erbjuder ett antal predikat som kan användas för att skapa Prolog-implementationer av specifika DALMASar. Implementationen är uppbyggd kring ett antal Prolog-strukturer som håller reda på systemets aktuella tillstånd och dess normativa system. En central del av *dnrDALMAS*-modulen är Prolog-predikatet `prohibited/3`, som för varje norm i normsystemet testar om den aktuella normen förbjuder en viss handling för en viss agent eller inte.

Två Prolog-implementationer av specifika DALMASar, COLOUR&FORM-systemet och WASTECOLLECTOR-systemet, har skapats med hjälp av den generella implementationen. De tester som har gjorts på dessa system indikerar att *dnrDALMAS*-modulen fungerar. Detta visar i sin tur att den abstrakta arkitekturen kan utgöra grunden för körbara logikprogram, vilket ger motiv för att vidareutveckla och förfina såväl den abstrakta arkitekturen som den generella *dnrDALMAS*-modulen.

Tack

Till att börja med vill jag tacka min arbetsgivare, Högskolan i Gävle, för möjligheten att använda en del av min arbetstid för uppsatsskrivning. Magnus Boman och Jan Odelstad förtjänar stort tack för god vägledning och för den stora förmånen att ha ett gott råmaterial att arbeta med. Särskilt Jans outtröttliga arbete med att vara handledare, bollplank, mentor och inspiratör har varit helt ovärderligt. Våra diskussioner har hela tiden gett ny inspiration och ny kunskap - precis det som behövs för att skriva en uppsats! Jag vill också tacka min ämnesgranskare Roland Bol för konstruktiv kritik, värdefulla synpunkter och goda råd.

Sist men inte minst vill jag tacka Kajsa för stort tålamod, mycket uppmuntran och oändligt mycket kärlek. Kajsa, utan ditt stöd hade uppsatsen aldrig blivit klar. Tack!

Magnus Hjelmblom[1], Gävle i februari 2007

[1] **Fotnot**: Under tiden från färdigställandet av min uppsats till tryckningen av denna rapport har jag bytt efternamn från Blom till Hjelmblom.

1 Introduction

In recent years, the study of norm-regulated multi-agent systems has attracted a lot of attention; see, e.g., Boman [4], Dignum [5], Vázquez-Salceda et al. [21, 22], Wooldridge [23, pp. 213-218], and Verhagen [20]. The use of deontic logic within this area has been studied by, for instance, Jones and Sergot. [6, 17]

In [14], Odelstad and Boman introduced the notion of norm-regulated DALMAS (deontic action-logic based multi-agent system) which is an abstract architecture for a multi-agent system regulated by a normative system. A DALMAS is a global clock, global state and global dynamics system. When agent ω is to move, it chooses an act out of a set of feasible acts. The result is a new state, depending on the state of the system when the act is performed. The choice of act depends on two things:

1. which acts are the *most preferable* for the agent in the current state, according to the *preference structure* of the DALMAS; and
2. which acts are *permissible* in the current state, according to the *deontic structure* of the DALMAS.

The deontic structure is defined in terms of a normative system which regulates the DALMAS. An act is permissible if it does not lead to a state which satisfies a condition that is forbidden according to the normative system. The representation of norms is based on an algebraic representation of normative systems [9, 10] based on the Kanger-Lindahl theory of normative positions [8].

The chief aim of this work is to show that the theory of DALMASes may be instrumentalized through executable programs; or, in other words, to show that the DALMAS architecture may be implemented in software. For this purpose, a general-level implementation of the abstract architecture will be created in Prolog. The implementation is called *dnrDALMAS*, which stands for *deterministic norm-regulated DALMAS*. It will in turn be used to implement two specific DALMASes. The Prolog code for *dnrDALMAS* and the two specific DALMAS implementations is available in electronic form.

The abstract architecture for DALMAS will be outlined in section 5. Some issues when going from a set-theoretically defined architecture to a Prolog implementation will be addressed in section 6, and an overview of the general-level Prolog implementation is given in section 7. Two specific DALMASes, the COLOUR&FORM system and the WASTE-COLLECTORS system, will be introduced in sections 2.1 and 2.2. These systems will be used as running examples to illustrate and clarify some of the ideas in the abstract architecture and their representation in the Prolog implementation. The implementations of these specific DALMASes will also be used for testing purposes, to verify the correctness of the general-level implementation.

2 Multi-Agent Systems

According to [23], an agent is a "computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives". Russel and Norvig [16] stress that an agent can perceive its environment through *sensors*, and act upon its environment through *effectors*. The properties of environment for agents may vary considerably. Russel and Norvig classify these properties in five categories [16, p. 46]:

1. *Accessible* vs. *inaccessible* environment: In an accessible environment, the agent can obtain complete and accurate information about the state of the environment through its sensors.
2. *Deterministic* vs. *non-deterministic* environment: A deterministic environment is one where the next state of the environment is completely determined by the current state and the actions selected by the agent.
3. *Episodic* vs. *non-episodic* environment: An environment episodic is an environment in which the agent's experience can be divided into "episodes" consisting of the agent perceiving and acting, and where the quality of action only depends on the current episode.
4. *Static* vs. *dynamic* environment: If the environment only changes as a result of the actions performed by the agent, then it is static. This means that the environment remains unchanged while the agent is considering its next move. In a dynamic environment, other processes operate simultaneously on the environment, which means that the environment may change as time passes and beyond the agent's control. A *semi-dynamic* environment does not change by the passage of time, but the agent's performance score does.
5. *Discrete* vs. *continuous* environment: In a discrete environment there are a finite number of distinct and clearly defined percepts and actions.

It is natural to distinguish between hardware agents (embodied agents or robots) and software agents (simulated agents or softbots) based on the type of environment that the agent is situated in and the type of its sensors and effectors.

Four categories of agents are discussed in [16, pp. 40ff]: *Reflex* agents, *Model-based* agents, *Goal-based* agents and *Utility-based* agents. A simple reflex agent perceives properties of its environment, and reacts to the perceptions, but does not have a "memory" that keeps track of earlier percepts or actions. In a model-based architecture, the agent constructs an internal model of the environment through the perceptions of the world and the knowledge about how the agent's actions affect its environment. In addition, goal-based agents have *goals* that describe a desirable state for the agent and mechanisms (such as planning) for choosing actions that lead to these goals. For utility-based agents, different states may be more or less desirable, which is indicated by the *utility* of states. The utility is computed by some *utility function* which maps states to a utility value, e.g. a real number, representing the "happiness" of the agent is in a particular state. A common architecture for such agents is the so-called *Belief-Desire-Intention* (BDI) model [23], but many other architectures and models exist.

A multi-agent system (abbreviated MAS) is a system of autonomous agents acting in relation to each other and to the environment. Luger and Stubblefield [11] give four criteria for intelligent MAS:

1. *Situated*: The agents are situated in an environment and capable of perceiving information from the environment and acting upon it.
2. *Autonomous*: The agents are capable of autonomous interaction with the environment.
3. *Flexible*: The agents are intelligent and receptive to change in the environment.
4. *Social*: The agents can communicate with and cooperate and/or negotiate with other agents.

Many applications of multi-agent systems have been suggested and investigated, for instance e-commerce agents, network security agents, agent-based simulation and the study of artificial social systems. An interesting area within this field is the study of norm-regulated multi-agent systems. One motivation for this is that formally represented norms may be a useful tool to create rules for the behaviour of different types of multi-agent systems. On the other hand, norm-regulated multi-agent systems may also be used for simulation of artificial societies, making it possible to implement and evaluate different normative systems.

2.1 The COLOUR&FORM system

This very simple MAS is situated in an accessible, deterministic, static and discrete environment. The system consists of only two agents called *chroma* and *forma*. The agents have two attributes, *colour* ("black" or "white") and *form* ("circle" or "square"). The values of these attributes for each agent represent a state of the system.

The agents can choose between two (always feasible) acts: *change colour* and *change form*. The utility function for the system is simple: it is defined in such a way that agent *chroma* prefers to change colour, and agent *forma* prefers to change colour.

A COLOUR & FORM DALMAS will be specified in section 5.3.4. The behaviour of the agents will be regulated by a normative system (see section 5.4) which will be described and formally defined in section 5.4.4.

2.2 The WASTE-COLLECTORS system

The WASTE-COLLECTORS system was used in [14] as an example, inspired by [19]. A brief description will be given here.

The WASTE-COLLECTORS system is a system of agents operating in an accessible, deterministic, static and discrete environment which we will call the "wasteland". This environment consists of a grid of squares ordered in rows and columns, making it possible to assign a coordinate consisting of an ordered pair of integers to each square. Some squares contain an amount of waste, represented by a number. An agent in the system (a "waste-collector" or "waste-agent") tries, in cooperation with other agents, to collect as much waste as possible. Therefore, each waste-collector has a utility function, designed in such a way that the utility depends on the amount of waste that is collected. The agents can move one square at a time in four directions: up/north, down/south, left/west and right/east. The agents may also pick up waste and pass, that is, do nothing. Note that choosing action *pass* is not necessarily the same as being "passive" (see section 4) with respect to some condition. To summarize, an agent may choose one of the following actions: go_{north}, go_{south}, go_{west}, go_{east}, *grab* or *pass*. In section 5.3.5, a WASTE-COLLECTOR DALMAS will be formally defined.

There are some restrictions on how the waste-collector agents may act, especially on how they may move near other agents. These restrictions will be expressed in terms of a normative system which regulates the behaviour of the waste-collectors. A normative system for the waste-collectors will be described and formally defined in section 5.4.5.

3 Deontic Action-Logic

The term *action-logic* will be used in this section to denote a logical system which includes the rules of propositional and predicate logic and elementary set theory, together with rules for the binary action operator *Do*. The intended interpretation of *Do(x, p)* is "*x* sees to it that *p*". Lindahl postulates the following rules for *Do*[2]: [8]

1. If $p \Leftrightarrow q$, then $Do(x, p) \Leftrightarrow Do(x, q)$.
2. $Do(x, p) \Rightarrow p$.

These rules give some precision to the rather vague concept "sees to it that". Still, as Lindahl points out, the rules allow the concept to retain "a certain vagueness, and there is still room left for further precision in other ways" ([8], p. 68). A more precise interpretation of *Do* is the basis for the DALMAS architecture, as we will see in the following.

The fundamental ideas of *deontic logic* can be traced back to ancient Greece. In modern time, a lot of research within this area has followed a pioneer work by von Wright published in 1951. Deontic logic may be used to create normative sentences. Basically, two kinds of normative sentences exist: *purely normative sentences* and *conditional normative sentences*. A purely normative sentence is constructed by applying a norm-building operator to a descriptive sentence, which is a sentence expressing some fact about the state of the world. Examples of such norm-building operators are the deontic operators *Shall* and *May*. The intended interpretation of *Shall* is "it shall be that" ("it shall be the case that"), and the intended interpretation of *May* is "it may be that". An example of a purely normative sentence is *Shall s*, which is interpreted as "it shall be that *s*". A conditional normative sentence consists of a combination of a descriptive sentence and a purely normative sentence. A typical

[2] $\Rightarrow$ denotes the relation of logical consequence, and $\Leftrightarrow$ denotes the relation of logical equivalence.

conditional norm has the form $p \rightarrow Shall\ s$. The interpretation of this sentence would be "if p, then it shall be that s".

An important contribution to deontic logic was made by Stig Kanger, who combined it with action-logic. He assumed the following logical postulates for *Shall*:

3. If $p \Rightarrow q$, then $Shall\ p \Rightarrow Shall\ q$.
4. $(Shall\ p \wedge Shall\ q) \Rightarrow Shall(p \wedge q)$.
5. $Shall\ p \Rightarrow \neg Shall\ \neg p$.

To facilitate the presentation, we define the sentence *May p*:

$$May\ p \text{ iff } \neg Shall \neg p$$

Kanger's idea was then to combine the deontic operator *Shall* with the action operator *Do*. [7] The intended interpretation of *Shall Do(x, q)* is "it shall be the case that x sees to it that q", while the intended interpretation of $\neg Shall\ Do(x, \neg q)$ is "it is not the case that y shall see to it that not q". From the above definition of *May* it follows that

$$May\ Do(x, q) \text{ iff } \neg Shall \neg Do(x, q)$$

which is interpreted as "it may be the case that x sees to it that q".

The result of combining *Shall* or *May*, *Do* and $\neg$ is a powerful language for expressing normative sentences. A conditional norm formulated in this way may for example have the following form:

$$c(x,y) \rightarrow Shall\ Do(x, \neg d(y))$$

According to the rules for *Shall* and *Do*, this means that "if $c(x,y)$, then it shall be the case that x sees to it that not $d(y)$".

The language of deontic action-logic was used by Kanger as a basis for a theory of normative positions. This theory was further developed by Lindahl in [8]. We will return to this contribution in section 4.

3.3 Deontic Logic and Multi-Agent Systems

Many researchers have explored the use of deontic logic within the design of multi-agent systems. For instance, the use of Constraint Handling Rules (CHR) to express deontic constraints has been explored within the area of agent communication. [2] Another example is IMPACT [3], an agent platform where deontic operators of permission, obligation and prohibition is the basis for the specification of what an agent is obliged to do, may do or cannot do. The IMPACT platform uses the concept of a *feasible status set*. This set is provided by agent programs defining integrity constraints which are to be satisfied. If the constraints are not satisfied, the state of an agent is corrupted. A third example is Norman-G [17], a Prolog program developed by Sergot. Norman-G is based on the theory of normative positions, which will be presented in the next section.

4 Normative Positions

The Kanger-Lindahl theory of normative positions is based on Kanger's idea of combining the deontic operator *Shall* with the action operator *Do*, as described in the previous section. The theory contains three systems of types of normative positions. The simplest of these systems is a system of seven one-agent types of normative positions, based on the logic of *Shall* and *Do*. [8] The *i*:th type of *one-agent positions*, where $1 \le i \le 7$, is denoted $\mathbf{T}_i$.

In the following, the sentence $\neg Do(\omega, d) \wedge \neg Do(\omega, \neg d)$ will be abbreviated $Pass(\omega, d)$. The interpretation of this sentence is that "it is not the case that ω sees to it that d, and it is not the case that ω sees to it that not d. Lindahl proposes that $Pass(\omega, d)$ expresses ω's passivity with respect to d. (cf. [10, p. 74])

It is possible to make a list of all maximal and consistent conjunctions such that each conjunct satisfies the scheme $\pm May \pm Do(\omega, \pm d)$, where $\pm d$ stands for either d or $\neg d$ and *May* is defined as in section 3. Using $Pass(\omega, d)$ instead of $\neg Do(\omega, d) \wedge \neg Do(\omega, \neg d)$, the following list is obtained (see for example [8, p. 92] or [10, p. 75]):

$\mathbf{T}_1(\omega,d)$: $May\ Do(\omega, d) \wedge May\ Pass(\omega, d) \wedge May\ Do(\omega, \neg d)$
$\mathbf{T}_2(\omega,d)$: $May\ Do(\omega, d) \wedge May\ Pass(\omega, d) \wedge \neg May\ Do(\omega, \neg d)$
$\mathbf{T}_3(\omega,d)$: $May\ Do(\omega, d) \wedge \neg May\ Pass(\omega, d) \wedge May\ Do(\omega, \neg d)$
$\mathbf{T}_4(\omega,d)$: $\neg May\ Do(\omega, d) \wedge May\ Pass(\omega, d) \wedge May\ Do(\omega, \neg d)$
$\mathbf{T}_5(\omega,d)$: $May\ Do(\omega, d) \wedge \neg May\ Pass(\omega, d) \wedge \neg May\ Do(\omega, \neg d)$
$\mathbf{T}_6(\omega,d)$: $\neg May\ Do(\omega, d) \wedge May\ Pass(\omega, d) \wedge \neg May\ Do(\omega, \neg d)$
$\mathbf{T}_7(\omega,d)$: $\neg May\ Do(\omega, d) \wedge \neg May\ Pass(\omega, d) \wedge May\ Do(\omega, \neg d)$

For example, $\mathbf{T}_2(\omega,d)$ denotes the deontic action-logic sentence $May\ Do(\omega, d) \wedge May(\neg Do(\omega, d)\ \&\ \neg Do(\omega, \neg d)) \wedge \neg May\ Do(\omega, \neg d)$. Thus, the intended meaning of $\mathbf{T}_2(\omega,d)$ is that "ω may see to it that d will be the case, and ω may be passive with respect to d, and ω may not see to it that d will not be the case". Note that $\mathbf{T}_5(\omega,d)$ simply means $Shall\ Do(\omega, d)$, while $\mathbf{T}_6(\omega,d)$ means $Shall\ Pass(\omega, d)$ and $\mathbf{T}_7(\omega,d)$ means $Shall\ Do(\omega, \neg d)$.

The types $\mathbf{T}_1 - \mathbf{T}_7$ may be used as operators on descriptive conditions to get deontic conditions. If, as an example, d is a binary condition, then $T_i d$ $(1 \le i \le 7)$ is the ternary condition such that

$$T_i d(y,z,x) \text{ iff } \mathbf{T}_i(x, d(y,z))$$

where $\mathbf{T}_i(x, d(y,z))$ is the *i*:th formula of one-agent normative positions. We will return to these operators in section 5.2.3.

Many applications of the theory of normative positions have been suggested, for example within such areas as the formal representation of laws and legal contracts, the analysis of responsibility, authorization, delegation and similar notions, agent-oriented programming and agent communication languages. The theory of has been further developed by Jones and Sergot [6, 17], and its applications within computer science have been explored. The application of the theory of normative positions to normative systems has been further investigated by for example Lindahl and Odelstad. [9, 10]

5 Abstract Architecture for a Deterministic Norm-Regulated DALMAS

In this section, the abstract architecture for DALMAS will be outlined. For a complete description, see [14]. A DALMAS is situated in an accessible, deterministic, static and discrete environment. Odelstad and Boman give the following formal definition of a norm-regulated DALMAS [14, p. 158]:

*A norm-regulated DALMAS is a system [**D**,**N**] where **D** is a DALMAS and **N** is a normative system for **D**.*

Before we study the definitions of ***D*** (section 5.3) and ***N*** (section 5.4) we will examine two kinds of conditions that may be true or false of a number of agents in a multi-agent system: *state-conditions* and *sit-conditions*. [14, pp. 155f]

5.1 States and state-conditions

A *state* is a state of affairs holding for the DALMAS at a particular time. A state may be viewed as a state of the global *knowledge base* (see for instance [11]) shared by all agents in the system.

A ν-ary state-condition c is true or false of ν agents $\omega_1, \dots, \omega_\nu$ in a state s. This is written $c(\omega_1, \dots, \omega_\nu; s)$, where semicolon separates the agent arguments from the state argument.

5.1.1 Examples

As an example, let us define the condition *Diff* such that $Diff(\omega_1,\omega_2; s)$ iff $\omega_1 \neq \omega_2$. This means that *Diff* is true if and only if the two agent arguments refer to different agents, regardless of the state of the system.

For the COLOUR&FORM world (section 2.1), let us define the state-condition $Eq(\omega_1,\omega_2; s)$ with the intended meaning that agents ω_1 and ω_2 have identical attributes in state s.

Similarly, let us define a state-condition Lap_i for the WASTE-COLLECTORS world introduced in section 2.2. The intended meaning is that $Lap_i(\omega_1,\omega_2; s)$ holds if and only if the "protected spheres" of agents ω_1 and ω_2 overlap with i squares, where i is an integer parameter such that $0 \leq i \leq 9$. The *protected sphere* [14, p. 143] around an agent *ω* is defined as the von Neumann neighbourhood of *ω*, that is, it consists of the nine squares around *ω* with *ω* in the middle:

Fig. 1. Overlap of the protected spheres for agents ω_1, ω_2 and ω_3

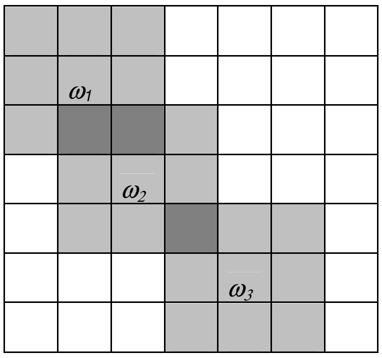


Figure 1 illustrates a state *s* where $Lap_2(\omega_1,\omega_2; s)$ holds, since the protected spheres of ω_1 and ω_2 overlap with two squares. Similarly, we have $Lap_0(\omega_1,\omega_3; s)$ and $Lap_1(\omega_2,\omega_3; s)$, since the protected spheres of ω_2 and ω_3 overlap with one square while the protected spheres of ω_1 and ω_3 do not overlap at all. We may note that the protected spheres of two agents can overlap with 0, 1, 2, 3, 4, 6 or 9 squares. Table 1 summarizes the possible changes of condition when the moving agent ω_m performs one of the six possible acts listed in section 2.2:

Table 1. Possible changes of condition

Condition	**Possible conditions in the next state**
$Lap_0(\omega_m,\omega_i)$	$Lap_0(\omega_m,\omega_i)$, $Lap_1(\omega_m,\omega_i)$, $Lap_2(\omega_m,\omega_i)$, $Lap_3(\omega_m,\omega_i)$
$Lap_1(\omega_m,\omega_i)$	$Lap_0(\omega_m,\omega_i)$, $Lap_1(\omega_m,\omega_i)$, $Lap_2(\omega_m,\omega_i)$
$Lap_2(\omega_m,\omega_i)$	$Lap_0(\omega_m,\omega_i)$, $Lap_1(\omega_m,\omega_i)$, $Lap_2(\omega_m,\omega_i)$, $Lap_3(\omega_m,\omega_i)$, $Lap_4(\omega_m,\omega_i)$
$Lap_3(\omega_m,\omega_i)$	$Lap_0(\omega_m,\omega_i)$, $Lap_2(\omega_m,\omega_i)$, $Lap_3(\omega_m,\omega_i)$, $Lap_6(\omega_m,\omega_i)$
$Lap_4(\omega_m,\omega_i)$	$Lap_2(\omega_m,\omega_i)$, $Lap_4(\omega_m,\omega_i)$, $Lap_6(\omega_m,\omega_i)$
$Lap_6(\omega_m,\omega_i)$	$Lap_3(\omega_m,\omega_i)$, $Lap_4(\omega_m,\omega_i)$, $Lap_6(\omega_m,\omega_i)$, $Lap_9(\omega_m,\omega_i)$
$Lap_9(\omega_m,\omega_i)$	$Lap_6(\omega_m,\omega_i)$, $Lap_9(\omega_m,\omega_i)$

The state-conditions presented in this section will later be used to form normative systems for the COLOUR&FORM system and the WASTE-COLLECTORS system.

5.2 Situations and sit-conditions

A *situation* for a DALMAS is represented by an ordered pair <*ω*,*s*>, meaning that *s* is the state of the system and agent *ω* is to move.

A ν-ary sit-condition *d* is true or false of ν agents $\omega_1, \ldots, \omega_\nu$ in a situation <*ω*,*s*>. This is written $d(\omega_1, \ldots, \omega_\nu; \omega, s)$, where semicolon separates the agents from the situation. Sit-conditions may be formed by applying special operators to state-conditions. The abstract architecture describes the *move*-operator and the *type*-operators. The introduction and definition of the *move*-operator was motivated by the algebraic representation of norms as ordered pairs of conditions, which will be described in section 5.4.

5.2.1 The move-operator M

The move-operator M transforms a ν-ary state-condition $c(\omega_1, \dots, \omega_\nu; s)$ to a ν+1-ary sit-condition $Mc(\omega_1, \dots, \omega_\nu, \omega_{\nu+1}; \omega,s)$. It is defined in the following way [14, p. 156]:

$$Mc(\omega_1, \dots, \omega_\nu, \omega_{\nu+1}; \omega,s) \text{ iff } \omega_{\nu+1} = \omega \wedge c(\omega_1, \dots, \omega_\nu; s)$$

The intended interpretation of Mc is that $c(\omega_1, \dots, \omega_\nu; s)$ holds, and $\omega_{\nu+1}$ is to move in situation $<\omega,s>$. As an example, let us apply this operator to the state-conditions in section 5.1.1. We note that $MDiff(\omega_1,\omega_2,\omega_3; \omega,s)$ would mean that $\omega_1 \neq \omega_2$ and ω_3 is to move in situation $<\omega,s>$. Similarly, $MEq(\omega_1,\omega_2,\omega_3; \omega,s)$ means that ω_1 and ω_2 have identical attributes and ω_3 is to move, and $MLap_i(\omega_1,\omega_2,\omega_3; \omega,s)$ means that the protected spheres of ω_1 and ω_2 overlap with i squares, and ω_3 is to move.

5.2.2 The move-operator M_n

Recent studies [15] have discovered that the Move-operator as defined in [14] does not give sufficient expressiveness to allow the formulation of an important class of norms. This issue has been discussed with the authors of the DALMAS architecture [14], who have suggested an extension of the abstract architecture: the M_n-operator. This operator is very similar to the original M-operator, but it also unifies the acting agent with one of the agent arguments of the sit-condition that the operator is applied to:

$$M_nc(\omega_1, \dots, \omega_\nu, \omega_{\nu+1}; \omega,s) \text{ iff } \omega_{\nu+1} = \omega \wedge c(\omega_1, \dots, \omega_\nu; s) \wedge \omega_{\nu+1} = \omega_n$$

where ω_n is the n:th agent argument of c.

For example, if $M_1Eq(\omega_1,\omega_2,\omega_3; \omega,s)$ holds, then ω_1 and ω_2 have identical attributes, and ω_3 is to move, and $\omega_3 = \omega_1$. Similarly, $M_2Lap_2(\omega_1,\omega_2,\omega_3; \omega,s)$ means that the protected spheres of ω_1 and ω_2 overlap with two squares, and ω_3 is to move, and $\omega_3 = \omega_2$.

5.2.3 The type-operator T_i

The type-operator T_i, $1 \leq i \leq 7$, transforms a ν-ary state-condition $d(\omega_1, \dots, \omega_\nu; s)$ to a ν+1-ary sit-condition $T_id(\omega_1, \dots, \omega_\nu, \omega_{\nu+1}; \omega,s)$, which is defined in the following way:

$$T_id(\omega_1, \dots, \omega_\nu, \omega_{\nu+1}; \omega,s) \text{ iff } \mathbf{T}_i(\omega_{\nu+1}, d(\omega_1, \dots, \omega_\nu; s^+))$$

where s^+ is the state which will be the result of the action taken by agent ω_{v+1} in situation $<\omega,s>$, and $\mathbf{T}_i$ is the *i*:th type of the one-agent positions (see section 4). For example, if $i = 5$, the intended interpretation of $T_5Lap_1(\omega_1, \omega_2, \omega_3; \omega,s)$ is

$$T_5Lap_1(\omega_1, \omega_2, \omega_3; \omega,s) \text{ iff } \mathbf{T}_5(\omega_3, Lap_1(\omega_1, \omega_2; s^+)) \text{ iff}$$
$$May\ Do(\omega_3, Lap_1(\omega_1, \omega_2; s^+)) \wedge \neg May\ Pass(\omega_3, Lap_1(\omega_1, \omega_2; s^+))$$
$$\wedge \neg May\ Do(\omega_3, \neg Lap_1(\omega_1, \omega_2; s^+))$$

The type-operators are central for the definition of a normative system (section 5.4) for a norm-regulated DALMAS. To facilitate the use of the type-operators, six new operators E_i^a, $2 \leq i \leq 7$ are defined. In conjunction with the corresponding T_i, these operators will regulate if act *a* is prohibited or not:

2. $E_2{}^a d(\omega_1,\ldots,\omega_v,\omega_{v+1}; \omega,s)$ iff $[d(\omega_1,\ldots,\omega_v; s) \wedge \neg d(\omega_1,\ldots,\omega_v; s^+)]$
3. $E_3{}^a d(\omega_1,\ldots,\omega_v,\omega_{v+1}; \omega,s)$ iff $[d(\omega_1,\ldots,\omega_v; s) \leftrightarrow d(\omega_1,\ldots,\omega_v; s^+)]$
4. $E_4{}^a d(\omega_1,\ldots,\omega_v,\omega_{v+1}; \omega,s)$ iff $[\neg d(\omega_1,\ldots,\omega_v; s) \wedge d(\omega_1,\ldots,\omega_v; s^+)]$
5. $E_5{}^a d(\omega_1,\ldots,\omega_v,\omega_{v+1}; \omega,s)$ iff $[\neg d(\omega_1,\ldots,\omega_v; s^+)]$
6. $E_6{}^a d(\omega_1,\ldots,\omega_v,\omega_{v+1}; \omega,s)$ iff $\neg[d(\omega_1,\ldots,\omega_v; s) \leftrightarrow d(\omega_1,\ldots,\omega_v; s^+)]$
7. $E_7{}^a d(\omega_1,\ldots,\omega_v,\omega_{v+1}; \omega,s)$ iff $[d(\omega_1,\ldots,\omega_v; s^+)]$

where $s^+ = a(\omega_{v+1},s)$. The purpose of these operators[3] is to eliminate the need to fall back on the definition of the corresponding $\mathbf{T}_i$. Since T_1 does not impose any restrictions on ω's actions, there is no need for a corresponding $E_1{}^a$ operator.

Note that by definition s^+ is the resulting state when ω_{v+1} performs act *a* in situation $<\omega,s>$. Here it is tempting to assume that ω_{v+1} and ω must be the same agent. After all, how would it be possible for an agent other than the moving agent ω to perform an act leading to state s^+, thereby seeing to it that s^+? The answer is that the distinction between ω_{v+1} and ω is made in order to prepare for future development of the framework, for example allowing other agents than the moving agent to perform "reaction acts". This would require the introduction of other operators than the move-operators. But for all norms that use the move-operators *M* or M_i, it is in fact the case that $\omega_{v+1} = \omega$, due to the definitions of these operators. The move-operators identify ω_{v+1} with the moving agent ω. Therefore, it is the moving player ω who sees to it that some condition is the case in s^+.

To illustrate, let us consider the intended meaning of $E_5{}^a$. We note that $E_5{}^a Lap_1(\omega_1, \omega_2, \omega_3; \omega,s)$ is true if and only if $\neg Lap_1(\omega_1, \omega_2; s^+)$ holds, where s^+ is the resulting state when ω_3 performs act *a* in situation $<\omega,s>$. That is, $E_5{}^a Lap_1(\omega_1, \omega_2, \omega_3; \omega,s)$ means that ω_3 sees to it that $\neg Lap_1(\omega_1, \omega_2; s^+)$ by performing act *a*. This is either done "actively", if act *a* changes the state from Lap_1 in state *s* to $\neg Lap_1$ in s^+, or "passively", if $\neg Lap_1$ already holds in state *s* and *a* does not change this in s^+. Compare with $E_2{}^a Lap_1(\omega_1, \omega_2, \omega_3; \omega,s)$, which is true if and only if $Lap_1(\omega_1, \omega_2; s)$ and $\neg Lap_1(\omega_1, \omega_2; s^+)$. The meaning of $E_2{}^a Lap_1$ is that ω_3 "actively" sees to it that $\neg Lap_1$ by performing an act *a*, which changes the state from Lap_1 in *s* to $\neg Lap_1$ in s^+.

[3] The operators may be called *act position operators*, using the terminology in [17].

For a deeper discussion and motivation of the E_i operators and their definition, see [14, p. 160ff].

5.3 Deterministic DALMAS and Simple Deterministic DALMAS

A deterministic DALMAS ***D*** is defined in [14, p. 152f] as an ordered 9-tuple $\langle\Omega, A, \mathrm{S}, A_f, \Delta, \Pi, \Gamma, \tau, \gamma\rangle$, where the arguments are specific sets, operators and functions which define the DALMAS. +In this presentation, mnemonic names will be used instead of symbols. Using these names, the deterministic DALMAS is defined as an ordered 9-tuple $\langle\Omega$, *A*, *S*, *FeasibleActs*, *DeonticStructure*, *PreferenceStructure*, *ChoiceSet*, *NextToMove*, *BreakTie*$\rangle$. Let us examine the arguments of this 9-tuple in more detail:

1. Ω is the *set of agents* in ***D***.
2. *A* is an *action set*, such that an element *a* in *A* is a function a: $\Omega\times S \rightarrow S$. The interpretation of $a(\omega,s) = s^+$ is that if agent ω performs act *a* in state *s*, then the resulting state will be s^+.
3. *S* is the *state space* of ***D***. The state space is the set of all states that may be reached when the agents perform feasible actions. Since each action in the action set *A* defines a transformation rule from one state to another, the state space is implicitly defined by the action set and the *FeasibleAct* function (see below).
4. *FeasibleActs*: $\Omega\times S \rightarrow \wp(A)$[4] is a function such that *FeasibleActs*(ω,s) is *the set of feasible acts* for agent ω in state *s*. In [14], the notation for this function is a calligraphic *A*. It may be defined in the following way: *FeasibleActs*$(\omega,s) = \{a\in A$: *Feasible*$_{\omega,s}(a)\}$, where *Feasible*$_{\omega,s}(a)$ is true if and only if *a* is a feasible (accessible) act for agent ω in state *s*.
5. *DeonticStructure*: $\Omega\times S \rightarrow \mathrm{D}$ is a deontic structure-operator, such that *DeonticStructure*(ω,s) is ω's *deontic structure* on *FeasibleActs*(ω,s) in state *s*. D is a set of deontic structures of the same type, with subsets of *A* as domains. In [14], the notation for the deontic structure-operator is Δ. A *simple* DALMAS uses the following simplified version of *DeonticStructure*: *DeonticStructure*$(\omega,s) \subseteq$ *FeasibleActs*(ω,s), where *DeonticStructure*(ω,s) = *Permissible*$_{\omega,s}$ = $\{a\in$ *FeasibleActs*(ω,s): *Permissible*$_{\omega,s}(a)\}$. In other words, if ***D*** is a simple DALMAS, then the deontic structure for ω in state *s* is the set of permissible actions for ω in state *s*. We will return to the definition of *Permissible*$_{\omega,s}(a)$ in the following subsection.
6. *PreferenceStructure*: $\Omega\times S \rightarrow \mathrm{P}$ is an operator such that *PreferenceStructure*(ω,s) is ω's *preference structure* on *FeasibleActs*(ω,s) in state *s*, where P is a set of preference structures of the same type, with subsets of *A* as domains. A preference structure is an ordering of the acts in *FeasibleActs*(ω,s) according to the "utility" of the acts for the agent. The preference structure-operator is denoted Π in [14]. In a *simple* DALMAS, *PreferenceStructure*$(\omega,s) = \langle$*FeasibleActs*$(\omega,s), W\rangle$ where *W* is a *weak ordering*[5]. This weak ordering may be defined in terms of a "utility function" U_ω: $S \rightarrow \mathrm{M}$ in the following way: $a\ W_{\omega,s}\ b$ iff $U_\omega(a(\omega,s)) \geq U_\omega(b(\omega,s))$. M may be

[4] $\wp(A)$ denotes the *power set* of *A*.
[5] For a definition of *weak ordering*, see [14, p. 153].

any set for which the $\geq$ relation is defined, such as the set of real numbers or the set of natural numbers.

7. *ChoiceSet*: $\Omega \times S \rightarrow \wp(A)$ is a function such that *ChoiceSet*(ω,s) is *the set of actions for ω to choose from* in state s, ordered according to the preference structure. The symbol used in [14] for the choice-set function is Γ. In a *simple* DALMAS, *ChoiceSet*(ω,s) =
   $\{x \in$ *DeonticStructure*(ω,s): $\forall y \in$ *DeonticStructure*(ω,s): $x\ W\ y\}$, which means that the choice-set consists of the most preferred of the permissible actions.
8. *NextToMove*: $\Omega \times S \rightarrow \Omega$ is a turn-operator such that *NextToMove*$(\omega_1) = \omega_2$ means that ω_2 is to move after ω_1. This operator is denoted τ in [14].
9. *BreakTie*: $\wp(A) \rightarrow A$ is a tie-breaking function, where *BreakTie*$(\{a_1, \ldots, a_n\}) = a$ means that a is *the act to choose* out of a set of permissible and equally preferred actions. In [14], this function is denoted γ.

5.3.1 Simple DALMAS

Note that a *simple* DALMAS is a DALMAS with simple versions of *DeonticStructure*, *PreferenceStructure* and *ChoiceSet* [14, p. 153], as described above. When we define a specific simple deterministic DALMAS, we provide definitions for the agent set Ω, the set of action functions A, the "feasibility predicate" *Feasible*, the utility function U_ω, the turn-operator *NextToMove* and the tie-breaker *BreakTie*.

The deontic structure operator is particularly interesting. According to our definition, the deontic structure in a simple deterministic DALMAS consists of all permissible acts. For a norm-regulated DALMAS, the idea is to let a normative system decide which acts are permissible. A very natural interpretation is that an act is permissible if it is not explicitly prohibited by the normative system ***N***:

$$\mathit{Permissible}_{\omega,s}(a) \text{ iff } \neg \mathit{Prohibited}_{\omega,s}(a)$$

where $\mathit{Prohibited}_{\omega,s}(a)$ means that act a is prohibited for agent ω in state s, according to ***N***. However, this is not the only possible interpretation. For instance, we could say that an act is prohibited if it is not explicitly permissible according to ***N***. We will return to the definition of *Prohibited* in section 5.5.

5.3.2 Primary operators and functions

The elements a of the action set A, the "feasibility predicate" *Feasible*, the utility function U_ω, the turn-operator *NextToMove* and the tie-breaker *BreakTie* may be regarded as *primary* operators/functions in the DALMAS architecture. This means that they are not defined in terms of a formula containing other operators or functions in the theory. An interpretation is assigned to them when a specific DALMAS is defined, that is, when a model for the theory of DALMASs is constructed. Together with the agent set Ω, the primary operators and functions give each specific system their unique properties and behaviours.

5.3.3 Secondary operators and functions

By definition, the *DeonticStructure* operator, the *PreferenceStructure* operator and the *ChoiceSet* function in a simple DALMAS are totally defined in terms of the primary operators and functions. These operators and functions may be viewed as *secondary* operators/functions which are explicitly defined within the theory. In other words, they are defined by a formula containing primary or other secondary terms. Generally, there should be no need to redefine the secondary predicates when specifying a simple deterministic DALMAS.

Since $a = BreakTie(ChoiceSet(\omega,s))$ is an element of A, it follows that $a(\omega,s) = [BreakTie(ChoiceSet(\omega,s))](\omega,s)$ is an element of the state space S. To determine the situation $<\omega^+,s^+>$ which follows directly after another situation $<\omega,s>$, we define a function *NextSituation*: $\Omega\times S \rightarrow \Omega\times S$ such that $<\omega^+,s^+> = NextSituation(\omega,s) = \langle NextToMove(\omega), [BreakTie(ChoiceSet(\omega,s))](\omega,s)\rangle$. In [14], this function is denoted by f.

By iteration of *NextSituation* we may define $NextSituation^n$ in the following way:

$$NextSituation^1(\omega,s) = NextSituation(\omega,s), \text{ and}$$
$$NextSituation^n(\omega,s) = NextSituation(NextSituation^{n-1}(\omega,s))$$

NextSituation and $NextSituation^n$ may also be viewed as secondary functions of a DALMAS. They in turn are used to define the *run* of a DALMAS: The *k*-event *run* of a deterministic DALMAS determined by the initial situation $<\omega_0,s_0>$ is the sequence $\langle<\omega_0,s_0>, NextSituation^1(\omega_0,s_0), \ldots, NextSituation^k(\omega_0,s_0)\rangle$.

5.3.4 A COLOUR & FORM DALMAS

For the simple COLOUR&FORM system introduced in section 2.1 we define a DALMAS $\boldsymbol{D}^F$ in the following way:

- ω_c denotes agent *chroma*;
- ω_f denotes agent *forma*;
- a_c denotes action *change colour*; and
- a_f denotes action *change form*.

Thus, the agent set $\Omega = \{\omega_c, \omega_f\}$ and the action set $A = \{a_c, a_f\}$. A state of the system will be represented by a 2-tuple $\langle S_c, S_f\rangle = \langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle$ where $c_x\in\{black, white\}$ is the colour of agent ω_x and $f_x\in\{circle, square\}$ is the form of agent ω_x. The state space $S = S_c\times S_f$ is a set of states containing 16 elements.

A situation of the system is characterized by an agent ω_m that is to move, and a state: $\langle\omega_m, \langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle\rangle$. The set of possible situations of the system contains 32 elements.

The primary operators and functions are defined in the following way: Let $flip(black) = white$, $flip(white) = black$, $flip(circle) = square$, and $flip(square) = circle$. Then the action functions may be defined as follows:

- $a_c(\langle\omega_c, \langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle\rangle) = \langle\langle\omega_c, flip(c_c), f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle$
- $a_c(\langle\omega_f, \langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle\rangle) = \langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, flip(c_f), f_f\rangle\rangle$

- $a_f(\langle\omega_c, \langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle\rangle) = \langle\langle\omega_c, c_c, flip(f_c)\rangle, \langle\omega_f, c_f, f_f\rangle\rangle$
- $a_f(\langle\omega_f, \langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle\rangle) = \langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, flip(f_f)\rangle\rangle$

In this simple system, all actions in the action set are always feasible: $Feasible_{\omega,s}(a)$ iff $a \in A$, where A is the action set of $\boldsymbol{D}^F$ in state s.

The utility function assigns high utility to act a_c and lower utility to act a_f for agent ω_c, and the opposite for ω_f:

- $U_\omega(a_c(\langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle)) = high$ if $\omega = \omega_c$
- $U_\omega(a_c(\langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle)) = low$ if $\omega = \omega_f$
- $U_\omega(a_f(\langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle)) = high$ if $\omega = \omega_f$
- $U_\omega(a_f(\langle\langle\omega_c, c_c, f_c\rangle, \langle\omega_f, c_f, f_f\rangle\rangle)) = low$ if $\omega = \omega_f$

The turn-operator is simple:

- $NextToMove(\omega_c) = \omega_f$, and
- $NextToMove(\omega_f) = \omega_c$.

The tie-breaker simply chooses the first element of the enumeration of the ordered choice-set: $BreakTie(\{a_1, \dots , a_n\}) = a_1$.

The secondary predicates are completely defined in terms of the primary predicates, and need not be changed.

5.3.5 A WASTE-COLLECTOR DALMAS

The specification of a DALMAS for the WASTE-COLLECTORS system (section 2.2) is done in a similar way.

The agent set $\Omega = \{\omega_1, \omega_2, \dots , \omega_n\}$ and the action set $A = \{go_{north}, go_{south}, go_{west}, go_{east}, grab, pass\}$. An action in the action set is feasible if it does not lead to a state where the moving agent is on a position outside the boundaries of the grid.

An element of the state space S is characterized by the three functions *Position*, *Waste* and *Collected*, such that

- *Position*: $\Omega \rightarrow \aleph^2$ assigns a position $(x_s(\omega),y_s(\omega))$ to each agent in the agent set;
- *Waste*: $\aleph^2 \rightarrow \Re$ assigns an amount of waste to each position; and
- *Collected*: $\Omega \rightarrow \Re$ assigns the amount of waste collected by each agent.

These functions are formally specified in [14, pp. 154-155], where they are denoted π, γ and σ. The action function go_{north} is defined as follows: $go_{north}(\langle\omega, s\rangle) = \langle Position^+, Waste, Collected\rangle$, where $Position^+ = (x_s(\omega), y_s(\omega)+1)$. The functions go_{south}, go_{west} and go_{east} are defined analogously.

For *grab*, we define $grab(\langle\omega, s\rangle) = \langle Position, Waste^+, Collected^+\rangle$ where $Waste^+(Position_s(\omega)) = 0$, and $Collected^+(\omega) = Collected_s(\omega) + Waste_s(Position_s(\omega))$. The *pass* function does not change the state at all: $pass(\langle\omega, s\rangle) = \langle Position_s, Waste_s, Collected_s\rangle$.

There are many ways to define the utility function U_ω. We will examine one such possibility in section 6.6. The turn-operator lets the agent take turns to move:

- $NextToMove(\omega_x) = \omega_{x+1}$ if $x<n$, where n is the number of agents in Ω; and
- $NextToMove(\omega_n) = \omega_1$.

The tie-breaker is defined in the same way as in the previous section. The secondary predicates are completely defined in terms of the primary predicates.

5.4 Normative Systems

We will now turn our attention to the other part of the definition of a norm-regulated DALMAS: the normative system.

A normative system $\boldsymbol{N}$ for a DALMAS $\boldsymbol{D}$ is a so-called *ground-consequence-system*, abbreviated *gc-system*[6]. More precisely, a normative system is a gc-system $\langle \boldsymbol{C}_M, \boldsymbol{D}_T, \boldsymbol{J}\rangle$ where $\boldsymbol{J}$ is a set of *norms*. $\boldsymbol{C}_M$ is a system of possible *grounds* and $\boldsymbol{D}_T$ is a system of possible *consequences* for these norms.

5.4.1 Overview of Norms

In predicate logic, a norm is often represented as a universal sentence which correlates two open sentences $p(x_1, \dots, x_v)$ and $q(x_1, \dots, x_v)$:

$$\forall x_1, \dots, x_v\colon p(x_1, \dots, x_v) \rightarrow q(x_1, \dots, x_v)$$

We may also view p and q as *conditions*, and a norm as a relational statement which correlates these conditions: $p\ R\ q$. The free variables in $p(x_1, \dots, x_v)$ must be the same, and in the same order, as the free variables in $q(x_1, \dots, x_v)$. However, it is not necessary that p and q have the same arity, as we will see in section 5.4.3. The statement $p\ R\ q$ is a relational statement equivalent to $\langle p, q\rangle \in R$. A norm can then be represented as the ordered pair $\langle p, q\rangle$, where p is called the *ground* (a descriptive condition) of the norm and q is called the *consequence* (a normative condition) of the norm. It is easy to form conjunctions, disjunctions and negations of such conditions, which makes it possible to construct Boolean algebras of conditions.

Using the move-operators and the type-operators (sections 5.2.1 - 5.2.3) we can express norms of the form

$$\forall x_1, \dots, x_v\colon Mc(x_1, \dots, x_v) \rightarrow T_i d(x_1, \dots, x_v)$$

where $Mc(x_1, \dots, x_v)$ is the ground of the norm and $T_i d(x_1, \dots, x_v)$ is its consequence. Such a norm may then be represented as the ordered pair $\langle M_n c, T_j d\rangle$.

The algebraic notation for norms was introduced by Odelstad and Lindahl in [9] to facilitate the formulation and study of normative systems. We will examine this

[6] We will study the theory of gc-systems in section 5.4.2. The formal definition of a ground-consequence-system is given in [14, p. 151].

notation in detail in section 5.4.3, but before we do this we will take a brief look at its theoretical foundation: *gc-systems*, *condition-implication structures* and *Boolean quasi-orderings*. The latter notions are discussed in detail in [14, pp. 156ff]. In subsection 5.4.4 and 5.4.5 the algebraic notation for norms will be used to describe normative systems for the COLOUR&FORM system and the WASTE-COLLECTORS system, respectively. In section 5.5 we shall see how *Prohibited* may be defined in terms of a normative system.

5.4.2 Theoretical foundation: gc-systems, Bqo and cis

A gc-system $\langle \boldsymbol{R_1}, \boldsymbol{R_2}, \boldsymbol{J} \rangle$ is a *Boolean joining-system*[7] if $\boldsymbol{R_1}$ and $\boldsymbol{R_2}$ are condition-implication structures, abbreviated *cis*. A *cis* is a special kind of a *Boolean quasi-ordering* (*Bqo*), a notion which was introduced in [8]. The theory of Boolean quasi-orderings was further developed by Odelstad and Lindahl in [9, 10]. A *Bqo* is a relational structure based on a Boolean algebra[8] and satisfying certain requirements:

Definition. The structure $\boldsymbol{B} = \langle B, \wedge, \neg, R \rangle$ is a *Boolean quasi-ordering* if $\langle B, \wedge, \neg \rangle$ is a *Boolean algebra* and R is a binary, reflexive and transitive relation on B such that R satisfies the following conditions for all a, b and c in B:

1. $a\ R\ b$ and $a\ R\ c$ implies $a\ R\ (b \wedge c)$
2. $a\ R\ b$ implies $\neg b\ R\ \neg a$
3. $(a \wedge b)\ R\ a$
4. not $\top\ R\ \bot$

The formal aspects of Boolean quasi-orderings are investigated in [10, pp. 67ff] and [14, pp. 149ff], together with an important class of models of the theory of Boolean quasi-orderings: condition-implication structures. The domains of such structures are sets of conditions.

Definition. A *condition-implication-structure* (*cis*) is a Boolean quasi-ordering $\boldsymbol{B} = \langle B, \wedge, \neg, R \rangle$ such that B is a domain of conditions, and R is a relation such that $a\ R\ b$ represents that *a implies b*.

If a and b are ν-ary conditions, then $a\ R\ b$ is the representation of

$$\forall x_1, \dots, x_\nu \colon a(x_1, \dots, x_\nu) \rightarrow b(x_1, \dots, x_\nu)$$

The extensions of conditions are relations, and relations may be regarded as sets of ordered n-tuples. The counterparts of negation, conjunction and disjunction for conditions are complement, intersection and union for relations. Note, however, that if R_1 and R_2 are relations of different arity, then their intersection $R_1 \cap R_2$ is empty and

[7] For the definition of a Boolean joining-system, see [14, p. 151].

[8] The postulates for a Boolean algebra $\langle B, \wedge, \neg \rangle$ with $\top$ as unit element and $\bot$ as zero element, where $\wedge$ represents conjunction and $\neg$ represents negation, are presented in [10]. Note that $\vee$ (disjunction) may be defined in terms of $\wedge$ and $\neg$, and that [10] uses ‘ for negation.

their union $R_1 \cup R_2$ is not a relation. For intersection and union of conditions, it is not required that the conditions have the same arity. This must be dealt with in some way, since we wish to express conditional norms involving conditions of different arity within the theory of Boolean quasi-orderings. This issue is addressed in [10, p. 72], and further discussed in the following subsection.

We now define two specific condition-implication structures for grounds that contain the move-operators M and M_n and consequences that contain the type-operators T_i [14, pp. 157f]:

Definition. An *m-cis* over a Boolean quasi-ordering $\boldsymbol{C} = \langle C, \wedge, \neg, R\rangle$ is a Boolean quasi-ordering $\boldsymbol{C_M} = \langle C_M, \wedge_M, \neg_M, R_M\rangle$ where

1. $C_M = \{Mc\colon c\in C\}$, where C is a domain of conditions;
2. $Mc \wedge_M Md = M(c \wedge d)$;
3. $\neg_M(Mc) = M(\neg c)$; and
4. $Mc\ R_M\ Md$ iff $c\ R\ d$.

Definition. A *normative-position-cis* (*np-cis*) over a Boolean quasi-ordering $\boldsymbol{D} = \langle D, \wedge, \neg, R\rangle$ is a Boolean quasi-ordering $\boldsymbol{D_T} = \langle D^*_T, \wedge_T, \neg_T, R_T\rangle$ with $\top_T$ as unit element and $\bot_T$ as zero element, where

1. D^*_T is defined as described below;
2. $(T_ic \wedge_T T_jd)(\omega_1, \ldots, \omega_v, \omega_{v+1}; \omega,s)$ iff
   $T_ic(\omega_1, \ldots, \omega_v, \omega_{v+1}; \omega,s) \wedge T_jd(\omega_1, \ldots, \omega_v, \omega_{v+1}; \omega,s)$;
3. $\neg_T(T_ic)(\omega_1, \ldots, \omega_v, \omega_{v+1}; \omega,s)$ iff $\neg T_ic(\omega_1, \ldots, \omega_v, \omega_{v+1}; \omega,s)$; and
4. R_T meets the requirements specified in [14, p. 157].

D is a domain of conditions, and $D_T = \{T_id\colon d\in D \text{ and } 1 \leq i \leq 7\}$ is the set of *normative positions* over D. We define D^*_T recursively as follows:

1. $D_T \subseteq D^*_T$
2. If $p\in D^*_T$ and $q\in D^*_T$, then $(p \wedge_T q)\in D^*_T$
3. If $p\in D^*_T$, then $\neg_T p\in D^*_T$
4. D^*_T contains no other members than those resulting from a finite number of applications of 1 and 2.

These specific condition-implication structures will be used in the definition of a normative system given in the following subsection.

5.4.3 Definition of a Normative System

As already mentioned, a normative system $\boldsymbol{N} = \langle \boldsymbol{C_M}, \boldsymbol{D_T}, \boldsymbol{J}\rangle$ for a DALMAS $\boldsymbol{D}$ is a gc-system such that $\boldsymbol{C_M}$ is the m-cis over the Boolean quasi-ordering $\langle C, \wedge, \neg, R_C\rangle$ and $\boldsymbol{D_T}$ is the np-cis over the Boolean quasi-ordering $\langle D, \wedge, \neg, R_D\rangle$, where C and D are sets of state-conditions for a DALMAS. In other words, $\boldsymbol{C_M} = \langle C_M, \wedge_M, \neg_M, R_M\rangle$ is a system of possible *grounds* and $\boldsymbol{D_T} = \langle D^*_T, \wedge_T, \neg_T, R_T\rangle$ is a system of possible *consequences*, and

N is a joining of a Bqo of grounds to a Bqo of consequences. For more details, see [14, p. 158]. From the definitions in the previous subsection it follows that an element in C_M is of the form *Mc* or M_nc and an element in D_T is (in simple cases) of the form T_jd. Note that *Mc* and T_jd are situation-conditions formed by applying the move- and type-operators to the state-conditions $c(\omega_1,\ldots, \omega_p; s)$ and $d(\omega_1,\ldots, \omega_q; s)$, as described in section 5.2.

In the following, we will use the term *norm-set* for ***J***. An element in a norm-set is called a *norm*, and is represented by a 2-tuple $\langle x,y\rangle$ where *x* is the *ground* of the norm and *y* is its *consequence*. An *elementary* norm is a norm $\langle x,y\rangle$ where the consequence *y* is an element of D_T. For example, the intended interpretation of the elementary norm $\langle M_nc, T_jd\rangle$ is a sentence of the following form:

$$\forall\omega_1,\ldots, \omega_v,\omega_{v+1}\in\Omega\text{: } \forall s\in S\text{: } M_nc(\omega_1,\ldots, \omega_p, \omega_{v+1}; \omega,s) \rightarrow T_jd(\omega_1,\ldots, \omega_q, \omega_{v+1}; \omega,s)$$

where $v = max(p,q)$.

A non-elementary norm is a norm $\langle x,y\rangle$ where the consequence *y* is an element of D^*_T, which means that it is a Boolean combination of elements of the form T_id. An example of such a norm is $\langle M_1c, T_2d_1\vee T_3d_2\rangle$, with the intended interpretation

$$\forall\omega_1,\ldots, \omega_v,\omega_{v+1}\in\Omega\text{: } \forall s\in S\text{: } M_1c(\omega_1,\ldots, \omega_p,\omega_{v+1}; \omega,s) \rightarrow$$
$$T_2d_1(\omega_1,\ldots, \omega_q,\omega_{v+1};\omega,s) \vee T_3d_2(\omega_1,\ldots, \omega_r,\omega_{v+1}; \omega,s)$$

where $v = max(p,q,r)$.

Note that the free variables in the sit-conditions of a particular norm must be the same (and in the same order) as the other conditions of the same norm, but it is not necessary that all sit-conditions are of the same arity. [14, p. 146] Suppose, for instance, that $p=2$, $q=2$ and $r=4$ in the previous example. Then the norm must have the following form:

$$\forall\omega_1,\omega_2,\omega_3,\omega_4,\omega_5\in\Omega\text{: } \forall s\in S\text{: } M_1c(\omega_1,\omega_2,\omega_5; \omega,s) \rightarrow$$
$$T_2d_1(\omega_1,\omega_2,\omega_5; \acute{}\omega,s) \vee T_3d_2(\omega_1,\omega_2,\omega_3,\omega_4,\omega_5; \omega,s)$$

Recall that the last two arguments, ω and *s*, denote a situation of the system. In this example we have $\omega_5 = \omega_1 =\omega$, due to the definition of the move-operator M_1.

5.4.4 Normative System for the COLOUR&FORM system

The COLOUR&FORM system is regulated by a very simple normative system: An agent may not act so that both agents end up with the same values on both attributes. In other words, if $\omega_1 \neq \omega_2$, then the moving agent must not act so that all of ω_1*'s* attributes are identical with all of ω_2*'s* attributes. This may easily be expressed in logical form (with the universal quantifier $\forall$ omitted) using the conditions *Diff* and *Eq* which were defined in section 5.1.1:

$$Diff(\omega_1, \omega_2) \rightarrow Shall\ Do(\omega_1, \neg Eq(\omega_1, \omega_2))$$

This norm may be expressed using the T_7 operator introduced in section 5.2.3:

$$M_1 Diff(\omega_1, \omega_2; s) \rightarrow T_7 Eq(\omega_1, \omega_2; \omega_m, s)$$

The algebraic form of this norm is $\langle M_1 Diff, T_7 Eq\rangle$. The ground-set and consequence-set of this norm-system each contain one condition: $C = \{Diff\}$ and $D = \{Eq\}$.

5.4.5 Normative System for the WASTE-COLLECTORS system

An example of a normative system for the WASTE-COLLECTORS system has been described in [14, p. 144]. This normative system contains eight rules of the same general form: if some condition on the position of ω_1 is fulfilled, then ω_1 may or may not see to it that some other condition will or will not be the case. Suppose that ω_1 is about to move. Then the following rules govern the behaviour of ω_1:

(n_1) If the protected spheres (see section 5.1.1) of ω_1 and ω_2 do not overlap then ω_1 may not move so that the protected spheres of ω_1 and ω_2 overlap with two or three squares.

(n_2) If the protected spheres of ω_1 and ω_2 do not overlap then ω_1 may move so that the protected spheres of ω_1 and ω_2 overlap with zero or one square.

(n_3) If the protected spheres of ω_1 and ω_2 overlap with one or two squares, then ω_1 may move so that the protected spheres of ω_1 and ω_2 overlap with any number of squares (even zero).

(n_4) ω_1 may move so that the protected spheres of ω_1 and ω_2 overlap with six squares only if the protected spheres of ω_1 and ω_2 overlap with at least six squares.

(n_5) If the protected spheres of ω_1 and ω_2 overlap with four squares, then ω_1 shall move so that the protected spheres of ω_1 and ω_2 do not overlap with three squares.

(n_6) If the protected spheres of ω_1 and ω_2 overlap with six squares, then ω_1 must move so that the protected spheres of ω_1 and ω_2 overlap with at least four squares.

(n_7) ω_1 may never move so that the protected spheres of ω_1 and ω_2 overlap with nine squares.

(n_8) ω_1 may always move so that the protected spheres of ω_1 and ω_2 overlap with zero squares.

Using the logical notation described in section 3 and the predicates introduced in section 5.1.1, the rules (n_1) through (n_8) can be expressed in the following way:

(n_1) $Lap_0(\omega_1, \omega_2) \rightarrow \neg May\ Do(\omega_1, Lap_2(\omega_1, \omega_2) \vee Lap_3(\omega_1, \omega_2))$

(n_2) $Lap_0(\omega_1, \omega_2) \rightarrow May\ Do(\omega_1, Lap_0(\omega_1, \omega_2)) \wedge May\ Do(\omega_1, Lap_1(\omega_1, \omega_2))$

(n_3) $Lap_1(\omega_1, \omega_2) \vee Lap_2(\omega_1, \omega_2) \rightarrow May\ Do(\omega_1, Lap_i(\omega_1, \omega_2))$ for all j, $0 \leq j \leq 9$

(n_4) $\neg Lap_4(\omega_1, \omega_2) \wedge \neg Lap_6(\omega_1, \omega_2) \wedge \neg Lap_9(\omega_1, \omega_2) \rightarrow \neg May\ Do(\omega_1, Lap_6(\omega_1, \omega_2))$

(n_5) $Lap_4(\omega_1, \omega_2) \rightarrow Shall\ Do(\omega_1, \neg Lap_3(\omega_1, \omega_2))$

(n_6) $Lap_6(\omega_1, \omega_2) \rightarrow Shall\ Do(\omega_1, Lap_4(\omega_1, \omega_2) \vee Lap_6(\omega_1, \omega_2) \vee Lap_9(\omega_1, \omega_2))$

(n_7) $\omega_1 \neq \omega_2 \rightarrow \neg May\ Do(\omega_1, Lap_9(\omega_1, \omega_2))$

(n_8) $\mathrm{T} \rightarrow May\ Do(\omega_1, Lap_0(\omega_1, \omega_2))$

In (n_8), T denotes the relation of logical truth. The basic idea of this normative system is to make sure that the agents do not move too close to each other. Table 2 shows which of the possible changes of condition (see the table in section 2.2) are prohibited by each norm:

Table 2. Prohibited changes of condition

Condition	**Prohibited conditions in the next state**	**Rule**
$Lap_0(\omega_m,\omega_i)$	$Lap_2(\omega_m,\omega_i)$ $Lap_3(\omega_m,\omega_i)$	(n_1) (n_1)
$Lap_1(\omega_m,\omega_i)$	-	-
$Lap_2(\omega_m,\omega_i)$	-	-
$Lap_3(\omega_m,\omega_i)$	$Lap_6(\omega_m,\omega_i)$	(n_4)
$Lap_4(\omega_m,\omega_i)$	-	-
$Lap_6(\omega_m,\omega_i)$	$Lap_3(\omega_m,\omega_i)$ $Lap_9(\omega_m,\omega_i)$	(n_6) (n_7)
$Lap_9(\omega_m,\omega_i)$	$Lap_9(\omega_m,\omega_i)$ if $\omega_m \neq \omega_i$	(n_7)

Some of the prohibitions in table 2 may seem a bit strange. For example, it is perhaps a bit odd that the moving agent may not change the condition from Lap_6 to Lap_3. However, it is important to stress that this normative system is not intended to be a particularly "good" one. As Odelstad and Boman point out, its main purpose is to illustrate the logical form of norms and their formal connection to other parts of the DALMAS architecture. [14, pp. 143ff]

Note that the rules (n_2), (n_3) and (n_8) do not impose any restrictions on the acts of the moving agent.[9] Instead, they explicitly state that some changes of condition are permissible. For a simple norm-regulated DALMAS, where an act is permissible if and only if it is not prohibited, this is not necessary. In such a system, an act is (implicitly) permissible if it is not explicitly prohibited by the normative system. Therefore, (n_2), (n_3) and (n_8) are in a sense meaningless in the normative system for a simple norm-regulated DALMAS. However, in a non-simple DALMAS it would be

[9] Rule (n_5) prohibits the moving agent to change the condition from Lap_4 to Lap_3. Since this change is not possible anyway, rule (n_5) has no effect.

possible to use other interpretations of *Permissible*. For example, an act could be considered as prohibited if it is not explicitly permissible according to the normative system. In such a system, (n_2), (n_3) and (n_8) would be meaningful.

Table 3 shows which changes of condition are implicitly permissible (that is, not prohibited) by the normative system:

Table 3. Possible changes of condition that are implicitly permissible

Condition	**Implicitly permissible conditions in the next state**
$Lap_0(\omega_m,\omega_i)$	$Lap_0(\omega_m,\omega_i)$, $Lap_1(\omega_m,\omega_i)$
$Lap_1(\omega_m,\omega_i)$	$Lap_0(\omega_m,\omega_i)$, $Lap_1(\omega_m,\omega_i)$, $Lap_2(\omega_m,\omega_i)$
$Lap_2(\omega_m,\omega_i)$	$Lap_0(\omega_m,\omega_i)$, $Lap_1(\omega_m,\omega_i)$, $Lap_2(\omega_m,\omega_i)$, $Lap_3(\omega_m,\omega_i)$, $Lap_4(\omega_m,\omega_i)$
$Lap_3(\omega_m,\omega_i)$	$Lap_0(\omega_m,\omega_i)$, $Lap_2(\omega_m,\omega_i)$, $Lap_3(\omega_m,\omega_i)$
$Lap_4(\omega_m,\omega_i)$	$Lap_2(\omega_m,\omega_i)$, $Lap_4(\omega_m,\omega_i)$, $Lap_6(\omega_m,\omega_i)$
$Lap_6(\omega_m,\omega_i)$	$Lap_4(\omega_m,\omega_i)$, $Lap_6(\omega_m,\omega_i)$
$Lap_9(\omega_m,\omega_i)$	*None* if $\omega_m \neq \omega_i$, otherwise $Lap_9(\omega_m,\omega_m)$

The algebraic forms of (n_1) - (n_8) are shown here:

1. $\langle M_1Lap_0, T_4Lap_2 \vee T_6Lap_2 \vee T_7Lap_2\rangle$
2. $\langle M_1Lap_0, T_4Lap_3 \vee T_6Lap_3 \vee T_7Lap_3\rangle$
3. $\langle M_1Lap_0, T_1Lap_0 \vee T_2Lap_0 \vee T_3Lap_0 \vee T_5Lap_0\rangle$
4. $\langle M_1Lap_0, T_1Lap_1 \vee T_2Lap_1 \vee T_3Lap_1 \vee T_5Lap_1\rangle$
5. $\langle M_1Lap_1, T_1Lap_j \vee T_2Lap_j \vee T_3Lap_j \vee T_5Lap_j\rangle$ for all j, $0 \leq j \leq 9$
6. $\langle M_1Lap_2, T_1Lap_j \vee T_2Lap_j \vee T_3Lap_j \vee T_5Lap_j\rangle$ for all j, $0 \leq j \leq 9$
7. $\langle M_1(\neg Lap_4 \wedge \neg Lap_6 \wedge \neg Lap_9), T_7Lap_6\rangle$
8. $\langle M_1Lap_4, T_7Lap_3\rangle$
9. $\langle M_1Lap_6, T_5(Lap_4 \vee Lap_6 \vee Lap_9)\rangle$
10. $\langle M_1Diff, T_7Lap_9\rangle$
11. $\langle M\mathrm{T}, T_1Lap_0 \vee T_2Lap_0 \vee T_3Lap_0 \vee T_5Lap_0\rangle$

This formulation follows the example in [14, p. 159], but uses the new M_n-operator which was introduced in section 5.2.2. Together these norms express the eight rules in section 5.4.5. The ground-set C of this norm-system contains eight conditions, and the consequence-set D contains seven conditions: $C = \{Diff, Lap_j\}$ and $D = \{Lap_j\}$, where $j \in \{0,1,2,3,4,6,9\}$.

5.5 Definition of Prohibited

The idea behind the abstract architecture for a norm-regulated DALMAS is that some acts are prohibited for an agent, if they lead to states that are forbidden according to the normative system. Let $Prohibited_{\omega,s}$ denote the set of prohibited actions for agent ω in state s. The definition is straightforward:

$$Prohibited_{\omega,s} = \{a \in FeasibleActs(\omega,s): Prohibited_{\omega,s}(a)\}$$

We will now examine how $Prohibited_{\omega,s}(a)$ may be defined in terms of the normative system. It follows from the definition of the one-agent type of normative position $\mathbf{T}_i$ (section 4) and the definition of the corresponding operators T_i and E_i^a (see section 5.2.3) that if $(T_i d \wedge E_i^a d)(\omega_1,\ldots,\omega_v,\omega_{v+1};\omega,s)$ holds, then act a is prohibited for agent ω_{v+1}.

Note that $T_i d$ may be the consequence of an elementary norm. This means that act a is prohibited for agent ω_{v+1} in situation $<\omega,s>$ if the following holds:

1. There are conditions c and d such that $\langle M_n c, T_i d\rangle$ is an elementary norm for some i, $2 \le i \le 7$; and
2. there are agents $\omega_1,\ldots,\omega_v$ in the agent set Ω such that $M_n c(\omega_1,\ldots,\omega_x,\omega_{v+1};\ \omega,s)$ and $E_i^a d(\omega_1,\ldots,\omega_v,\omega_{v+1};\ \omega,s)$.

Also note that from the definition of M_n it follows that $\omega_{v+1} = \omega$ and $\omega_{v+1} = \omega_n$, where ω_n is the n:th agent argument of condition c. To summarize,

$$Prohibited_{\omega,s}(a) \leftarrow M_n c(\omega_1,\ldots,\omega_x,\omega;\ \omega,s) \wedge \langle M_n c, T_i d\rangle \wedge E_i^a d(\omega_1,\ldots,\omega_v,\omega;\ \omega,s).$$

Let us illustrate with a simple example: Suppose that the normative system contains the norm $\langle M_1 Diff, T_7 Lap_9\rangle$, where *Diff* and *Lap* are defined as in section 5.1.1. Let us further assume that $M_1 Diff(\omega_1,\ \omega_2,\ \omega_3;\ \omega,s)$ holds in situation $\langle\omega,s\rangle$. According to the definitions of *Diff* and the M_1 operator, this would mean that $\omega_1 \neq \omega_2$, and $\omega_1 = \omega_3 = \omega$. Now, any act a for which $E_7^a Lap_9(\omega,\ \omega_2,\ \omega;\ \omega,s)$ holds will be prohibited for ω. That is, an act a which leads to $Lap_9(\omega_1,\ \omega_2;\ s^+)$ in the following state s^+ is prohibited, no matter if $Lap_9(\omega_1,\ \omega_2;\ s)$ holds in state s or not. Let us as an example assume that ω_1 is at position (1,1), and ω_2 is at position (2,1), and ω_1 is the moving agent. Then act $a_e =$ *go(east)* would be prohibited for ω_1, since $\omega_1 \neq \omega_2$ and $Lap_9(\omega_1,\ \omega_2;\ s^+)$ for $s^+ = a_e(\omega_1,\ s)$.

For non-elementary norms the situation is a little more complicated. For consequences consisting of disjunctions of $T_i d$ we have [14, p. 162]

$$(T_i d_1 \vee T_j d_2)(\omega_1,\ldots,\omega_v,\omega;\ \omega,s) \rightarrow$$
$$[(E_i^a d_1 \wedge E_j^a d_2)(\omega_1,\ldots,\omega_v,\omega;\ \omega,s) \rightarrow Prohibited_{\omega,s}(a)]$$

which leads us to conclude that

$$Prohibited_{\omega,s}(a) \leftarrow [M_n c(\omega_1,\ldots,\omega_x,\omega;\ \omega,s) \wedge \langle M_n c,\ T_i d_1 \vee T_j d_2\rangle \wedge$$
$$(E_i^a d_1 \wedge E_j^a d_2)(\omega_1,\ldots,\omega_v,\omega;\ \omega,s)].$$

This may be generalised: A consequence consisting of any number of *disjunctions* of $T_i d_x$ leads to prohibition of act a if the *conjunction* of all the corresponding $E_i^a d_x$ holds. Similarly, for consequences consisting of conjunctions of $T_i d$ we have

$$(T_i d_1 \wedge T_j d_2)(\omega_1,\ldots,\omega_v,\omega;\ \omega,s) \rightarrow$$
$$[(E_i^a d_1 \vee E_j^a d_2)(\omega_1,\ldots,\omega_v,\omega;\ \omega,s) \rightarrow Prohibited_{\omega,s}(a)].$$

It is, however, not a recommended style to express norms with consequences consisting of conjunctions of $T_i d$. Instead, such norms may be expressed with two separate elementary norms with the same ground.

6 From Set-Theory to Prolog

If we specify an implementation of a specific DALMAS by defining extensions of the right type for the primary predicates, then we should get a model for the theory of DALMASs. This is a reasonable criterion for the correctness of the Prolog framework: if correct extensions of the primary predicates are provided, then the framework gives the extensions of the secondary predicates that follow from the theory.

When going from a set-theoretical description of the theory of DALMASs to a Prolog implementation, there are some issues that need to be addressed. One key issue is the representation of *sets*. A set may be defined by *facts* (that is, by an enumeration of the elements in the set) or by *rule*. Finite sets may easily be enumerated in Prolog by the use of lists or other data structures. Definition by rule is accomplished by creating Prolog predicates that generate an enumeration of the elements in the set. Typically, such enumerations are finite, but some Prolog implementations can at least to some extent handle infinite recursively defined sets.

In a set-theoretical definition, sets are rather abstract entities. Issues regarding computability, efficiency and in which order computations are carried out are of no interest. These issues may not be totally disregarded in Prolog. There are many situations where a choice can be made between a formulation of a predicate that leads to efficient code and a formulation that is closely modeled after the description in the abstract architecture. The latter choice has generally been made in this work.

Furthermore, the notion of *functions* does not exist in Prolog, only predicates. Therefore, a natural first step is to express the operators and functions introduced in section 5.3 as predicates. The aim of this process is to define a set of predicates which may be used as a kind of pseudo code to be easily translated to corresponding Prolog predicates. In the following, if g is a function or operator in the abstract architecture, then $g°$ will denote the corresponding predicate.

6.1 Predicate Formulation of state-conditions and sit-conditions

A state-condition $c_p(\omega_1,\ldots,\omega_x; s)$ is represented by $c°(p, \omega_1,\ldots,\omega_x; s)$, where p stands for zero or more parameters to the condition c. Some examples from the COLOUR&FORM system and the WASTE-COLLECTORS system will be given in sections 6.5 and 6.6.

Sit-conditions are formed by applying operators to state-conditions. For the move-operator M in section 5.2.1, we define a predicate $M°$ such that

$$M°(c°, p, \langle\omega_1, \ldots, \omega_x\rangle, \langle\omega, \ldots\rangle, <\omega_m, s>) \text{ iff } \omega = \omega_m \wedge c°(p, \omega_1,\ldots,\omega_x; s)$$

where p represents zero or more parameters to the state-condition c_p. Similarly, for the move-operator M_n in section 5.2.2, we define $M°$ such that

$$M°(n, c°, p, \langle \omega_1, \ldots, \omega_n, \ldots, \omega_x \rangle, \langle \omega, \ldots \rangle, <\omega_m, s>) \text{ iff}$$
$$\omega = \omega_m \wedge \omega = \omega_n \wedge c°(p, \omega_1, \ldots, \omega_x; s)$$

where p represents zero or more parameters to the state-conditon c_p, and ω_n is the n:th element in $\langle \omega_1, \ldots, \omega_n, \ldots, \omega_x \rangle$. Note that $M°$ is a second-order predicate, since it operates on other predicates.

For the E_i^a-operators ($2 \leq i \leq 7$) in section 5.5 we define a second-order predicate $E°$. For instance, $E_2{}^a$ is represented with

$$E°(2, d°, p, \langle \omega_1, \ldots, \omega_x \rangle, \langle \omega, a, \ldots \rangle, <\omega_m, s>) \text{ iff}$$
$$d°(p, \omega_1, \ldots, \omega_x\, s) \wedge \mathbf{A}°(a, \omega, s, s^+) \wedge \neg d°(r, \omega_1, \ldots, \omega_x; s^+)$$

where p may represent zero or more parameters to the state-condition d_p. The other clauses of $E°$ are defined analogously. The definition of $\mathbf{A}°$ is shown in section 6.2.1.

6.2 Predicate Formulation of DALMAS Functions and Operators

For a DALMAS $\boldsymbol{D}$, the *agent set* Ω may be represented as a finite enumeration of agents. It is useful to define a binary predicate $\Omega°$ that is true if and only if an agent ω is a member of the agent set: $\Omega°(\omega, \Omega)$ iff $\omega \in \Omega$.

In the abstract architecture, the elements of the *action set A* are functions, where each function defines a transformation rule from one state in the state space to another. In Prolog, it would be possible to represent these functions as predicates, and let the action set consist of an enumeration of the functors and arity of these predicates. However, this could cause some tricky implementation issues. Instead, a choice has been made to represent each action with a unique identifier. The transformation rules corresponding to each action will be represented by different clauses of the “action result predicate”; see below.

The action set may be represented in the same way as the agent set. We define a binary predicate $A°$ that is true if and only if an action a is a member of the action set A: $A°(a, A)$ iff $a \in A$.

6.2.1 Primary Predicates

The *state space S* of $\boldsymbol{D}$ is implicitly defined by the transformation rules that lead from one state to another. We define the predicate $\mathbf{A}°(a, \omega, s, s^+)$ such that s^+ is the resulting state of $\boldsymbol{D}$ when act a is performed by agent ω in state s. This predicate defines transformation rules from one state to another, so it must be defined for each feasible combination of action, agent and state.

In analogy with the definitions of $\Omega°$ and $A°$, it may be tempting to define a predicate $S°(s, S)$ which is true if and only if s is a member of the state space S. But since the state space may be infinite, this predicate would not be very useful in an implementation. Here we must assume that **A°** is defined in such a way that all possible valid states (and nothing but valid states) in the state space may be reached by application of the transformation rules defined by **A°**. This would mean that to decide if a state is a member of the state space S, we must apply all feasible actions to all possible initial states, and iterate the applications of the rules to see if the state is reached. Alternatively, it may be possible to prove that s may be reached. But for the sake of implementation we adopt a more “practical” view: If a state is reached by application of the transformation rules to valid states, then it is a member of the state space.

For the *feasible action predicate* we define a primary predicate $Feasible°(<\omega_m,s>, \omega, a)$ such that a is a feasible action for ω in situation $<\omega_m,s>$.

The *utility function* is represented by a utility predicate $U°(a, <\omega_m,s>, \omega, U_a)$ where U_a is a value describing the utility for ω to perform act $a \in \mathbf{a}$ in situation $<\omega_m,s>$.

For the *turn-operator* we define $NextToMove°(<\omega_m,s>, \omega^+)$ such that it is true if ω^+ is the agent to move after ω_m in situation $<\omega_m,s>$.

The *tie-breaker* may be represented by the predicate $BreakTie°(\Gamma, <\omega_m,s>, a)$ where a is the action chosen among the equally preferred actions in the choice-set Γ in situation $<\omega_m,s>$. Note that this formulation of the tie-breaking predicate is more complex than the corresponding function described in section 5.3. This reformulation has been made to give the possibility to make more informed decisions of which act to select in a specific situation. However, in many cases it is sufficient to simply let the tie-breaker choose the first element in the choice-set.

6.2.2 Secondary Predicates

For the *feasible actions function* (see section 5.3) we define a secondary predicate $FeasibleActions°(<\omega_m,s>, \omega, A^F)$ such that A^F is an enumeration of all actions a in the action set A which are feasible (accessible) for ω in the situation $<\omega_m,s>$:

$$FeasibleActions°(<\omega_m,s>, \omega, A^F) \text{ iff } A^F = \{\forall a \in A\text{: } Feasible°(<\omega_m,s>, \omega, a)\}$$

The representation of the *deontic structure operator* is defined as the predicate $DeonticStructure°(\boldsymbol{\alpha}, <\omega_m,s>, \omega, \Delta)$, where $\boldsymbol{\alpha}$ is a set of acts and Δ is a subset of $\boldsymbol{\alpha}$ consisting of the acts which are permissible for ω in situation $<\omega_m,s>$. It is in turn defined by the predicate $Permissible°(\boldsymbol{\alpha}, <\omega_m,s>, \omega, a)$, which is true if and only if $a \in \boldsymbol{\alpha}$ is a permissible action for ω in situation $<\omega_m,s>$:

$$DeonticStructure°(\boldsymbol{\alpha}, <\omega_m,s>, \omega, \Delta) \text{ iff } \Delta = \{\forall a \in \boldsymbol{\alpha}\text{: } Permissible°(\boldsymbol{\alpha}, <\omega_m,s>, \omega, a)\}$$

Recall that in a norm-regulated DALMAS it is the normative system that determines which actions are permissible in a given situation.

The *preference structure operator* is represented by the predicate $PreferenceStructure°(\boldsymbol{\alpha}, <\omega_m,s>, \omega, \Pi)$ which is defined to be true if Π is a subset of the set of actions $\boldsymbol{\alpha}$ ordered by the utility of the actions in $\boldsymbol{\alpha}$ for agent ω in situation $<\omega_m,s>$. More formally,

$$PreferenceStructure°(\boldsymbol{\alpha}, <\omega_m,s>, \omega, \Pi) \text{ iff } \Pi = \langle \boldsymbol{\alpha}, W\rangle$$

where W is defined in the following way: $a\ W\ b$ iff $U_a \geq U_b$.

With the implementation in mind, we may note that this definition relies on constructing a set that is ordered according to the weak ordering defined by $U_a \geq U_b$. As we shall see in section 7, it is possible to use the built-in Prolog predicate `setof/3` in a very elegant way, provided that the ordering is reversed. This means that we will assume that a *lower value* means *higher utility*, that is $a\ W\ b$ iff $U_a \leq U_b$.

For the *choice-set function* the corresponding predicate $ChoiceSet°(\boldsymbol{\alpha}, <\omega_m,s>, \omega, \Gamma)$ is defined to be true if Γ is a weak ordering of the deontic structure Δ for agent ω in situation $<\omega_m,s>$, according to the utility predicate $U°$:

$$ChoiceSet°(\boldsymbol{\alpha}, <\omega_m,s>, \omega, \Gamma) \text{ iff}$$
$$DeonticStructure°(\boldsymbol{\alpha}, <\omega_m,s>, \omega, \Delta) \wedge PreferenceStructure°(\Delta, <\omega_m,s>, \omega, \Pi) \wedge \Gamma = \Pi$$

We could also regard the choice-set as the (still ordered) intersection between the deontic structure and the preference structure, which is formed by application of the preference structure operator to the set of feasible actions. This view of the choice-set could in some cases lead to better computational efficiency in the corresponding Prolog implementation.

The "next situation function" *NextSituation* is represented with a predicate *NextSituation*° such that

$$NextSituation°(<\omega_m,s>, <\omega^+,s^+>, \omega, a) \text{ iff}$$
$$FeasibleActions°(<\omega_m,s>, \omega, A^F) \wedge DeonticStructure°(A^F, <\omega_m,s>, \omega, \Delta) \wedge$$
$$ChoiceSet°(\Delta, <\omega_m,s>, \omega, \Gamma) \wedge BreakTie°(\Gamma, <\omega_m,s>, a) \wedge \mathbf{A}°(a, \omega, s, s^+) \wedge$$
$$NextToMove°(<\omega_m,s>, \omega^+)$$

The k-event *run* of a DALMAS may be expressed as partial run between step 0 and step k:

$$Run°(<\omega_0,s_0>, k, Q) \text{ iff } PRun°(<\omega_0,s_0>, 0, k, \langle<\omega_0,s_0>\rangle, Q)$$

where Q is a sequence of k situations starting with the initial situation $<\omega_0,s_0>$. A *partial run* from step n to step k is defined recursively as the difference between the sequences Q_{n-1} and Q_k, where $Q_0 = \langle<\omega_0,s_0>\rangle$:

$PRun°(<\omega_n,s_n>, n, k, Q_{k-1}, Q_k)$ if
$n = k \wedge Q_{k-1} = \langle<\omega_0,s_0>, \dots , <\omega_{k-1},s_{k-1}>\rangle \wedge Q_k = \langle<\omega_0,s_0>, \dots , <\omega_{k-1},s_{k-1}>, <\omega_n,s_n>\rangle$

$PRun°(<\omega_n,s_n>, n, k, Q_{n-1}, Q)$ if
$n < k \wedge Q_{n-1} = \langle<\omega_0,s_0>, \dots , <\omega_{n-1},s_{n-1}>\rangle \wedge Q_n = \langle<\omega_0,s_0>, \dots , <\omega_{n-1},s_{n-1}>, <\omega_n,s_n>\rangle \wedge$
$NextSituation°(<\omega_n,s_n>, <\omega_{n+1},s_{n+1}>, \omega, a) \wedge PRun°(<\omega_{n+1},s_{n+1}>, n+1, k, Q_n, Q)$

6.3 Representation of Normative Systems

As already mentioned, the deontic structure for a simple DALMAS consists of the set of permissible actions for an agent in a given state. The idea behind a norm-regulated DALMAS is to define permissible in terms of what is not prohibited according to a normative system. As described in section 5.4.3, a normative system $\boldsymbol{N}$ for a DALMAS $\boldsymbol{D}$ is an ordered triple $\boldsymbol{N} = \langle \boldsymbol{C}_M, \boldsymbol{D}_T, \boldsymbol{J} \rangle$ where $\boldsymbol{C}_M$ is an *m-cis* of grounds and $\boldsymbol{D}_T$ is an *np-cis* of consequences. The representation and implementation of condition-implication structures and normative systems has been a central issue, which will be addressed in the following subsections.

6.3.1 Representation of Condition-Implication Structures

According to the definition in section 5.4.2, an m-cis $\boldsymbol{C}_M = \langle C_M, \wedge_M, \neg_M, R_M \rangle$ for a DALMAS is defined over a Boolean quasi-ordering $\boldsymbol{C} = \langle C, \wedge, \neg, R \rangle$ of grounds. Therefore, an m-cis may be represented by its corresponding set of conditions C together with the definition of the move-operators and the Boolean operations $\wedge_M$, $\neg_M$ and R_M. In the following, C will be called the *ground-set* of a DALMAS. The ground-set is represented as an enumeration of state-conditions of the form $c = f_c(p, \omega_1, \ldots, \omega_a; s)$, where $a \geq 0$ is the number of agent arguments for the state-condition c. We define the predicate *GroundSet*° such that

$$\textit{GroundSet}°(s_{\boldsymbol{D}}, C) \text{ iff } C \text{ is the ground-set for DALMAS } \boldsymbol{D} \text{ in state } s.$$

We also define a second-order predicate *Ground*° which is true if the predicate representation of a condition c_p is a member of a ground-set:

$$\textit{Ground}°(c_p, C, c°, p, a) \text{ iff } c_p \in C \text{ and } c_p \text{ is represented by } c°(p, \omega_1, \ldots, \omega_a; s)$$

where $p = \langle p_1, p_2, \ldots, p_n \rangle$ represents zero or more non-agent parameters to the condition c, and C is an enumeration of the ground-set for a DALMAS $\boldsymbol{D}$.

An np-cis $\boldsymbol{D}_T = \langle D^*_T, \wedge_T, \neg_T, R_T \rangle$ is defined over a Boolean quasi-ordering $\boldsymbol{D} = \langle D, \wedge, \neg, R \rangle$ of consequences. An np-cis may then be represented by its corresponding set of conditions D together with the definition of the type-operators and the Boolean operations $\wedge_T$, $\neg_T$ and R_T. We will use the term *consequence-set* for D. This set is represented in the same way as the ground-set. We define *ConsequenceSet*° and *Consequence*° in the following way:

$$\textit{ConsequenceSet}°(s_{\boldsymbol{D}}, D) \text{ iff } D \text{ is the consequence-set for DALMAS } \boldsymbol{D} \text{ in state } s\text{; and}$$

$$\textit{Consequence}°(d_p, D, d°, p, a) \text{ iff } d_p \in D \text{ and } d_p \text{ is represented by } d°(p, \omega_1, \ldots, \omega_a; s)$$

where $p = \langle p_1, p_2, \ldots, p_n \rangle$ represents zero or more non-agent parameters to d, and D is an enumeration of the consequence-set for a DALMAS $\boldsymbol{D}$.

We will return to the implementation of the move- and type-operators and the corresponding Boolean operations (including the operations $\wedge$, $\neg$ and R) in section

7.4.1. The implementation must guarantee that if the norms in the norm-system $\boldsymbol{N}$ are specified in accordance with the theory of normative systems, then the deontic structure that follows from $\boldsymbol{N}$ is correct.

6.3.2 Norm-systems, Norm-sets and Norms

The *dnrDALMAS* representation of a normative system will be called a *norm-system.* A norm-system is a 3-tuple $\langle C, D, \boldsymbol{J}\rangle$ where C is a ground-set, D is a consequence-set and $\boldsymbol{J}$ is a *norm-set.* An element in a norm-set $\boldsymbol{J}$ is a norm $\langle x,y\rangle$ where x is of the form Mc or M_nc and y is a Boolean combination[10] of T_id. The representation of $\boldsymbol{J}$ is straightforward: we let $\boldsymbol{J}$ be an enumeration of individual norms.

Let us define a predicate *Norm°* which is true if $\langle x,y\rangle_n$ is a norm (with identifier n) in the norm-system $\boldsymbol{N}$:

$$Norm°(\boldsymbol{N}, n, x, y) \text{ iff } \boldsymbol{N} = \langle \boldsymbol{C}_M, \boldsymbol{D}_T, \boldsymbol{J}\rangle \wedge \langle x,y\rangle_n \in \boldsymbol{J}$$

Also, *NormSystem°* is defined such that $NormSystem°(s_{\boldsymbol{D}}, \boldsymbol{N})$ iff $\boldsymbol{N}$ is the norm-system for the DALMAS $\boldsymbol{D}$ in state s.

In section 7.4.1, we will develop a kind of syntax for grounds and consequences, so that norms may be represented in a suitable way in Prolog. Their semantics is connected to how $Prohibited_{\omega,s}(a)$ is defined in terms of the norm-system. This will be the topic of the next subsection.

6.4 Predicate Formulation of Prohibited

Let $Prohibited°(a, \omega, <\omega_m,s>)$ be true if act a is prohibited for agent ω in situation $<\omega_m,s>$. The definition follows directly from the definition of $Prohibited_{\omega,s}(a)$ in section 5.5 and the definitions of *NormSystem°*, *Norm°*, *M°* and *E°*:

$$\begin{gathered} Prohibited°(a, \omega, <\omega_m,s>) \leftarrow \\ NormSystem°(s, \boldsymbol{J}) \wedge Norm°(\boldsymbol{J}, k, M_nc_p, T_id_q \vee \dots \vee T_jg_r) \wedge \\ M°(n, c, p, \langle\omega_1, \dots, \omega_n, \dots, \omega_x\rangle, \langle\omega\rangle, <\omega_m,s>) \wedge \\ E°(i, d, q, \langle\omega_1, \dots, \omega_y\rangle, \langle\omega, a\rangle, <\omega_m,s>) \wedge \dots \wedge E°(j, g, r, \langle\omega_1, \dots, \omega_z\rangle, \langle\omega, a\rangle, <\omega_m,s>) \end{gathered}$$

As we shall see in section 7.5, the Prolog implementation is not as straightforward as could be expected.

6.5 Predicates for the COLOUR&FORM system

The state-conditions of the COLOUR & FORM system (see section 5.1.1) are easily represented as predicates:

[10] In the following, the set of all possible Boolean combinations of elements of the form T_id is denoted $\boldsymbol{D}^*_T$.

- $\mathit{Diff}°(\omega_1, \omega_2; s)$ iff $\omega_1 \neq \omega_2$
- $\mathit{Eq}°(\omega_1, \omega_2; s)$ iff $s = \langle\langle \omega_c, c_c, f_c\rangle, \langle \omega_f, c_f, f_f\rangle\rangle \wedge c_c = c_f \wedge f_c = f_f$.

The primary predicates of the system are defined in the following way: Let *flip*(*black*) = *white*, *flip*(*white*) = *black*, *flip*(*circle*) = *square*, and *flip*(*square*) = *circle*. Then the action result predicate may be defined as follows:

- $\mathbf{A}°(a_c, \omega_c, s, s^+)$ iff $s = \langle\langle \omega_c, c_c, f_c\rangle, \langle \omega_f, c_f, f_f\rangle\rangle \wedge s^+ = \langle\langle \omega_c, \mathit{flip}(c_c), f_c\rangle, \langle \omega_f, c_f, f_f\rangle\rangle$
- $\mathbf{A}°(a_f, \omega_c, s, s^+)$ iff $s = \langle\langle \omega_c, c_c, f_c\rangle, \langle \omega_f, c_f, f_f\rangle\rangle \wedge s^+ = \langle\langle \omega_c, c_c, \mathit{flip}(f_c)\rangle, \langle \omega_f, c_f, f_f\rangle\rangle$
- $\mathbf{A}°(a_c, \omega_f, s, s^+)$ iff $s = \langle\langle \omega_c, c_c, f_c\rangle, \langle \omega_f, c_f, f_f\rangle\rangle \wedge s^+ = \langle\langle \omega_c, c_c, f_c\rangle, \langle \omega_f, \mathit{flip}(c_f), f_f\rangle\rangle$
- $\mathbf{A}°(a_f, \omega_f, s, s^+)$ iff $s = \langle\langle \omega_c, c_c, f_c\rangle, \langle \omega_f, c_f, f_f\rangle\rangle \wedge s^+ = \langle\langle \omega_c, c_c, f_c\rangle, \langle \omega_f, c_f, \mathit{flip}(f_f)\rangle\rangle$

The formulation of *flip* as a predicate is straightforward, but not shown here.

For feasible actions we simply let all actions in the action set be always feasible: $\mathit{Feasible}°(<\omega_m, s_{\mathbf{D}}>, \omega, a)$ iff $A°(a, A)$, where A is the action set of $\boldsymbol{D}$ in state s.

The utility predicate assigns for ω_c a lower value (meaning higher utility) to action a_c, and a higher value (meaning lower utility). The opposite holds for ω_f:

- $U°(a_c, <\omega_c,s>, \omega_c, 0)$;
- $U°(a_f, <\omega_c,s>, \omega_c, 1)$;
- $U°(a_c, <\omega_f,s>, \omega_f, 1)$; and
- $U°(a_f, <\omega_f,s>, \omega_f, 0)$.

The turn-operator is simple:

- $\mathit{NextToMove}°(<\omega_c,s>, \omega_f)$; and
- $\mathit{NextToMove}°(<\omega_f,s>, \omega_c)$.

The tie-breaker simply chooses the first element of the enumeration of the ordered choice-set:

$$\mathit{BreakTie}°(\Gamma, <\omega_m,s>, a) \text{ iff } \Gamma = \{a, \ldots\}$$

The secondary predicates are completely defined in terms of the primary predicates, and need not be changed.

6.6 Predicates for the WASTE-COLLECTORS system

The basic state-conditions of the WASTE-COLLECTORS system are defined in terms of the “agent position function” *Position*, the “amount of waste function” *Waste*, and the “collected waste function” *Collected* (section 5.3.5). These functions will be represented by ground facts in a global knowledge base of the system. In this sense, a state of the system corresponds to a state of the system’s knowledge base; see section 7.1 for a deeper discussion of this. The “agent position function” *Position* is represented by a fact $\mathit{Pos}°(\omega, \langle x,y\rangle)$, one for each agent in the agent set. Similarly, *Waste* is represented by a fact $Waste°(\langle x,y\rangle, n)$, $n \leq 0$, one for each position within the boundaries of the grid, and *Collected* is represented by a fact $Collected°(\omega, m)$, $m \geq 0$, one for each agent. We will use the notation $Pos°(\omega, \langle x,y\rangle) \in s$ to indicate that the position of ω is $\langle x,y\rangle$, according to a fact in the knowledge base of state s. Similarly, $Waste°(\langle x,y\rangle, n) \in s$ means that n is the amount of waste at position $\langle x,y\rangle$ in state s, and $Collected°(\omega, m) \in s$ means that m is the amount of waste carried by agent ω in state s.

For the protected sphere of an agent we define $Protected°(\omega, P; s)$, meaning that P is the protected sphere of ω in state s, but the exact definition of this predicate is left out here. For state-condition Lap_i (section 5.1.1) the overlap is defined as the number of elements (squares) in the intersection of protected spheres P_1 and P_2:

- $Lap°(i, \omega_1, \omega_2; s)$ iff $Protected°(\omega_1, P_1; s) \wedge Protected°(\omega_2, P_2; s) \wedge i$ is the number of elements in $P_1 \cap P_2$.

The action result predicate may be defined in the following way:

- $\mathbf{A}°(go_{north}, \omega, s, s^+)$ if
  $Pos°(\omega, \langle x,y\rangle) \in s \wedge s^+ = [s \setminus Pos°(\omega, \langle x,y\rangle)] \cup Pos°(\omega, \langle x,y+1\rangle)$
- $\mathbf{A}°(go_{south}, \omega, s, s^+)$ if
  $Pos°(\omega, \langle x,y\rangle) \in s \wedge s^+ = [s \setminus Pos°(\omega, \langle x,y\rangle)] \cup Pos°(\omega, \langle x,y-1\rangle)$
- $\mathbf{A}°(go_{west}, \omega, s, s^+)$ if
  $Pos°(\omega, \langle x,y\rangle) \in s \wedge s^+ = [s \setminus Pos°(\omega, \langle x,y\rangle)] \cup Pos°(\omega, \langle x-1,y\rangle)$
- $\mathbf{A}°(go_{east}, \omega, s, s^+)$ if
  $Pos°(\omega, \langle x,y\rangle) \in s \wedge s^+ = [s \setminus Pos°(\omega, \langle x,y\rangle)] \cup Pos°(\omega, \langle x+1,y\rangle)$
- $\mathbf{A}°(grab, \omega, s, s^+)$ if
  $Pos°(\omega, \langle x,y\rangle) \in s \wedge Waste°(\langle x,y\rangle, n) \in s \wedge Collected°(\omega, m) \in s \wedge$
  $s^+ = [\mathrm{s} \setminus Waste°(\langle x,y\rangle, n) \setminus Collected°(\omega, m)]$
  $\cup\, Waste°(\langle x,y\rangle, 0) \cup Collected°(\omega, m+n)$
- $\mathbf{A}°(pass, \omega, s, s^+)$ if $s^+ = s$

The utility predicate $U°$ assigns a very low value (meaning high utility) to the *grab* action if the agent's position contains waste; otherwise it assigns a high value to *grab*. The *pass* action is always assigned a high value, so that the agents only use it when no better actions can be found. For action go_{Dir}, the utility predicate assigns higher values for directions leading closer to a square containing waste. However, the details of the utility predicate will be left out here.

The turn-operator simply chooses the agent which comes after the current moving agent in an enumeration of the agent set, thus treating the agent set as a "circular queue" of agents:

- $NextToMove°(<\omega_i, s_{\mathbf{D}}>, \omega_{i+1})$ if $\Omega_{\mathbf{D}} = \{\omega_1, \ldots, \omega_i, \omega_{i+1}, \ldots\}$; and
- $NextToMove°(<\omega_n, s_{\mathbf{D}}>, \omega_1)$ if $\Omega_{\mathbf{D}} = \{\omega_1, \ldots, \omega_n\}$

where $\Omega_{\mathbf{D}}$ is the agent set of $\boldsymbol{D}$ in state s.

The feasible actions predicate *Feasible*° and the tie-breaker predicate *BreakTie*° are defined in exactly the same way as in the previous section, and the secondary predicates are completely determined by the primary predicates.

7 Implementation of dnrDALMAS

An overview of the implementation is given in this section. The *dnrDALMAS* framework is a Prolog module consisting of implementations of the predicates described in section 6, and a set of predicates designed to simplify the process of implementing a specific deterministic DALMAS. The module has been developed and tested under SICStus Prolog version 3.12.2. The source files will be available on SourceForge.net[11] and will be publicly and freely disseminated. A simple "users manual", describing how to use the *dnrDALMAS* module, is provided in appendix 1.

The implementation is modelled after the formal definition of a deterministic DALMAS given in section 5. In most cases, the Prolog code follows very naturally from the definitions in section 6. When this is not the case, some remarks will be given in the text.

A norm-regulated DALMAS is represented by the Prolog term `dnrDALMAS/2`, where the first argument is the descriptor of a deterministic DALMAS, and the second argument is a norm-system. The representation of the deterministic DALMAS will be examined in section 7.3, and the representation of norm-systems will be examined in section 7.4. A summary of the Prolog files that are part of *dnrDALMAS* is given in appendix 1.

7.1 States and State-conditions

In a Prolog environment, the state of a system may be represented *implicitly* or *explicitly*. In the first case, Prolog dynamically modifies its own internal database with predicates such as `assert/1` and `retract/1`. In the case of explicit representation of states, state descriptors are passed as arguments to the predicates of the program. The state arguments may for instance be terms or list containing known facts about the environment. The *dnrDALMAS* implementation uses the latter approach. A *dnrDALMAS state* is described by the term `dnrDALMAS_state/2`, consisting of two parts:

1. The current *knowledge base* of the system.
2. The `dnrDALMAS/2` term, representing the DALMAS itself.

Here we make an important observation: The term *state* has slightly different meanings in the Prolog implementation and in the abstract architecture. A "state" in the abstract architecture corresponds to a *state of the system's knowledge base* in the implementation. In the theory of DALMASes, the description of the DALMAS itself is not considered as a part of a state or of the state space. This is in contrast to the

[11] URL: http://www.sourceforge.net/ [project name: dnrdalmas]

Prolog implementation, where "state" refers to a `dnrDALMAS_state` term. This term includes the system's current knowledge base and a `dnrDALMAS/2` descriptor, which describes the DALMAS itself. Normally, state-conditions are conditions on the state of the knowledge base. Therefore, the user of *dnrDALMAS* usually defines the state-condition predicates in terms of queries to the knowledge base of the `dnrDALMAS_state` term. The *dnrDALMAS* framework contains a predicate `knowledgeBase/2` to give the user access to the system's knowledge base. Let `c` be the functor of a Prolog predicate which implements a state-condition predicate $c°(p, \omega_1, \ldots, \omega_x; s)$, where $p = \langle p_1, p_2, \ldots , p_n\rangle$ represents zero or more non-agent parameters. The clauses of `c` should generally have the following form:

```
c(State,P1,…,Pn,W1,…,Wx) :-
     knowledgeBase(State,KB),
     … %%% Code containing queries to the KB
```

where `State` is bound to a `dnrDALMAS_state` term, `W1` through `Wx` are agents and `P1` through `Pn` are (optional) non-agent parameters.

The form and implementation of the knowledge base is not specified by *dnrDALMAS*; the user may choose a suitable representation. In fact, if the user prefers an implicit representation of the state of the system's knowledge base, this argument may be unbound.

7.1.1 Examples

Let us examine the Prolog implementation of the state-condition predicates in section 6.5 and 6.6. The knowledge base for the COLOUR&FORM agents is represented as a Prolog list of `wState/4` terms, holding attribute information for each agent. For $Diff°(\omega_1, \omega_2; s)$, the implementation is almost trivial:

```
diff(_,W1,W2) :- W1 \== W2.
```

The state argument of `diff/3` is unbound, since this relation does not depend of the current state of the system. The implementation of $Eq°(\omega_1, \omega_2; s)$ is also fairly simple:

```
eq(State,W1,W2) :-
   dnrDALMAS:knowledgeBase(State,KB),
   member(wState(W1,_,C,F),KB),
   member(wState(W2,_,C,F),KB).
```

The knowledge base for the waste-collectors is represented as a Prolog list of known facts about the wasteland. *Lap°* is implemented by `overlap/4`, and the code for this predicate is shown in appendix 4.

7.2 Situations and Sit-conditions

A *dnrDALMAS* situation is represented by the term `dnrDALMAS_sit/2`. This term has two arguments: an agent identifier, and a `dnrDALMAS_state` term.

To form sit-conditions, operators are applied to state-conditions. A move-operator (section 5.2.1 and 5.2.2) is applied to a state-condition predicate `c` by a call to `move/6`. This is a second-order predicate which generates the term `m/5` or `m/6` representing the resulting sit-condition. They may in turn be meta-called as predicates using SICStus Prolog's built-in meta-predicate `call/1`. The `m/5` and `m/6` predicates are the Prolog implementations of $M°(c°, p, \langle\omega_1, ..., \omega_x\rangle, \langle\omega, ...\rangle, <\omega_m,s>)$ and $M°(n, c°, p, \langle\omega_1, ... , \omega_n, ..., \omega_x\rangle, \langle\omega, ...\rangle, <\omega_m,s>)$, which are described in section 6.4. The Prolog code for `move/6`, `m/5` and `m/6` is shown in appendix 2.

Similarly, a type-operator is applied to a state-condition `d` by a call to the predicate `t/6`. This is also a second-order predicate which generates the term `e/6` representing the corresponding sit-condition. Like `m/5` and `m/6`, this term may be meta-called using `call/1`. The `e/6` predicate is the implementation of $E°(2, d°, p, \langle\omega_1, ..., \omega_x\rangle, \langle\omega, a, ...\rangle, <\omega_m,s>)$. See section 6.4 for the definition of $E°$, and appendix 2 for the implementation of `t/6` and `e/6`.

Note that for simple DALMASes, the user of the *dnrDALMAS* framework does not have to define and implement sit-conditions. A simple DALMAS uses the default implementation of the deontic structure-operator, which is defined in terms of *Prohibited*. In this case, the *dnrDALMAS* automatically creates the corresponding sit-conditions by using the predefined move- and type-operators. However, the framework allows redefinition of the deontic structure-operator and the creation of user-defined operators if desired.

7.3 Implementation of a Deterministic DALMAS

The 9-tuple defining a deterministic DALMAS (section 5.3) is represented by a Prolog term `dDALMAS/9` whose first argument, corresponding to the agent set Ω, is represented by a Prolog list of agent identifiers. The second argument, which corresponds to the action set *A*, is a list of action names.

The other arguments hold the functors and arity of the primary predicates and some of the secondary predicates.

7.3.1 Primary Predicates

For each of the primary predicates, the *dnrDALMAS* framework has a corresponding Prolog predicate. These predicates are summarized in table 4.

Table 4. Primary Predicates and their dnrDALMAS Counterparts

Predicate	Generic Prolog Implementation	Default Implementation
$\mathbf{A}°(a, \omega, s, s^+)$	`actionResult/4`	–
$\mathit{Feasible}°(<\omega_m,s>, \omega, a)$	`feasibleAction/3`	`feasibleAction_default/3`
$U°(a, <\omega_m,s>, \omega, U_a)$	`utilityFunction/4`	–
$\mathit{NextToMove}°(<\omega_m,s>, \omega^+)$	`turnOperator/2`	`turnOperator_default/2`
$\mathit{BreakTie}°(\Gamma, <\omega_m,s>, a)$	`tieBreaker/3`	`tieBreaker_default/3`

The framework also contains predicates for updating the functor and arity of each of the currently registered primary predicates, making it possible to provide a new implementation of a primary predicate if desired. Therefore, `actionResult/4`, `feasibleAction/3`, `utilityFunction/4`, `turnOperator/2` and `tieBreaker/3` are implemented as generic second-order predicates that make meta-calls to the corresponding user-defined or default implementations, as shown in the code in appendix 2.

The default implementations are provided for the convenience of the user, and may be useful in many specific implementations. The action result predicate and the utility function have no default implementations, since they are completely dependent on the specific *dnrDALMAS* to be created.

Similarly, the framework contains predicates for asking the system for the functor and arity of each of the currently registered primary predicates.

7.3.2 Secondary Predicates

The secondary predicates (see section 6.2.2) are summarized in table 5 together with their implementation counterparts. The framework also contains predicates for updating the functor and arity of some of the currently registered secondary predicates.

Table 5. Secondary Predicates and their dnrDALMAS Counterparts

Predicate	**Generic Prolog Implementation**	**Default Implementation**
FeasibleActions°(<ω_m,*s*>, ω, *A*)	`feasibleActions/3`	`Calls findall/3 on feasibleAction/3`
Permissible°(*A*, <ω_m,*s*>, ω, δ)	`permissibleAction/4`	`permissibleAction_default/4`
DeonticStructure°(*A*, <ω_m,*s*>, ω, Δ)	`deonticStructure/4`	`Calls findall/3 on permissibleAction/4`
PreferenceStructure°(*A*, <ω_m,*s*>, ω, Π)	`preferenceStructure/4`	`Calls setof/3 on utilityFunction/4`
ChoiceSet°(*A*, <ω_m,*s*>, ω, Γ)	`choiceSet/4`	`choiceSet_default/4`

`permissibleAction/4` and `choiceSet/4` are second-order predicates which make meta-calls to the corresponding user-defined or default implementations.

The implementation of `feasibleActions/3` follows directly from the definition of *FeasibleActions*°: the built-in Prolog predicate `findall/3` is called on the primary predicate `feasibleAction/3`, creating a list of all feasible actions.

Similarly, for `deonticStructure/4` we use `findall/3` on the secondary predicate `permissibleAction/4`. The result is a list of all permissible actions.

The implementation of `preferenceStructure/4` follows from the definition of *PreferenceStructure*°. The predicate should create a set of acts ordered by their utility for the acting agent. To accomplish this, we let `utilityFunction/4` create a "utility structure" `p(U,A)` such that `U` is the utility (according to the utility predicate) of act `A`. Then we may use the built-in predicate `setof/3` to create a list of acts ordered by the Prolog Standard Total Ordering (see [18, pp. 168f]) of the utility term. See appendix 10 for the code.

The implementation of `choiceSet_default/4` follows the pattern suggested for *ChoiceSet*° in section 6.2.2: The choice-set consists of a preference ordering of the deontic structure, which in turn is a subset of the set of feasible acts. For `permissibleAction_default/4` the implementation follows the pattern for *DeonticStructure*°: An act is permissible if it is a member of the set of feasible acts, and not prohibited according to the predefined implementation of `prohibited/3`. The code for the default implementations is shown in appendix 11.

The remaining secondary predicates are treated in a similar way: *NextSituation*° is implemented by `nextSituation/4`, *Run*° is implemented by `run/3` and *PRun*° is implemented by `partialRun/5`. See appendix 11 for the code.

7.4 Implementation of Norm-systems

A norm-system (see section 6.3) is represented by a term `ns/3`, where the first argument holds the ground-set *C*, the second argument holds the consequence-set *D* and the third argument holds the norm-set ***J*** of a particular norm-system. The framework provides predicates for and getting and updating (replacing) the norm-system of a `dnrDALMAS/2` descriptor: `normSystem/2` and `normSystemUpdate/3`. Table 6 summarizes the predicates provided for getting and updating the current ground-set, consequence-set and norm-set of a norm-system:

Table 6. Norm-system predicates

Predicate	**Get**	**Update (replace)**
GroundSet°	`groundSet/2`	`updateGroundSet/3`
ConsequenceSet°	`consequenceSet/2`	`updateConsequenceSet/3`
NormSet°	`normSet/2`	`updateNormSet/3`

7.4.1 Implementation of Condition-Implication Structures

The ground-set and consequence-set of a norm-system are represented as association lists, using the `assoc` library [18, pp. 353f] in SICStus Prolog. The elements of a ground-set and the consequence-set are of the form `CID-Mod:CF/PA/AA`, where `CID` is a unique identifier of conditions, `CF` is the functor of the corresponding state-condition predicate, which is defined in module `Mod` and has `PA` parameters and `AA` agent arguments. A condition is added to the ground-set by calling `addGroundToGS/6`, and to the consequence-set by calling `addConsequenceToCS/6`. The user provides the information needed, and this information is then added to the association list by a call to the library routine `put_assoc/4`. The implementations of *Ground*° and *Consequence*° work the other way around: the identifier of a condition is used to pick out the corresponding state-condition predicate from the association list of grounds or consequences. The code for these predicates is shown in appendix 12.

The form of a ground `G` is `m*C` or `m(N)*C`, where `m` represents the move-operator *M* and `m(N)` represents the move-operator M_n. `C` is a unique identifier of a state-condition predicate, which may be used for looking up the corresponding functor and arity in the ground-set.

For elementary norms, the form of a consequence C is `t(I)*D`, where `t(I)` represents the type-operator T_i, and `D` is an identifier of a state-condition predicate in the consequence-set.

A non-elementary norm is of the same form, but the consequence may be a negation or a disjunction of other consequences:

1. `Consequence = t(I)*D`, where `t(I)` is the representation of type-operator T_i and `D` is an identifier of a state-condition predicate in the consequence-set.
2. `Consequence = not(C)`, where `C` is a consequence.
3. `Consequence = or(C1,C2)`, where `C1` and `C2` are consequences.
4. `Consequence = and(C1,C2)`, where `C1` and `C2` are consequences.

This syntax of grounds and consequences is motivated by the definition of *m-cis* and *np-cis*; see section 5.4.2. Note that C_M is implicitly represented by all possible applications of `m` and `m(N)` to all conditions in the ground-set *C*, and the set of normative positions D_T is implicitly represented by all possible applications of `t(I)` to all conditions in the consequence-set *D*, and D^*_T is implicitly defined by all possible applications of 1 – 3 to all conditions in *D*. Examples of norms expressed with this syntax will be given in sections 7.6 and 7.7.

The semantics of grounds and consequences is determined by how this syntax is interpreted in the implementation of *Prohibited*°. This will be the topic of the following sections.

To prepare for future development, the framework allows norms which are defined in terms of other (user-defined) operators than the move- and type operators. Also, grounds that are disjunctions or conjunctions of simpler grounds are allowed.

7.4.2 Implementation of Norms and Norm-sets

A norm (section 6.3.2) is represented by a term `n/3` having the form `n(ID/N,Ground,Consequence)`, where `ID` is an identifier of a norm-system, `N` is an identifier of a specific norm, and `Ground` and `Consequence` are bound to terms following the syntax described in the previous subsection.

The norm-set ***J*** of a *dnrDALMAS* is represented as an ordered set of `n/3` terms, using the `ordsets` library [18, pp. 371f] in SICStus Prolog. The ordering is determined by the Prolog Standard Total Ordering of the norm identifiers. The implementation of *Norm*° is straightforward:

```
norm(NS,ID/N,G,C) :-
   normSet(NS,J),
   member(n(ID/N,G,C),J).
```

The framework also contains predicates for adding norms to and removing norms from a norm-system, and a predicate for getting and updating (replacing) the norm-system of a *dnrDALMAS*: `addNorm/5`, `removeNorm/5`, `normSystem/2` and `normSystemUpdate/3`.

7.5 Implementation of Prohibited

The definition of *Prohibited*°(*a, ω,* <ω_m,*s*>) in section 6.4 is the model for the Prolog implementation `prohibited/3`. There are, however, a couple of issues that need to be resolved. First, the consequences of non-elementary norms have a variable number of disjuncts of `t(I,D)`, which means that the number of calls to the corresponding `e(I,D)` is not known in advance. Second, an act is prohibited if and only if both the ground and the consequence of a norm are satisfied for the same configuration of arguments. Therefore, the implementation must first try to satisfy the ground of a norm for a particular configuration of arguments. If the ground is satisfied for this configuration, then the implementation must try to satisfy the consequence of the current norm for the same configuration. If the arity *a* of a particular sit-condition is less than the arity of the previous sit-condition of the same norm, then only the *a* first agent arguments should be used. On the other hand, if *a* is greater than the arity of the previously tried sit-condition, then additional agents must be added to the argument list. See the discussion in section 5.4.3. If the ground or the consequence is not satisfied for a particular configuration of arguments, then another configuration should be tried.

To resolve these issues, `prohibited/3` calls the auxiliary predicates `groundSatisfied/6` and `consequenceSatisfied/6`. See appendix 12 for the implementation of these predicates. The code for `prohibited/3` is given here:

```
prohibited(Act,W,Sit) :-
    state(Sit,State),
    normSystem(State,NS),
    groundSet(NS,GS),
    consequenceSet(NS,CS),
        %%% Try norm N:
    norm(NS,N,Gr,Co),
        %%% Is the ground of norm N met for W
        %%% in situation Sit?
    groundSatisfied(Gr,GS,[],[W],Sit,WArgs2),
        %%% Is the consequence of norm N met for W
        %%% performing Act in situation Sit?
    consequenceSatisfied(Co,CS,WArgs2,[W,Act],Sit,_),
        %%% Then Act is prohibited for W
        %%% in situation Sit!
```

For a specific act, the system picks a norm in the norm-system, and then calls `groundSatisfied/6`. The configuration of arguments is empty from the beginning, but during execution the argument list will grow to a list containing *a* agent identifiers, where *a* is the maximum "agent arity" of the conditions in the ground of this specific norm. If the call fails, Prolog backtracks to try to find another configuration of agent arguments that makes the condition of the ground true. Otherwise, if the conditions of the ground are met for some configuration of agent arguments, the argument list is passed on to `consequenceSatisfied/6`. Since the consequence may be a chain of disjunctions of `t(I,D)`, this predicate checks for each such `t(I,D)` if the conditions of the consequence are met for the same configuration of agent arguments. Again, the argument list will grow to a list containing *b* agent identifiers, where *b* is the maximum “agent arity” of the conditions in the consequence.

If `consequenceSatisfied/6` succeeds, then `prohibited/3` immediately succeeds, which means that the act is prohibited. Otherwise Prolog backtracks to try to find another configuration of agent arguments that make the ground and the consequence true. If this does not succeed, the system backtracks to find another norm that is tried in the same way. If all norms have been tried for all possible configurations of agent arguments, and none of them have succeeded, then `prohibited` fails, which means that the act is *not* prohibited.

7.6 Prolog Implementation of the COLOUR&FORM system

The code for the state-condition predicates of the C&F system has already been shown in section 7.1.1. The norm-system for this system (see section 5.4.4) is created as follows:

```
initialNormSystem(NormSystem) :-
   emptyNormSystem(NS),
   addGroundToGS(diff,user:diff,0,2,NS,NS2),
   addConsequenceToCS(eq,user:eq,0,2,NS2,NS3),
   addNorm(cf/n1,move(1)*diff,t(7)*eq,NS3,NormSystem).
```

The rest of the code, which is presented in appendix 3, follows naturally from the predicate specifications of the systems (see section 6.5). It will not be further commented here.

An example of a 4-event run of the COLOUR&FORM system is shown and commented in appendix 3.

7.7 Prolog Implementation of the WASTE-COLLECTORS system

The whole implementation of the WASTE-COLLECTORS system is available at *SourceForge.net.* The code follows the specifications in section 6.6, but the implementation of some predicates (such as *Lap*°) is sometimes a little more complicated (mostly due to technicalities in Prolog) than is suggested in the text. The specification of the norm-system for the waste-collectors is shown in appendix 4.

A very simple 2-event run of the WASTE-COLLECTORS implementation is shown in appendix 4.

8 Discussion

Since a DALMAS is a global clock, global state and global dynamics system, it does not meet the four criteria listed in section 2. However, if the action set contains a *pass* action, and very short clock cycles are used, then the system may obtain a behaviour which is close to asynchronous. In a DALMAS, all agents share the same knowledge of the environment by sharing the same knowledge base. The *dnrDALMAS* framework does not explicitly support "local-state" systems, where each agent has its own knowledge base. It is, however, worth noting that "local-state" systems are not prevented either, since the form of the knowledge base is not specified. For instance, the user may let the global knowledge base consist of a list of individual knowledge bases for each agent if that is desirable. In such a system, agent communication may become an important issue, but this is beyond the scope of this text. We may note, however, that there is some resemblance between the global knowledge base of a DALMAS and the Social Environment Knowledge Base (SEKB) which is part of the agent society model presented in [2], whereas the norm-system of a DALMAS could be seen as part of the Social Organization Knowledge Base (SOKB).

Several tests have been performed to verify that the general-level implementation correctly handles elementary norms. The tests follow the pattern used for the tests performed in [15]. The tests, and their result, are described in appendix 4. The result shows that elementary norms are handled correctly by the implementation. For non-elementary norms, inspections of different runs of the WASTE-COLLECTORS system suggest that the implementation also handles non-elementary norms correctly.

The tests suggest that the "act position operators" E_i^a, as defined in section 5.2.3, are correctly implemented in the sense that an act a is prohibited by a certain type T_i if and only if the corresponding E_i^a is satisfied. The definition of E_i^a is the result of an analysis (see [14, pp. 160f]) of the intended meaning of the action-operator *Do*. However, in a deterministic environment the interpretation of *Do* (and, more specifically, *Pass*) is not as unambiguous as might be expected. Therefore, there may be other feasible ways to define E_i^a. This does not affect the current version of *dnrDALMAS*, but may result in future revision and/or development of the theory for DALMAS and of the implementation. [13]

9 Conclusion and Future Work

A general-level Prolog implementation of the DALMAS was presented, together with two specific DALMAS implementations. The first implementation was a very simple two-agent DALMAS for the COLOUR&FORM system. The second implementation was a DALMAS for the WASTE-COLLECTORS system, which was used as an example in [14].

The specific implementations have been tested, and the tests suggest that the general-level Prolog module correctly implements the abstract model. This in turn shows that the DALMAS may be turned into a useful executable model for norm-regulated multi-agent systems. It also gives motivation for further development of the theory for DALMAS and a deeper theoretical analysis of, for example, the "act position operators" E_i^a and the interpretation of *Do* and *Pass* in a static and deterministic environment.

Applications of DALMAS can be found in many areas. An interesting application within the area of forest cleaning has been proposed by Ahonen-Jonnarth and Odelstad [1], and other possible applications have been suggested by Odelstad and Boman [14]:

For a specific DALMAS ***D***, which normative system would give the most efficient behaviour of the system? Together with a function that evaluates the result of a run of ***D***, and some mechanism of learning, such as a genetic algorithm, it is possible to create a system that itself determines the optimal normative system for a particular task. [14, pp. 164-165]. The ground-set *C* and consequence-set *D* would constitute a "vocabulary" of the system. Given such a vocabulary, the norm-system ***J*** could be represented with some data structure joining these two sets, making it easy to test all possible sets of minimal norms.

Other possible extensions have also been suggested. For instance, the descriptive conditions studied in [14] are conditions on agents in a state. Which changes of the *dnrDALMAS* framework are necessary if the architecture incorporates generalized conditions on both agents and other objects?

The DALMAS architecture uses the theory of one-agent type-operators, which is only a small part of Kanger-Lindahl theory of normative positions. An interesting question is which changes have to be made to *dnrDALMAS* if it is developed to incorporate the theory of two-agent type-operators [8], as suggested in [14, p. 164].

In the current formulation, the norms are constituitive: it is not possible for the agents to break the rules. A possible extension would be to view the normative system as regulative, making it possible for agents to break the rules. Such behaviour could in turn give another agent the right to "punish" it or perform a "reaction act". For instance, some agents could function as "policemen". This would require specification of norms regulating the behaviour of such agents, which in turn may require definition of special kinds of operators. Since the *dnrDALMAS* architecture allows the definition of other operators than the move- and type-operators, the framework could be used to study such systems.

Finally, alternative ways of representing norms could be investigated, such as use of the Constraint Handling Rules language. Would it be possbile to represent norms and create a better implementation of `prohibited/3` in a manner similar to the ideas in [2]?

10 References

1. Ahonen-Jonnarth, U. and Odelstad, J., Simulation of cleaning of young forest stands. CML Technical Report no. 2005:02. Creative Media Lab, University of Gävle, Gävle, 2005.
2. Alberti et al., *Logic Based Semantics for an Agent Communication Language*. DEIS Technical Report no. DEIS-LIA-03-001. LIA Series no. 62.
3. Arisha, K.A., Ozcan, F., Ross, R., Subrahamanian, V.S., Eiter, T. and Kraus, S., *IMPACT: A platform for collaborating agents*, IEEE Intelligent Systems 14(2): 64-72, 1999.
4. Boman, M., *Norms in Artificial Decision Making*, Artificial Intelligence & Law 7(1), 17—35, 1999.
5. Dignum, F., *Autonomous Agents with Norms*, Artificial Intelligence & Law 7(1): 69—79, 1999.
6. Jones, A. JI, Sergot, M.J. On the Characterisation of Law and Computer Systems: The Normative Systems Perspective. Ch. 12 in Meyer, J-H.Ch., Wieringa, RJ. (eds.) *Deontic Logic in Computer Science: Normative System Specification.* Wiley, 1993.

7. Kanger, S., New foundations for ethical theory, Part 1 (1957), in: Holmström-Hintikka, et al. (Eds.), Collected Papers of Stig Kanger With Essays on His Life and Work, vol. 1, Kluwer, Dordrecht, 2001, pp. 99-119.
8. Lindahl, L., *Position and Change: A study in Law and Logic*, Reidel, Dordrecht, 1977.
9. Lindahl, L. and Odelstad, J., *An Algebraic Analysis of Normative Systems.* Ratio Juris Vol. 13 No. 3 September 2000, 261-278.
10. Lindahl, L. and Odelstad, J., *Normative Positions within an Algebraic Approach to Normative Systems.* Journal of Applied Logic 2 (2004) 63-91.
11. Luger, G.F. and Stubblefield, W.A. *Artificial Intelligence. Structures and Strategies for Complex Problem Solving*, third ed. Addison-Wesley, 1998.
12. Morrone, G., *Implementation av normer för multiagent-system i Prolog.* Candidate Thesis, University of Gävle, 2004.
13. Odelstad, J., personal communication, 2006.
14. Odelstad, J. and Boman, M., *Algebras for Agent Norm-Regulation.* Annals of Mathematics and Artificial Intelligence 42 (2004) 141-166.
15. Olsson, J., *Normsystem för Wastecollectors-systemet.* Candidate Thesis, University of Gävle, 2006.
16. Russel, S. and Norvig, R., *Artificial Intelligence – A modern approach.* Prentice Hall, 1995.
17. Sergot, M.J., *A computational theory of normative positions*, ACM Trans. Comput. Logic (TOCL) 2 (2001) 581-622.
18. SICStus Prolog User's Manual. The Intelligent Systems Laboratory, Swedish Institute of Computer Science, Kista, 2005.
19. Steels, L., Cooperation between distributed agents through self organization. In proc. of Maamaw 89, Decentralized AI. Y. Demazeau et JP Muller, eds, pp,175--196,. Elsevier Science Publisher, 1990.
20. Verhagen, H., *Norm Autonomous Agents*. Doctoral Thesis, Department of Computer and System Sciences, Stockholm University, Stockholm, 2000.
21. Vázquez-Salceda, J., Aldewereld, H. and Dignum, F., *Implementing Norms in Multiagent Systems.* In G. Lindemann, J. Denzinger, I. J. Timm and R. Unland (eds.) Proc MATES 2004, LNAI 3187, pp. 313-327, Springer-Verlag, 2004.
22. Vázquez-Salceda, J., Aldewereld, H. and Dignum, F., *Norms in Multiagent Systems: from Theory to Practice*, Computer Systems Science & Engineering, 20(4), pp. 225—236, 2005.
23. Wooldridge, M., *An Introduction to MultiAgent Systems.* Wiley, Chichester, 2002.

11 Appendices

11.1 Appendix 1: Using dnrDALMAS

11.1.1 Overview of the dnrDALMAS Framework

The *dnrDALMAS* implementation consists of a Prolog module named `dnrdalmas`, defined in the following files:

1. `dnrDALMAS.pl`: Predicates for manipulation of situations, states, and DALMAS descriptors, and generic implementations of primary and secondary predicates.
2. `dnrDALMAS_cis.pl`: Predicates for manipulation of ground-sets and consequence-sets (condition-implication structures) and application of operators to state-conditions.
3. `dnrDALMAS_mcis.pl`: Definition of the move-operators.
4. `dnrDALMAS_npcis.pl`: Definition of the e-operator.
5. `dnrDALMAS_normsystem.pl`: Predicates for manipulation of norm-systems, norm-sets and norms.
6. `dnrDALMAS_defaults.pl`: Predefined implementations of some primary and secondary predicates.

   Selected code from these files is presented in appendix 2 (section 11.2).

11.1.2 Exported Predicates

The dnrDALMAS module contains a set of predicates for querying and updating dnrDALMAS situations, states and dnrDALMAS descriptors. These predicates are listed in table 7, and the code for two of them is presented in section 11.2.1.

Table 7. Predicates for Manipulation of Situations, States and dnrDALMAS Descriptors.

Name	**Purpose**
initialSituation/3	Create an initial situation from a state and an agent to move.
initialState/4	Create an initial state from an initial knowledge base, a norm-system and an initial DDALMAS descriptor.
initialDDALMAS/10	Create an initial DDALMAS descriptor from an initial agent set, an initial action set and the functors and arity of primary and secondary predicates.
atMove/2	Get the agent to move in a given situation.
atMoveUpdate/3	Create an updated situation from a previous situation and a new agent to move.

Name	Purpose
state/2	Get the state corresponding to a given situation.
stateUpdate/3	Create an updated situation from a previous situation and a new state.
knowledgeBase/2	Get the knowledge base corresponding to a given state.
knowledgeBaseUpdate/3	Create an updated state from a previous state and a new knowledge base.
dnrDALMAS/2	Get the dnrDALMAS descriptor corresponding to a given state.
dnrDALMASUpdate/3	Create an updated state from a previous state and a new dnrDALMAS descriptor.
dDALMAS/2	Get the dDALMAS descriptor corresponding to a given state.
dDALMASUpdate/3	Create an updated state (or dnrDALMAS descriptor) from a previous state (or dnrDALMAS descriptor) and a new dDALMAS descriptor.
normSystem/2	Get the norm-system corresponding to a given state.
normSystemUpdate/3	Create an updated state (or dnrDALMAS descriptor) from a previous state (or dnrDALMAS descriptor) and a new norm-system.
agentSet/2	Get the agent set corresponding to a given state.
agentSetUpdate/3	Create an updated state (or dnrDALMAS descriptor) from a previous state (or dnrDALMAS descriptor) and a new agent set.
actionSet/2	Get the action set corresponding to a given state.
actionSetUpdate/3	Create an updated state (or dnrDALMAS descriptor) from a previous state (or dnrDALMAS descriptor) and a new action set.

The dnrDALMAS module also contains predicates for retrieving and updating primary and secondary predicates. They are shown in table 8, and selected code is shown in section 11.2.2.

Table 8. Predicates for Manipulation of Primary and Secondary Predicates.

Name	Purpose
actionResultPredicate/2	Get the functor and arity of the system's action result predicate from a given dDALMAS descriptor.
actionResultPredicateUpdate/3	Register a new action result predicate by creating an updated dDALMAS descriptor from a previous dDALMAS descriptor and the new functor and arity of this predicate.
feasibleActionPredicate/2	Get the functor and arity of the system's feasible action predicate from a given dDALMAS descriptor.

Name	**Purpose**
feasibleActionPredicateUpdate/3	Register a new feasible action predicate by creating an updated dDALMAS descriptor from a previous dDALMAS descriptor and the new functor and arity of this predicate.
permissibleActionPredicate/2	Get the functor and arity of the system's permissible action predicate from a given dDALMAS descriptor.
permissibleActionPredicateUpdate/3	Register a new permissible action predicate by creating an updated dDALMAS descriptor from a previous dDALMAS descriptor and the new functor and arity of this predicate.
utilityFunctionPredicate/2	Get the functor and arity of the system's utility function predicate from a given dDALMAS descriptor.
utilityFunctionPredicateUpdate/3	Register a new utility function predicate by creating an updated dDALMAS descriptor from a previous dDALMAS descriptor and the new functor and arity of this predicate.
choiceSetPredicate/2	Get the functor and arity of the system's choice set predicate from a given dDALMAS descriptor.
choiceSetPredicateUpdate/3	Register a new choice set predicate by creating an updated dDALMAS descriptor from a previous dDALMAS descriptor and the new functor and arity of this predicate.
turnOperatorPredicate/2	Get the functor and arity of the system's turn operator predicate from a given dDALMAS descriptor.
turnOperatorPredicateUpdate/3	Register a new turn operator predicate by creating an updated dDALMAS descriptor from a previous dDALMAS descriptor and the new functor and arity of this predicate.
tieBreakPredicate/2	Get the functor and arity of the system's tie-break predicate from a given dDALMAS descriptor.
tieBreakPredicateUpdate/3	Register a new tie-break predicate by creating an updated dDALMAS descriptor from a previous dDALMAS descriptor and the new functor and arity of this predicate.

The dnrDALMAS module contains generic implementations for the primary and secondary predicates, as shown in table 9. Section 11.2.4 and 11.2.6 presents the code for these predicates. The framework also contains default implementations of some primary and secondary predicates. The code for these predicates is shown in section 11.2.3 and 11.2.5. The code for `prohibited/3` is discussed in 7.5.

Table 9. Generic and Default Implementations of Primary and Secondary Predicates

Generic implementation	**Default implementation**	**Description**
feasibleAction/3	*feasibleAction_default/3*	See section 7.3.1.
feasibleActions/3	-	See section 7.3.2.
permissibleAction/4	*permissibleAction_default/3*	See section 7.3.2.
deonticStructure/4	-	See section 7.3.2.
utilityFunction/4	-	See section 7.3.1.
preferenceStructure/4	-	See section 7.3.2.
choiceSet/4	*choiceSet_default/3*	See section 7.3.2.
turnOperator/2	*turnOperator_default/2*	See section 7.3.1.
tieBreaker/3	*tieBreaker_default/3*	See section 7.3.1.
nextSituation/4	-	See section 7.3.2.
run/3	-	See section 7.3.2.
partialRun/5	-	See section 7.3.2.
prohibited/3	-	See section 7.5.

For manipulation of norm-systems, ground-sets and consequence-sets, the dnrDALMAS module contains the predicates listed in table 10. See section 7.4 for a discussion of some of these predicates. Selected code is shown in 11.2.7.

Table 10. Manipulation of Norm-Systems, Ground-Sets and Consequence-Sets

Name	**Purpose**
emptyNormSystem/1	Create an empty norm-system.
groundSet/2	Get the ground-set of a given norm-system.
updateGroundSet/3	Create an updated norm-system from a previous norm-system and a new ground-set.
addGroundToGS/6	Add the functor and arity of a state-condition predicate to a ground-set.
consequenceSet/2	Get the consequence-set of a given norm-system.
updateConsequenceSet/3	Create an updated norm-system from a previous norm-system and a new consequence-set.
addConsequenceToCS/6	Add the functor and arity of a state-condition predicate to a consequence-set.
normSet/2	Get the norm-set of a given norm-system.
updateNormSet/3	Create an updated norm-system from a previous norm-system and a new norm-set.
addNorm/5	Add a norm to a norm-set.
removeNorm/5	Remove a norm from a norm-set.
norm/4	Get a norm from a norm-set.

11.1.3 Steps

The following steps must be performed when using the dnrDALMAS framework to create a deterministic norm-regulated DALMAS:

1. Provide implementations of user-defined predicates.
2. Create a `dDALMAS/9` structure holding the agent set, action set and names and arity of predefined and user-defined predicates to be used by the dnrDALMAS.
3. Create a norm-system (including a ground-set, a consequence-set and a norm-set) for the dnrDALMAS.
4. Create a `dnrDALMAS_state/2` structure representing the initial state of the system.

Step 1
The first step is to provide implementations of user-defined primary and secondary predicates, such as the action result predicate and the various state-conditions that are used to specify norms in the norm-system.

Step 2
A `dDALMAS` structure is created by calling `initialDDALMAS/10`, with the agent set and action set (ordered lists) as parameters together with the functors and arity of the basic primary and secondary predicates; see section 7.3.

Some of the predicates to be registered, such as the action result predicate, are completely domain-dependent. The user must provide implementations of these predicates, as described in step 1. For other predicates, such as the deontic structure-operator, the *dnrDALMAS* framework contains predefined predicates (see appendix 2) that may be used if desired.

Step 3
To create a norm-system, an empty norm-system is first created with `emptyNormSystem/1`. Then the functors and arity of the predicates in the ground-set are added by calling `addGroundToGS/6`, and the functors and arity of the consequence-set predicates are added by calling `addConsequenceToCS/6`. Finally, norms are added to the norm-set by calls to `addNorm/5`.

Step 4
An initial state is created by a call to `initialState/4`, with the initial knowledge base, the initial norm-system and the initial `dDALMAS` structure as parameters.

11.2 Appendix 2: Selected dnrDALMAS Code

This section presents selected code for the dnrDALMAS implementation. Brief descriptions of the predicates are given in 11.1.2. For a more detailed discussion of the implementation of a particular predicate, follow the reference.

11.2.1 Retrieving and Updating Information about States and Situations

Predicate: `knowledgeBase/2`
Reference: 7.1
Code:

```
%%% knowledgeBase(+<KB,DnrDALMAS>,?KB) :-
% <KB,DnrDALMAS> = dnrDALMAS_state(KB,DnrDALMAS)
% and KB is the state of the system's knowledge base
%
knowledgeBase(dnrDALMAS_state(KB,_),KB).
```

Predicate: `knowledgeBaseUpdate/3`
Code:

```
%%% knowledgeBaseUpdate(+<_,DnrDALMAS>,+KB,?<KB,DnrDALMAS>) :-
% <KB,DnrDALMAS> = dnrDALMAS_state(KB,DnrDALMAS)
% and <KB,DnrDALMAS> is an updated state with knowledge base KB
%
knowledgeBaseUpdate(dnrDALMAS_state(_,DnrDALMAS),NewKB,
dnrDALMAS_state(NewKB,DnrDALMAS)).
```

11.2.2 Retrieving and Updating Primary or Secondary Predicates

Predicate: `actionResultPredicate/2`
Reference: 7.3.1
Code:

```
%%% actionResultPredicate(+<..., ARP/A, ...>,?ARP/A) :-
% <..., Module:ARP/A, ...> = dDALMAS(..., Module:ARP/A, ...)
% and ARP is the functor of a predicate with arity A
% defined in module Module
%
actionResultPredicate(dDALMAS(_,_,Module:ARP/A,_,_,_,_,_,_),
Module:ARP/A).
```

Predicate: `actionResultPredicateUpdate/3`
Reference: 7.3.1
Code:

```
%%% actionResultPredicateUpdate(+DDALMAS,+ARP/A,-NewDDALMAS) :-
% DDALMAS = dDALMAS(...)
% and ARP is the functor of a predicate with arity A
%
actionResultPredicateUpdate(dDALMAS(Ws,As,_,F2,F3,F4,F5,F6,F7),
Module:ARP/A, dDALMAS(Ws,As,Module:ARP/A,F2,F3,F4,F5,F6,F7)).
```

11.2.3 Generic Implementations of Primary Predicates

Predicate: `actionResult/4`
Reference: 7.3.1
Code:

```
%%% actionResult(?A,+W,+S,?SPlus) :-
% SPlus is the resulting state if W performs action A in state S
%
% Corresponds to the “action funtion” a(A,W,S,SPlus)
%
% PRECONDITION:
% A unifies with a feasible action for agent W,
% and W is an agent in the agent set,
% and S is a well-formed dnrDALMAS state
%
actionResult(A,W,S,SPlus) :-
       dDALMAS(S,DDALMAS),
       actionResultPredicate(DDALMAS,Module:FnAR/N),
       functor(FnARGoal,FnAR,N),
       arg(1,FnARGoal,A),
       arg(2,FnARGoal,W),
       arg(3,FnARGoal,S),
       arg(4,FnARGoal,SPlus),
       call(Module:FnARGoal).
```

Predicate: `feasibleAction/3`
Reference: 7.3.1
Code:

```
%%% feasibleAction(+Sit,+W,?FA) :-
% FA is a feasible action for agent W in situation Sit
%
% PRECONDITION:
% Sit is a well-formed dnrDALMAS situation
% and W is an agent in the agent set
%
feasibleAction(Sit,W,FA) :-
       state(Sit,S),
       dDALMAS(S,DDALMAS),
       feasibleActionPredicate(DDALMAS,Module:FnFA/N),
       functor(FnFAGoal,FnFA,N),
       arg(1,FnFAGoal,Sit),
       arg(2,FnFAGoal,W),
       arg(3,FnFAGoal,FA),
       call(Module:FnFAGoal).
```

Predicate: `utilityFunction/4`
Reference: 7.3.1
Code:

```
%%% utilityFunction(+As,+Sit,?W,?p(Value,A)) :-
% Value is the utility of action A
% for agent W in situation Sit
% according to the FnU/N predicate,
% where A is a member of the action-set As
%
% PRECONDITION:
% As is a subset of the set of feasible actions
% and Sit is a well-formed dnrDALMAS situation
% and W unifies with an agent in the agent set
%
utilityFunction(As,Sit,W,p(Value,A)) :-
        state(Sit,S),
        dDALMAS(S,DDALMAS),
        utilityFunctionPredicate(DDALMAS,Module:FnU/N),
        functor(FnUGoal,FnU,N),
        member(A,As),
        arg(1,FnUGoal,A),
        arg(2,FnUGoal,Sit),
        arg(3,FnUGoal,W),
        arg(4,FnUGoal,Value),
        call(Module:FnUGoal).
```

Predicate: `turnOperator/2`
Reference: 7.3.1
Code:

```
%%% turnOperator(+Sit,?NextWm) :-
% NextWm is the agent that is next to move
% after the current moving agent
%
% PRECONDITION:
% Sit is a well-formed dnrDALMAS situation
% and the FnTO/N predicate finds the agent that is to move
% in the next situation
%
turnOperator(Sit,NextWm) :-
        state(Sit,S),
        dDALMAS(S,DDALMAS),
        turnOperatorPredicate(DDALMAS,Module:FnTO/N),
        functor(FnTOGoal,FnTO,N),
        arg(1,FnTOGoal,Sit),
        arg(2,FnTOGoal,NextWm),
        call(Module:FnTOGoal).
```

Predicate: `tieBreaker/3`
Reference: 7.3.1
Code:

```
%%% tieBreaker(+As,+Sit,?Action) :-
% Action is the action chosen from the choice-set As
%
tieBreaker(As,Sit,Action) :-
       state(Sit,S),
       dDALMAS(S,DDALMAS),
       tieBreakPredicate(DDALMAS,Module:FnTB/N),
       functor(FnTBGoal,FnTB,N),
       arg(1,FnTBGoal,As),
       arg(2,FnTBGoal,Sit),
       arg(3,FnTBGoal,Action),
       call(Module:FnTBGoal).
```

11.2.4 Default Implementations of Primary Predicates

Predicate: `feasibleAction_default/3`
Reference: 7.3.1
Code:

```
%%% feasibleAction_default(+Sit,+W,?Action) :-
% Action is a feasible action for the agent W in Sit
%
% Default: All actions are always feasible for the agent
% to move in DALMAS (and no actions are feasible for any
% other agent)
%
% PRECONDITIONS:
% Sit is a well-formed dnrDALMAS situation
% and W is an agent in the agent set
%
feasibleAction_default(Sit,W,Action) :-
       state(Sit,S),
       atMove(Sit,W),
       actionSet(S,As),
       member(Action,As).
```

Predicate: `turnOperator_default/2`
Reference: 7.3.1
Code:

```
%%% turnOperator_default(+Sit,?NextWm) :-
% NextWm is the next agent to move after situation Sit
%
% Default: The order is determined by the order in
% the agent set (simple DALMAS)
%
turnOperator_default(Sit,NextWm) :-
       atMove(Sit,Wm),
       state(Sit,S),
       agentSet(S,As),
       nextToMove(As,As,Wm,NextWm).
```

Predicate: `nextToMove/3 (auxiliary predicate)`
Code:

```
%%% nextToMove(+As,+As,+Wm,?NextWm) :-
% NextWm is the agent that follows Wm
% in the agent set As (with wrap-around)
%
% Base case 1: Wm is the last agent in the agent list
%
nextToMove([Wm],[NextWm|_],Wm,NextWm).

% Base case 2:
%
nextToMove([Wm,NextWm|_],_,Wm,NextWm).

% Recursive case:
%
nextToMove([W,W2|Ws],As,Wm,NextWm) :-
       W \== Wm,
       nextToMove([W2|Ws],As,Wm,NextWm).
```

Predicate: `tieBreaker_default/4`
Reference: 7.3.1
Code:

```
%%% tieBreaker_default(+As,+Sit,?Action) :-
% Action is the chosen action in situation Sit
% from the actions in the choice-set As
%
% Case 1: As is empty
% No action is bound
%
tieBreaker_default([],_Sit,_).

% Case 2: As is non-empty
% The first action in As is the chosen action
%
tieBreaker_default([Action|_],_Sit,Action).
```

11.2.5 Generic Implementations of Secondary Predicates

Predicate: `feasibleActions/3`
Reference: 7.3.2
Code:

```
%%% feasibleActions(+Sit,+W,?FAs) :-
% FAs is a subset of the system's action set,
% representing the set of actions accessible
% (feasible) for agent W in situation Sit
%
% PRECONDITION:
% Sit is a well-formed dnrDALMAS situation
% and W is an agent in the agent set
%
feasibleActions(Sit,W,FAs) :-
       findall(FA,feasibleAction(Sit,W,FA),FAs).
```

Predicate: `permissibleAction/4`
Reference: 7.3.2
Code:

```
%%% permissibleAction(+As,+Sit,+W,?PA) :-
% PA is a permissible action for agent W
% in situation Sit, where W is a member of As
%
% PRECONDITION:
% As is a subset of the set of feasible actions
% and Sit is a well-formed dnrDALMAS situation
% and W is an agent in the agent set
%
permissibleAction(As,Sit,W,Action) :-
       state(Sit,S),
       dDALMAS(S,DDALMAS),
       permissibleActionPredicate(DDALMAS,Module:FnPA/N),
       functor(FnPAGoal,FnPA,N),
       arg(1,FnPAGoal,As),
       arg(2,FnPAGoal,Sit),
       arg(3,FnPAGoal,W),
       member(Action,As),
       arg(4,FnPAGoal,Action),
       call(Module:FnPAGoal).
```

Predicate: `deonticStructure/4`
Reference: 7.3.2
Code:

```
%%% deonticStructure(+FAs,+Sit,+W,?PAs) :-
% PAs is the set of actions in FAs that are permissible
% for agent W in situation Sit
%
% PRECONDITION:
% As is a subset of the set of feasible actions
% and Sit is a well-formed dnrDALMAS situation
% and W is an agent in the agent set
```

```
%
deonticStructure(As,Sit,W,DS) :-
       findall(PA,permissibleAction(As,Sit,W,PA),DS).
```

Predicate: `preferenceStructure/4`
Reference: 7.3.2
Code:

```
%%% preferenceStructure(+As,+Sit,+W,?PS) :-
% PS is a preference ordering of the actions in As
% according to the system's utility function
% for agent W in situation Sit
%
% PRECONDITION:
% As is a subset of the set of feasible actions
% and Sit is a well-formed dnrDALMAS state
% and W is an agent in the agent set
% and the utility function finds actions in preference order
%
preferenceStructure(As,Sit,W,PS) :-
       setof(p(Value,Action),
         utilityFunction(As,Sit,W,p(Value,Action)),PS).
```

Predicate: `choiceSet/4`
Reference: 7.3.2
Code:

```
%%% choiceSet(+As,+Sit,+W,?ChoiceSet) :-
% ChoiceSet is the choice-set of actions
% from a set of feasible actions As for agent W
% in situation Sit, according to the FnCS/N predicate
%
% PRECONDITION:
% As is a subset of the set of feasible actions
% for agent W in situation Sit
% and Sit is a well-formed dnrDALMAS situation
% and W is an agent in the agent set
% and the choice-set predicate finds permissible actions
% in preference order
%
choiceSet(As,Sit,W,CS) :-
       state(Sit,S),
       dDALMAS(S,DDALMAS),
       choiceSetPredicate(DDALMAS,Module:FnCS/N),
       functor(FnCSGoal,FnCS,N),
       arg(1,FnCSGoal,As),
       arg(2,FnCSGoal,Sit),
       arg(3,FnCSGoal,W),
       arg(4,FnCSGoal,CS),
       call(Module:FnCSGoal).
```

11.2.6 Default Implementations of Secondary Predicates

Predicate: `permissibleAction_default/4`
Reference: 7.3.2
Code:

```
%%% permissibleAction_default(+FAs,+Sit,+W,?Action) :-
% Action is a member of FAs (a set of feasible actions)
% and is permissible for agent W in situation Sit
%
% Default: Simple deterministic norm-regulated DALMAS
% where permissible is defined with respect to a norm system
%
permissibleAction_default(FAs,Sit,W,Action) :-
       member(Action,FAs),
       not(prohibited(Action,W,Sit)).
```

Predicate: `choiceSet_default/4`
Reference: 7.3.2
Code:

```
%%% choiceSet_default(+FAs,+Sit,+W,?CS) :-
% CS is the deontic structure (a subset of FAs)
% ordered by preference for agent W in situation Sit
%
% Default: Simple deterministic norm-regulated DALMAS
% where the choice-set consists of all permissible actions
% (the deontic structure) ordered in preference order
%
choiceSet_default(FAs,Sit,W,CS) :-
       deonticStructure(FAs,Sit,W,DS),
       preferenceStructure(DS,Sit,W,PS),
       cs(PS,CS).
```

Predicate: `cs/2 (auxiliary predicate)`
Code:

```
%%% cs(+Ps,?As) :-
% As is a list of the actions corresponding to
% the p(Value,Action) pairs in the list Ps
%
cs([],[]).

cs([p(_V,Action)|Ps],[Action|As]) :-
       cs(Ps,As).
```

Predicate: `nextSituation/4`
Reference: 7.3.2
Code:

```
%%% nextSituation(+Sit,?NextSit,?W,?Action) :-
% NextSit is the next situation that follows
% when agent W performs Action in situation Sit
%
% This predicate corresponds to function f(Wm,S)
% with additional information of moving agent and chosen action
```

```
%
nextSituation(Sit,NextSit,W,A) :-
       atMove(Sit,W),
       feasibleActions(Sit,W,FAs),
       choiceSet(FAs,Sit,W,ChoiceSet),
       tieBreaker(ChoiceSet,Sit,Action),
       !,
       state(Sit,S),
       actionResult(Action,W,S,SPlus),
       stateUpdate(Sit,SPlus,Sit2),
       turnOperator(Sit,WNext),
       atMoveUpdate(Sit2,WNext,NextSit),
       Action = A.
```

Predicate: `run/3`
Reference: 7.3.2
Code:

```
%%% run(+InitialSit,+K,?Sequence) :-
% Sequence is the sequence of situations after K steps
% in the run of a dnrDALMAS with initial situation InitialSit
%
run(InitialSit,N,Sequence) :-
       partialRun(InitialSit,0,N,[InitialSit],Sequence).
```

Predicate: `partialRun/5`
Reference: 7.3.2
Code:

```
%%% partialRun(+Sit,+K,+N,+PartialSequence,?Sequence) :-
% Sequence is the sequence of situations after N steps
% of the run of a dnrDALMAS with current situation Sit
% where PartialSequence is the sequence of situations
% after K steps
%
partialRun(FinalSit,N,N,Sequence,[FinalSit|Sequence]).

partialRun(Sit,K,N,PartialSequence,Sequence) :-
       K < N,
       NextK is K+1,
       nextSituation(Sit,NextSit,W,Action),
       partialRun(NextSit,NextK,N,
          [[W,Action]|PartialSequence],Sequence).
```

11.2.7 Implementation of Cis Predicates

Predicate: `emptyGroundSet/1`
Code:

```
%%% emptyGroundSet(?GS) :-
% GS is an empty set of grounds (empty association list)
%
emptyGroundSet(GS) :-
       empty_assoc(GS).
```

Predicate: addGroundToGS/6
Reference: 7.4.1
Code:

```
%%% addGroundToGS(+CID,Mod:C,+PA,+AA,+NS,?NewNS) :-
% NewNS is the result of adding the well-formed
% condition C (defined in module Mod)
% with unique identifier CID,
% "parameter arity" PA and "agent arity" AA
% to the set of grounds of norm-system NS
%
addGroundToGS(CID,Mod:C,PA,AA,NS,NewNS) :-
       groundSet(NS,GS),
       put_assoc(CID,GS,Mod:C/PA/AA,NewGS),
       updateGroundSet(NS,NewGS,NewNS).
```

Predicate: ground/5
Reference: 7.4.1
Code:

```
%%% ground(+Cond,+GroundSet,?Mod:CF,?Params,?AA) :-
% Cond is a condition in GroundSet
% defined in module Mod with functor CF,
% parameters Params and "agent arity" AA
%
ground(Cond,GroundSet,Mod:CF,Params,AA) :-
       Cond =.. [CID|Params],
       length(Params,PA), %%% Length of Params matches PA?
       gen_assoc(CID,GroundSet,Mod:CF/PA/AA).
```

Predicate: groundSatisified/6
Reference: 7.5
Code:

```
%%% groundSatisfied(+G,+GS,+WArgs,+OtherArgs,+Sit,?WArgs2) :-
% G is a sit-condition in the ground-set GS
% (or conjuncion or disjunction of sit-conditions)
% and G is satisfied in situation Sit,
% where WArgs is a list of previously bound agent arguments
% and WArgs2 is a list of bound agent arguments
% (possibly extended, depending on the arity
% of the sit-conditions in G)

% Base case 1: Special case for the sit-condition true/0
%
groundSatisfied(move*true,_GroundSet,WArgs,[W|_],Sit,WArgs2) :-
       atMove(Sit,W),
       write(' true AND atMove('), write(W), write(')'),
       !,
       WArgs2 = WArgs.
```

```
% Base case 2: Ground consists of a single sit-condition
%
groundSatisfied(Op*CID,
GroundSet,WArgs,OtherArgs,Sit,WArgs2) :-
       ground(CID,GroundSet,Module:C,Params,AA),
          %%% Apply operator Op
          %%% and bind agent arguments if necessary
       applyOperator(Op,Module:C,
       Params,AA,WArgs,OtherArgs,Sit,WArgs2,OpC),
          %%% Is sit-cond OpC satisfied?
       call(OpC).

% Base case 3: Negated ground
%
groundSatisfied(not(G),
GroundSet,WArgs,OtherArgs,Sit,WArgs2) :-
       not(groundSatisfied(G,
       GroundSet,WArgs,OtherArgs,Sit,WArgs2)).

% Recursive case 1: Ground is a conjunction of simpler grounds
%
groundSatisfied(and(G1,G2),
GroundSet,WArgs,OtherArgs,Sit,WArgs2) :-
       groundSatisfied(G1,GroundSet,WArgs,OtherArgs,Sit,WA2),
       groundSatisfied(G2,GroundSet,WA2,OtherArgs,Sit,WArgs2).

% Recursive case 2: Ground is a disjunction of simpler grounds
%
groundSatisfied(or(G1,G2),
GroundSet,WArgs,OtherArgs,Sit,WArgs2) :-
       (
       groundSatisfied(G1,GroundSet,WArgs,OtherArgs,Sit,WArgs2);
       groundSatisfied(G2,GroundSet,WArgs,OtherArgs,Sit,WArgs2)
       ).
```

Predicate: `emptyConsequenceSet/1`
Code:

```
%%% emptyConsequenceSet(?CS) :-
% CS is an empty set of consequences (empty association list)
%
emptyConsequenceSet(CS) :-
       empty_assoc(CS).
```

Predicate: `addConsequenceToCS/6`
Reference: 7.4.1
Code:

```
%%% addConsequenceToCS(+CID,+Mod:C,+AA,+NS,?NewNS) :-
% NewNS is the result of adding the well-formed
% condition C (defined in module Mod)
% with unique identifier CID,
% "parameter arity" PA and "agent arity" AA
% to the set of consequences of norm-system NS
%
addConsequenceToCS(CID,Mod:C,PA,AA,NS,NewNS) :-
       consequenceSet(NS,CS),
       put_assoc(CID,CS,Mod:C/PA/AA,NewCS),
       updateConsequenceSet(NS,NewCS,NewNS).
```

Predicate: `consequence/5`
Reference: 7.4.1
Code:

```
%%% consequence(+C,+ConsequenceSet,?Mod:CF,?Params,?AA) :-
% C is a condition in ConsequenceSet
% defined in module Mod with functor CF,
% parameters Params and "agent arity" AA
%
consequence(Cond,ConsequenceSet,Mod:CF,Params,AA) :-
       Cond =.. [CID|Params],
       length(Params,PA), %%% Length of Params matches PA?
       gen_assoc(CID,ConsequenceSet,Mod:CF/PA/AA).
```

Predicate: `consequenceSatisfied/6`
Reference: 7.5
Code:

```
%%% consequenceSatisfied(+C,+CS,+WAs,+OtherArgs,+Sit,?WAs2) :-
% C is a sit-condition in the consequence-set CS
% (or conjuncion or disjunction of sit-conditions)
% and C is satisfied in Sit, where WAs is a list
% of previously bound agent arguments
% and WAs2 is a list of bound agent arguments
% (possibly extended, depending on the arity
% of the sit-conditions in C)
%
% or(TiD,TjD): For both TiD and TjD
% check if the corresponding EiD and EiD are satisfied.
%
% and(TiD,TjD): Check if any of the corresponding
% EiD or EjD are satisfied.

% Base case: Elementary consequence TiD; check if the
% corresponding EiD is satisfied.
%
consequenceSatisfied(Op*Cond,
ConsequenceSet,WArgs,OtherArgs,Sit,WArgs2) :-
       consequence(Cond,ConsequenceSet,Module:D,DArgs,AA),
          %%% Apply operator Op
          %%% and bind agent arguments if necessary
       applyOperator(Op,Module:D,
       DArgs,AA,WArgs,OtherArgs,Sit,WArgs2,OpD),
          %%% Is sit-cond OpC satisfied?
       call(OpD).

% Base case 2: Negated consequence
```

```
%
consequenceSatisfied(not(D),
ConsequenceSet,WArgs,OtherArgs,WArgs2) :-
        not(consequenceSatisfied(D,
        ConsequenceSet,WArgs,OtherArgs,WArgs2)).

consequenceSatisfied(or(D1,D2),
ConsequenceSet,WArgs,OtherArgs,WArgs2) :-
        consequenceSatisfied(D1,
        ConsequenceSet,WArgs,OtherArgs,WA2),
        consequenceSatisfied(D2,
        ConsequenceSet,WA2,OtherArgs,WArgs2).

consequenceSatisfied(and(D1,D2),
ConsequenceSet,WArgs,OtherArgs,WArgs2) :-
        (
        consequenceSatisfied(D1,
        ConsequenceSet,WArgs,OtherArgs,WArgs2);
        consequenceSatisfied(D2,
        ConsequenceSet,WArgs,OtherArgs,WArgs2)
        ).
```

11.2.8 Implementation of m-cis Predicates

Predicate: `move/6`
Reference: 7.2
Code:

```
%%% move/6:
% Applies operator M/Mi to condition C defined in Module
%
move([],Module:C,CArgs,WArgs,OtherArgs,Sit,
     m(Module:C,CArgs,WArgs,OtherArgs,Sit)).

move([N],Module:C,CArgs,WArgs,OtherArgs,Sit,
     m(N,Module:C,CArgs,WArgs,OtherArgs,Sit)).
```

Predicate: `m/5`
Reference: 7.2
Code:

```
%%% m(+Module:C,+CArgs,+WArgs,+[W|...],+Sit) :-
% W is to move in Sit and
% C(S,...) is satisfied in state S of situation Sit
%
m(Module:C,CArgs,WArgs,[W|_],Sit) :-
        atMove(Sit,W), %%% W is at move
        state(Sit,S),
        stateCondition(Module:C,CArgs,WArgs,S,CS),
        call(CS),
        write(' atMove('), write(W), write(') AND '),
        write(C), write(CArgs), write(WArgs).
```

Predicate: `m/6`
Reference: 7.2
Code:

```
%%% m(+N,+Module:C,+CArgs,+WArgs,+[W|...],+Sit) :-
% W is to move in Sit
% and W is bound to the N:th agent argument in WArgs
% and C(S,...) is satisfied in state S of situation Sit
%
m(N,Module:C,CArgs,WArgs,[W|_],Sit) :-
       atMove(Sit,W),
       % W is at move
       nth(N,WArgs,W),
       % W is the N:th agent in the agent argument list
       state(Sit,S),
       stateCondition(Module:C,CArgs,WArgs,S,CS),
       call(CS),
       write(' atMove('), write(W), write(') AND '),
       write(C), write(CArgs), write(WArgs).
```

11.2.9 Implementation of np-cis Predicates

Predicate: `t/7`
Reference: 7.2
Code:

```
%%% t/7:
% Applies operator Ti to condition D defined in Module
%
t([Ti],Module:D,DArgs,WArgs,OtherArgs,Sit,
  e(Ti,Module:D,DArgs,WArgs,OtherArgs,Sit)).
```

Predicate: `e/7 (implementation of E2 - E7)`
Reference: 7.2
Code:

```
%%% e(2,D,DArgs,WArgs,OtherArgs,WArgs2) :-
% D(DArgs) is satisfied in state S, and
% SPlus is the resulting state
% if W performs Action in state S, and
% D is not satisfied in SPlus
%
e(2,Module:D,DArgs,WArgs,[W,Action|_],Sit) :-
       state(Sit,S),
       stateCondition(Module:D,DArgs,WArgs,S,DS),
       call(DS),
       actionResult(Action,W,S,SPlus),
       stateCondition(Module:D,DArgs,WArgs,SPlus,DSPlus),
       not(call(DSPlus)),
       write(' '),write(e(2)*D),write(DArgs),write(WArgs),nl.

e(3,Module:D,DArgs,WArgs,[W,Action|_],Sit) :-
       state(Sit,S),
       stateCondition(Module:D,DArgs,WArgs,S,DS),
        ((
                call(DS),
                actionResult(Action,W,S,SPlus),
                stateCondition(Module:D,DArgs,WArgs,SPlus,DSPlus),
                call(DSPlus)
        );
        (
                not(call(DS)),
                actionResult(Action,W,S,SPlus),
                stateCondition(Module:D,DArgs,WArgs,SPlus,DSPlus),
                not(call(DSPlus))
        )),
        write(' '),write(e(3)*D),write(DArgs),write(WArgs),nl.

e(4,Module:D,DArgs,WArgs,[W,Action|_],Sit) :-
        state(Sit,S),
        stateCondition(Module:D,DArgs,WArgs,S,DS),
        not(call(DS)),
        actionResult(Action,W,S,SPlus),
        stateCondition(Module:D,DArgs,WArgs,SPlus,DSPlus),
        call(DSPlus),
        write(' '),write(e(4)*D),write(DArgs),write(WArgs),nl.

e(5,Module:D,DArgs,WArgs,[W,Action|_],Sit) :-
        state(Sit,S),
        actionResult(Action,W,S,SPlus),
        stateCondition(Module:D,DArgs,WArgs,SPlus,DSPlus),
        not(call(DSPlus)),
        write(' '),write(e(5)*D),write(DArgs),write(WArgs),nl.

e(6,Module:D,DArgs,WArgs,[W,Action|_],Sit) :-
        state(Sit,S),
        stateCondition(Module:D,DArgs,WArgs,S,DS),
        ((
                call(DS),
                actionResult(Action,W,S,SPlus),
                stateCondition(Module:D,DArgs,WArgs,SPlus,DSPlus),
                not(call(DSPlus))
        );
        (
                not(call(DS)),
                actionResult(Action,W,S,SPlus),
                stateCondition(Module:D,DArgs,WArgs,SPlus,DSPlus),
                call(DSPlus)
        )),
        write(' '),write(e(6)*D),write(DArgs),write(WArgs),nl.

e(7,Module:D,DArgs,WArgs,[W,Action|_],Sit) :-
        state(Sit,S),
        actionResult(Action,W,S,SPlus),
        stateCondition(Module:D,DArgs,WArgs,SPlus,DSPlus),
        call(DSPlus),
        write(' '),write(e(7)*D),write(DArgs),write(WArgs),nl.
```

11.2.10 Implementation of Norm-System Predicates

Predicate: `emptyNormSet/1`
Reference: 7.4
Code:

```
%%% emptyNormSet(?J) :-
% J is an empty set of norms (an empty ordered list)
%
emptyNormSet([]).
```

Predicate: `addNorm/5`
Reference: 7.4
Code:

```
%%% addNorm(+ID/N,+Ground,+Consequence,+OldNS,?NewNS) :-
% NewNS is the result of adding norm ID/N with ground Ground
% and consequence Consequence to the norm-system OldNS
%
addNorm(ID/N,Ground,Consequence,NS,NewNS) :-
        normSet(NS,J),
        ord_add_element(J,n(ID/N,Ground,Consequence),NewJ),
        updateNormSet(NS,NewJ,NewNS).
```

Predicate: `removeNorm/5`
Reference: 7.4
Code:

```
%%% removeNorm(+ID/N,?Ground,?Consequence,+NS,?NewNS) :-
% NewNS is the result of deleting norm ID/N
% from the norm-system OldNS
%
removeNorm(ID/N,_Ground,_Consequence,NS,NewNS) :-
        normSet(NS,J),
        ord_del_element(J,n(ID/N,_,_),NewJ),
        updateNormSet(NS,NewJ,NewNS).
```

Predicate: `norm/5`
Reference: 7.4
Code:

```
%%% norm(+NS,?ID/N,?G,?G) :-
% NS is a norm-system (ordered set of norms)
% and ID/N is the name of a norm
% and G is the ground (sit-condition) of the norm
% and C is the consequence of the norm
%
norm(NS,ID/N,G,C) :-
        normSet(NS,J),
        member(n(ID/N,G,C),J).
```

11.3 Appendix 3: Implementation of COLOUR&FORM

The implementation consists of two Prolog files:

1. cfworld.pl
2. cfdalmas.pl

The first file contains a "world engine" that initialises a colour and form system and starts a sequence of actions. The second file carries out step 1-4:

Step 1
Definitions are provided for the following predicates:
1. State-conditions `diff/3` and `eq/3`; see section 7.6.
2. `agentUtilityFunction/4`: For agent *chroma*, this predicate associates utility value *great* to act *change(colour)* and *medium* to act *change(form)*, and vice versa for agent *forma*. Note that the term `great` is "less than" term `medium` according to the Prolog Standard Ordering of terms.
3. `agentResultOfAction/4`: This is the action-result predicate that determines the resulting state when an agent performs an action in a certain state. It uses two auxiliary predicates: `flipColour/2` and `flipForm/2`.

Step 2
`initialDDALMAS/10` is called to create a dDALMAS structure holding the agent set (which is empty from start), the action set (which contains two actions) and the functors and arity of the basic primary and secondary predicates. In this example, the predefined versions of the feasible actions predicate, permissible action predicate, choice-set predicate, turn-operator predicate and tie-breaker predicate are registered.

Step 3
In `initialNormSystem/1`, an empty norm-system is created. State-condition predicate `diff/0` (with two agent arguments) is added to the ground-set, and state-condition predicate `eq/0` (with two agent arguments) is added to the consequence-set. Finally, a single norm is added to the norm-set.

Step 4
An initial state is created in the following way: An empty knowledge base is created, and passed as parameter to `initialState/1` together with the initial norm-system created in step 3 and the `dDALMAS` structure created in step 2. Two agents (with initial colour and form) are then added to the *dnrDALMAS*, modifying the agent list and the current knowledge base.

11.3.1 Code for the COLOUR & FORM system

cfworld.pl:

```
:- use_module(library(lists)).

:- ensure_loaded('cfdalmas.pl').


cfRun(N,Wm,C1,F1,C2,F2,Sequence) :-
       N > 0,
       initialCFSituation(Wm,C1,F1,C2,F2,InitialSit),
       dnrDALMAS:run(InitialSit,N,Sequence).
```

cfdalmas.pl:

```
:- use_module(library(lists),[member/2]).

:- use_module('..\\dnrDALMAS\\dnrDALMAS.pl').

%%% STEP 1: Implement state-conditions and primary predicates

%%% diff(State,+W1,+W2) :-
% W1 and W2 are different agents
%
diff(_,W1,W2) :-
       W1 \== W2.

%%% eq(+State,+W1,+W2) :-
% W1 and W2 have the same state (same colour and form)
%
eq(State,W1,W2) :-
       dnrDALMAS:knowledgeBase(State,KB),
       member(wState(W1,_,C,F),KB),
       member(wState(W2,_,C,F),KB).

%%% agentUtilityFunction(+Action,+Sit,+W,?Value) :-
% Value is the preference value for W performing Action
% in situation Sit
%
% Note: great @< medium (standard ordering of terms)
%
agentUtilityFunction(change(colour),Sit,W,Value) :-
       dnrDALMAS:atMove(Sit,W),
       dnrDALMAS:state(Sit,State),
       dnrDALMAS:knowledgeBase(State,KB),
       member(wState(W,chroma,_,_),KB),
       !,
       Value = great.
```

```
agentUtilityFunction(change(colour),Sit,W,medium) :-
       dnrDALMAS:atMove(Sit,W),
       dnrDALMAS:state(Sit,State),
       dnrDALMAS:knowledgeBase(State,KB),
       member(wState(W,forma,_,_),KB).

agentUtilityFunction(change(form),Sit,W,Value) :-
       dnrDALMAS:atMove(Sit,W),
       dnrDALMAS:state(Sit,State),
       dnrDALMAS:knowledgeBase(State,KB),
       member(wState(W,forma,_,_),KB),
       !,
       Value = great.

agentUtilityFunction(change(form),Sit,W,medium) :-
       dnrDALMAS:atMove(Sit,W),
       dnrDALMAS:state(Sit,State),
       dnrDALMAS:knowledgeBase(State,KB),
       member(wState(W,chroma,_,_),KB).

%%% agentResultOfAction(+change(Attribute),+W,+S,?SPlus) :-
% SPlus is the resulting state when agent W
% performs action change(Attribute) in state S
%
agentResultOfAction(change(colour),W,S,SPlus) :-
       dnrDALMAS:knowledgeBase(S,KB),
       select(wState(W,Type,Colour,Form),KB,Rest),
       flipColour(Colour,OtherColour),
       dnrDALMAS:knowledgeBaseUpdate(S,
      [wState(W,Type,OtherColour,Form)|Rest],SPlus).

agentResultOfAction(change(form),W,S,SPlus) :-
       dnrDALMAS:knowledgeBase(S,KB),
       select(wState(W,Type,Colour,Form),KB,Rest),
       flipForm(Form,OtherForm),
       dnrDALMAS:knowledgeBaseUpdate(S,
      [wState(W,Type,Colour,OtherForm)|Rest],SPlus).

flipColour(black,white).
flipColour(white,black).

flipForm(circle,square).
flipForm(square,circle).

initialCFSituation(Wm,C1,F1,C2,F2,InitialSit) :-
       initialCFState(C1,F1,C2,F2,InitialState),
       dnrDALMAS:initialSituation(Wm,InitialState,InitialSit).
```

```
initialCFState(C1,F1,C2,F2,InitialState) :-
        %%% STEP 2: Register agent set,
        %%% action set and predicates
       dnrDALMAS:initialDDALMAS([],
             [change(colour),change(form)],
             user:agentResultOfAction/4,
             dnrDALMAS:feasibleAction_default/3,
             dnrDALMAS:permissibleAction_default/4,
             user:agentUtilityFunction/4,
             dnrDALMAS:choiceSet_default/4,
             dnrDALMAS:turnOperator_default/2,
             dnrDALMAS:tieBreaker_default/3,DDALMAS),
        %%% STEP 3: Create a norm-system
       initialNormSystem(NormSystem),
        %%% STEP 4: Create an initial state
       dnrDALMAS:initialState([],NormSystem,DDALMAS,S1),
       addAgent(chroma,chroma,C1,F1,S1,S2),
       addAgent(forma,forma,C2,F2,S2,InitialState).


initialNormSystem(NormSystem) :-
       dnrDALMAS:emptyNormSystem(NS),
       dnrDALMAS:addGroundToGS(diff,user:diff,0,2,NS,NS2),
       dnrDALMAS:addConsequenceToCS(eq,user:eq,0,2,NS2,NS3),
       dnrDALMAS:addNorm(colourandform/n1,
                        move(1)*diff,t(7)*eq,NS3,NormSystem).


addAgent(W,Type,Colour,Form,State,NewState) :-
       dnrDALMAS:knowledgeBase(State,KB),
       dnrDALMAS:agentSet(State,Ws),
       dnrDALMAS:knowledgeBaseUpdate(State,
       [wState(W,Type,Colour,Form)|KB],S2),
       dnrDALMAS:agentSetUpdate(S2,[W|Ws],NewState).
```

11.3.2 Example: 4-event Run of COLOUR&FORM

This is an example of a 4-event run of the COLOUR&FORM DALMAS with initial situation $\langle\omega_c, \langle\omega_c,$ *black*, *square*$\rangle, \langle\omega_f,$ *white*, *circle*$\rangle\rangle$. Note that act *change(form)* is prohibited for ω_f in step 2 and 4, since that act would lead to a state where both agents have identical attributes (*white* and *square*). If the ground of a norm is satisfied but not its consequence, the corresponding output is printed in *italics*. If both the ground and the consequence of a norm are satisfied, the corresponding output is printed in **boldface**.

?- **cfRun(4,chroma,black,square,white,circle,Sequence).**

- My set of feasible actions [chroma]: [change(colour),change(form)]
- Trying colourandform/n1 [change(colour)]:
 atMove(chroma) AND diff[][chroma,forma] -->

```
- Trying colourandform/n1 [change(form)]:
 atMove(chroma) AND diff[][chroma,forma] -->
- My deontic structure [chroma]: [change(colour),change(form)]
- My preference structure [chroma]: [p(great,change(colour)),p(medium,change(form))]
- My choice set [chroma]: [change(colour),change(form)]
- I choose action change(colour)

- My set of feasible actions [forma]: [change(colour),change(form)]
- Trying colourandform/n1 [change(colour)]:
 atMove(forma) AND diff[][forma,chroma] -->
- Trying colourandform/n1 [change(form)]:
 atMove(forma) AND diff[][forma,chroma] -->
 e(7)*eq[][forma,chroma]
 --> PROHIBITED(change(form),forma)
- My deontic structure [forma]: [change(colour)]
- My preference structure [forma]: [p(medium,change(colour))]
- My choice set [forma]: [change(colour)]
- I choose action change(colour)

- My set of feasible actions [chroma]: [change(colour),change(form)]
- Trying colourandform/n1 [change(colour)]:
 atMove(chroma) AND diff[][chroma,forma] -->
- Trying colourandform/n1 [change(form)]:
 atMove(chroma) AND diff[][chroma,forma] -->
- My deontic structure [chroma]: [change(colour),change(form)]
- My preference structure [chroma]: [p(great,change(colour)),p(medium,change(form))]
- My choice set [chroma]: [change(colour),change(form)]
- I choose action change(colour)

- My set of feasible actions [forma]: [change(colour),change(form)]
- Trying colourandform/n1 [change(colour)]:
 atMove(forma) AND diff[][forma,chroma] -->
- Trying colourandform/n1 [change(form)]:
 atMove(forma) AND diff[][forma,chroma] -->
 e(7)*eq[][forma,chroma]
 --> PROHIBITED(change(form),forma)
- My deontic structure [forma]: [change(colour)]
- My preference structure [forma]: [p(medium,change(colour))]
- My choice set [forma]: [change(colour)]
- I choose action change(colour)

Sequence =
[dnrDALMAS_sit(chroma,dnrDALMAS_state([wState(forma,forma,white,circle),
wState(chroma,chroma,black,square)],
dnrDALMAS(dDALMAS([forma,chroma],[change(colour),change(form)],
user:agentResultOfAction/4,
dnrDALMAS:feasibleAction_default/3,
dnrDALMAS:permissibleAction_default/4,
user:agentUtilityFunction/4,
dnrDALMAS:choiceSet_default/4,
dnrDALMAS:turnOperator_default/2,
dnrDALMAS:tieBreaker_default/3),
ns(t(diff,user:diff/0/2,0,t,t),t(eq,user:eq/0/2,0,t,t),
[n(colourandform/n1,move(1)*diff,t(7)*eq)])))),
[forma,change(colour)],
[chroma,change(colour)],
[forma,change(colour)],
[chroma,change(colour)],
dnrDALMAS_sit(chroma,...)]
```

`Sequence` is a Prolog list consisting of the `dnrDALMAS_sit/2` descriptor of the initial situation, a sequence of agent-action pairs showing the sequence of actions for this run, and a descriptor of the final situation.

11.4 Appendix 4: Implementation of WASTE-COLLECTORS

The implementation consists of nine Prolog files:

1. wasteland\wasteland.pl
2. wasteland\world1.pl
3. wastedalmas\wastedalmas.pl
4. wastedalmas\wasteagent.pl
5. wastedalmas\lap.pl
6. wastedalmas\sphere.pl
7. wastedalmas\odelbom.pl
8. auxiliary\auxiliary.pl
9. auxiliary\knowledgebase.pl

File (1) initialises a "world engine" for a waste-collector world whose characteristics are defined in (2). Files (3) – (7) carry out step 1 to 4:

Step 1
File (5) provides definitions for the state-condition predicates `diff/3` and `overlap/4`; see section 7.1.1. The code for `overlap/4` is shown in section 11.4.2. It uses a very simple (and not very efficient) implementation of `sphere/1`, which represents the Moore Neighbourhood of a position in a square grid. This predicate is defined in (6). The code in (5) and (6) is adapted to the dnrDALMAS framework from earlier work by Morrone. [12]

File (4) defines the primary predicates `agentUtilityFunction/4` and `agentResultOfAction/4`, which are the implementations of the primary predicates in section 6.6. The code for the utility function predicate and the action result predicate will not be examined in detail, but the code is shown in section 11.4.3. The primary predicates and the state-condition predicates call the auxiliary predicates in (8) and the knowledge base predicates in (9). The knowledge base for a WASTE-COLLECTOR DALMAS is represented as a Prolog list of facts which is queried and updated through calls to `askKB/2` and `tellKB/3`.

Step 2 and 4
File (3) uses `initialWasteCollectorsState/4` to create an initial dnrDALMAS state with a dDALMAS descriptor containing an agent set, an action set and the functors and arity of the system's primary and secondary predicates. The system uses the predefined versions of the feasible actions predicate, permissible action predicate, choice-set predicate, turn-operator predicate and tie-breaker predicate. The agent set is populated by a a call to `aAddAgents/3`, which in turn calls `aAddAgent/3` to add a new agent to the system.

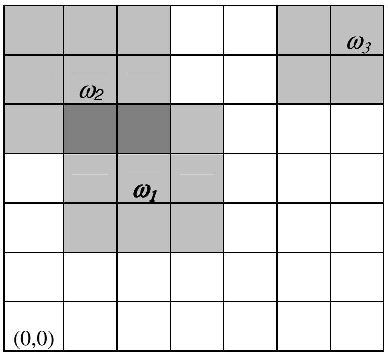
From the initial state an initial situation is created by a call to `initialWasteCollectorsSituation/5`.

Step 3
The norm-system of the system is created by calling `initialNormSystem/1`, which is defined in (7). This predicate creates an empty norm-system, adds `diff/0` and `lap/1` (with two agent arguments and one parameter) to the ground-set and adds `lap/1` to the consequence-set. Finally, the eleven norms presented in section 6.6 are added. The code for `initialNormSystem/1` is shown in section 11.4.4.

11.4.1 Test of Elementary Norms

Situation: ω_1 is at (2,3), ω_2 is at (1,5), ω_3 is at (6,6), and ω_1 is to move, as illustrated in figure 2.

Fig. 2. Situation in Test 1-7



Note that $Lap_9(\omega_x,\omega_x)$, $Lap_2(\omega_1,\omega_2)$, $Lap_2(\omega_2,\omega_1)$, $Lap_0(\omega_1,\omega_3)$, $Lap_0(\omega_3,\omega_1)$, $Lap_0(\omega_2,\omega_3)$ and $Lap_0(\omega_3,\omega_2)$ holds in this situation.

Test 1: T_iLap_0

Norm	**Expected result**	**Result**
$\langle M_1Lap_2, T_1Lap_0\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_2Lap_0\rangle$	All acts permissible	OK

$\langle M_1Lap_2, T_3Lap_0\rangle$	*go(south)* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_4Lap_0\rangle$	*go(south)* prohibited, all other acts permissible	OK
$\langle M_1Lap_2, T_5Lap_0\rangle$	*go(south)* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_6Lap_0\rangle$	*go(south)* prohibited, all other acts permissible	OK
$\langle M_1Lap_2, T_7Lap_0\rangle$	*go(south)* prohibited, all other acts permissible	OK

Test 2: T_iLap_1

Norm	**Expected result**	**Result**
$\langle M_1Lap_2, T_1Lap_1\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_2Lap_1\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_3Lap_1\rangle$	*go(east)* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_4Lap_1\rangle$	*go(east)* prohibited, all other acts permissible	OK
$\langle M_1Lap_2, T_5Lap_1\rangle$	*go(east)* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_6Lap_1\rangle$	*go(east)* prohibited, all other acts permissible	OK
$\langle M_1Lap_2, T_7Lap_1\rangle$	*go(east)* prohibited, all other acts permissible	OK

Test 3: T_iLap_2

Norm	**Expected result**	**Result**
$\langle M_1Lap_2, T_1Lap_2\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_2Lap_2\rangle$	*pass* & *grab* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_3Lap_2\rangle$	*pass* & *grab* prohibited, all other acts permissible	OK
$\langle M_1Lap_2, T_4Lap_2\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_5Lap_2\rangle$	*pass* & *grab* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_6Lap_2\rangle$	*pass* & *grab* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_7Lap_2\rangle$	*pass* & *grab* prohibited, all other acts permissible	OK

Test 4: T_iLap_3

Norm	**Expected result**	**Result**
$\langle M_1Lap_2, T_1Lap_3\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_2Lap_3\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_3Lap_3\rangle$	*go(west)* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_4Lap_3\rangle$	*go(west)* prohibited, all other acts permissible	OK
$\langle M_1Lap_2, T_5Lap_3\rangle$	*go(west)* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_6Lap_3\rangle$	*go(west)* prohibited, all other acts permissible	OK
$\langle M_1Lap_2, T_7Lap_3\rangle$	*go(west)* prohibited, all other acts permissible	OK

Test 5: T_iLap_4

Norm	Expected result	Result
$\langle M_1Lap_2, T_1Lap_4\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_2Lap_4\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_3Lap_4\rangle$	*go(north)* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_4Lap_4\rangle$	*go(north)* prohibited, all other acts permissible	OK
$\langle M_1Lap_2, T_5Lap_4\rangle$	*go(north)* permissible, all other acts prohibited	OK
$\langle M_1Lap_2, T_6Lap_4\rangle$	*go(north)* prohibited, all other acts permissible	OK
$\langle M_1Lap_2, T_7Lap_4\rangle$	*go(north)* prohibited, all other acts permissible	OK

Test 6: T_iLap_6

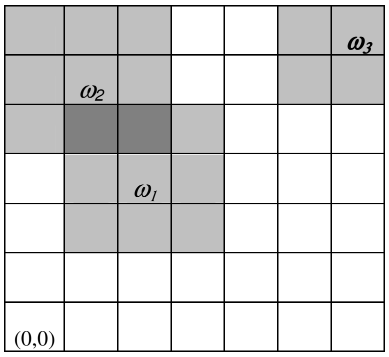
Norm	Expected result	Result
$\langle M_1Lap_2, T_1Lap_6\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_2Lap_6\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_3Lap_6\rangle$	All acts prohibited	OK
$\langle M_1Lap_2, T_4Lap_6\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_5Lap_6\rangle$	All acts prohibited	OK
$\langle M_1Lap_2, T_6Lap_6\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_7Lap_6\rangle$	All acts permissible	OK

Test 7: T_iLap_9

Norm	Expected result	Result
$\langle M_1Lap_2, T_1Lap_9\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_2Lap_9\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_3Lap_9\rangle$	All acts prohibited	OK
$\langle M_1Lap_2, T_4Lap_9\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_5Lap_9\rangle$	All acts prohibited	OK
$\langle M_1Lap_2, T_6Lap_9\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_7Lap_9\rangle$	All acts permissible	OK

In the following test, the situation is that ω_1 is at (2,3), ω_2 is at (1,5), $\omega 3$ is at (6,6), and ω_3 is to move. Since Lap_2 holds neither between ω_3 and ω_1 nor between ω_3 and ω_2, these norms should not give any restrictions on the acts of ω_3.

Fig. 3. Situation in Test 8



Test 8: T_iLap_3

Norm	**Expected result**	**Result**
$\langle M_1Lap_2, T_1Lap_3\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_2Lap_3\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_3Lap_3\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_4Lap_3\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_5Lap_3\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_6Lap_3\rangle$	All acts permissible	OK
$\langle M_1Lap_2, T_7Lap_3\rangle$	All acts permissible	OK

11.4.2 Code for overlap/4

Predicate: `overlap/4`
Reference: 7.7
Code:

```
%%% overlap(+State,?Overlap,+W1,+W2) :-
% W1 overlaps W2 with Overlap squares in state State
%
overlap(S,Overlap,W1,W2) :-
       knowledgeBase(S,KB),
       askKB(aAt(W1,W1Pos),KB),
       askKB(aAt(W2,W2Pos),KB),
       sphere(Sphere),
       agentSphere(W1Pos,Sphere,W1Sphere),
       agentSphere(W2Pos,Sphere,W2Sphere),
       lapSpheres(W1Sphere,W2Sphere,0,Overlap).
```

```
%%% overlap(+State,+(4;6;9),+W1,+W2) :-
% W1 overlaps W2 with 4, 6 or 9 squares in state State
%
overlap(S,(4;6;9),W1,W2) :-
       (overlap(S,4,W1,W2);
        overlap(S,6,W1,W2);
        overlap(S,9,W1,W2)).

%%% overlap(+State,+not469,+W1,+W2) :-
% W1 does not overlaps W2 with 4, 6 or 9 squares in state State
%
overlap(S,not469,W1,W2) :-
       not(overlap(S,4,W1,W2)),
       not(overlap(S,6,W1,W2)),
       not(overlap(S,9,W1,W2)).
```

Predicate: `sphere/1`
Code:

```
%%% sphere(-Sphere) :-
%    Sphere is a list of the Moore neighbourhood of [0,0]
%
sphere(ProtectedSphere) :-
       ProtectedSphere = [[0,0],
       [0,1],[1,1],[1,0],[1,-1],[0,-1],[-1,-1],[-1,0],[-1,1]].
```

Predicate: `agentSphere/3`
Code:

```
%%% agentSphere(+Pos,+Sphere,-AgentSphere) :-
% AgentSphere is the protected sphere around Pos,
% represented as a list of squares,
% where Sphere represents the Moore neighbourhood:
%
%   x x x
%   x O x
%   x x x
%
agentSphere(_,[],[]).
agentSphere(pos(X,Y),[[DX,DY]|Rest1],[pos(XPos,YPos)|Rest2]) :-
       XPos is X+DX,
       YPos is Y+DY,
       agentSphere(pos(X,Y),Rest1,Rest2).
```

Predicate: `lapSpheres/4`
Code:

```
%%% lapSpheres(+W1Sphere,+W2Sphere,+Count,?Overlap) :-
%    W1Sphere and W2Sphere overlap with Overlap squares,
%    where Count is an accumulating parameter
%
lapSpheres([],_,C,C).
lapSpheres([W1Pos|Rest],W2Sphere,Count,Count2) :-
       member(W1Pos,W2Sphere),
       C2 is Count + 1,
       lapSpheres(Rest,W2Sphere,C2,Count2).

lapSpheres([W1Pos|Rest],W2Sphere,Count,Count2) :-
       non_member(W1Pos,W2Sphere),
       lapSpheres(Rest,W2Sphere,Count,Count2).
```

11.4.3 Implementation of Primary Predicates

Predicate: `agentUtilityFunction/4`
Code:

```
agentUtilityFunction(grab,Situation,W,Value) :-
       dnrDALMAS:state(Situation,S),
       dnrDALMAS:knowledgeBase(S,KB),
       askKB(aAt(W,Pos),KB),
       askKB(aAt(waste,Pos),KB),
       !,
       Value = 0.

agentUtilityFunction(grab,Situation,W,Value) :-
       dnrDALMAS:state(Situation,S),
       dnrDALMAS:knowledgeBase(S,KB),
       askKB(aAt(W,Pos),KB),
       not(askKB(aAt(waste,Pos),KB)),
       !,
       Value = 40.

agentUtilityFunction(go(Dir),Situation,W,Value) :-
       dnrDALMAS:state(Situation,S),
       dnrDALMAS:knowledgeBase(S,KB),
       askKB(aAt(W,Pos),KB),
       adjacentLocation(Pos,Dir,APos),
       aGoodPosition(APos,KB),
       !,
       write('- There is a good position to the ')
       write(Dir), nl,
       Value = 10.

agentUtilityFunction(go(Dir),Situation,W,Value) :-
       dnrDALMAS:state(Situation,S),
       dnrDALMAS:knowledgeBase(S,KB),
       askKB(aAt(W,Pos),KB),
       adjacentLocation(Pos,Dir,APos),
       askKB(aVisited(APos),KB),
       not(askKB(aAt(wall,APos),KB)),
       adjacentLocation(APos,_,P2),
       aGoodPosition(P2,KB),
       !,
       write('- There is a good position not far from me'), nl,
       Value = 20.

agentUtilityFunction(go(_Dir),_Situation,_W,Value) :-
       !,
        write('- There are no good positions near me'), nl,
        Value = 40.

agentUtilityFunction(pass,_,_,30).
```

Predicate: `agentResultOfAction/4`
Code:

```
agentResultOfAction(pass,W,S,SPlus) :-
        dnrDALMAS:knowledgeBase(S,KB),
        tellKB(del(aPreviousAction(W,_)),KB,KB2),
        tellKB(add(aPreviousAction(W,pass)),KB2,NewKB),
        dnrDALMAS:knowledgeBaseUpdate(S,NewKB,SPlus).

agentResultOfAction(go(Dir),W,S,SPlus) :-
        dnrDALMAS:knowledgeBase(S,KB),
        askKB(aAt(W,Pos),KB),
        adjacentLocation(Pos,Dir,NextPos),
        Changes = [
                del(aPreviousAction(W,_)),
                add(aPreviousAction(W,go(Dir))),
                del(aAt(W,Pos)),
                add(aAt(W,NextPos))],
        tellKB(Changes,KB,NewKB),
        dnrDALMAS:knowledgeBaseUpdate(S,NewKB,SPlus).

agentResultOfAction(grab,W,S,SPlus) :-
        dnrDALMAS:knowledgeBase(S,KB),
        askKB(aAt(W,Pos),KB),
        not(askKB(aAt(waste,Pos),KB)),
        tellKB(del(aPreviousAction(W,_)),KB,KB2),
        tellKB(add(aPreviousAction(W,grab)),KB2,NewKB),
        dnrDALMAS:knowledgeBaseUpdate(S,NewKB,SPlus).

agentResultOfAction(grab,W,S,SPlus) :-
        dnrDALMAS:knowledgeBase(S,KB),
        askKB(aAt(waste,Pos),KB),
        askKB(aAt(W,Pos),KB),
        askKB(aHolding(W,waste(N)),KB),
        N2 is N+1,
        Changes = [del(aPreviousAction(W,_)),
                   del(aAt(waste,Pos)),
                   del(aHolding(W,waste(N))),
                   add(aHolding(W,waste(N2))),
                   add(aPreviousAction(W,grab))],
        tellKB(Changes,KB,NewKB),
        dnrDALMAS:knowledgeBaseUpdate(S,NewKB,SPlus).
```

11.4.4 Implementation of a Norm-System for WASTE-COLLECTORS

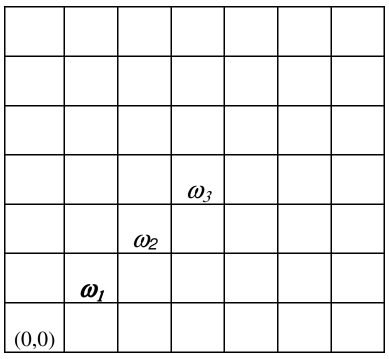
Predicate: `initialNormSystem/1`
Reference: 5.4.5
Code:

```
initialNormSystem(NormSystem) :-
    emptyNormSystem(NS),
    addGroundToGS(true,dnrDALMAS:true,0,0,NS,NS1),
    addGroundToGS(diff,diff,0,2,NS1,NS2),
    addGroundToGS(lap,overlap,1,2,NS2,NS3),
    addConsequenceToCS(lap,overlap,1,2,NS3,NS4),
    addNorm(odelbom/n1a, move(1)*lap(0),
      or(t(4)*lap(2),or(t(6)*lap(2),or(t(7)*lap(2)))),NS4,NS5),
    addNorm(odelbom/n1b, move(1)*lap(0),
      or(t(4)*lap(3),or(t(6)*lap(3),t(7)*lap(3))),NS5,NS6),
    addNorm(odelbom/n2a, move(1)*lap(0),
      or(t(1)*lap(0),or(t(2)*lap(0),or(t(3)*lap(0),
                                  t(5)*lap(0)))),NS6,NS7),
    addNorm(odelbom/n2b, move(1)*lap(0),
      or(t(1)*lap(1),or(t(2)*lap(1),or(t(3)*lap(1),
                                  t(5)*lap(1)))),NS7,NS8),
    addNorm(odelbom/n3a, move(1)*lap(1),
      or(t(1)*lap(J),or(t(2)*lap(J),or(t(3)*lap(J),
                                  t(5)*lap(J)))),NS8,NS9),
    addNorm(odelbom/n3b, move(1)*lap(2),
      or(t(1)*lap(J),or(t(2)*lap(J),or(t(3)*lap(J),
                                  t(5)*lap(J)))),NS9,NS10),
    addNorm(odelbom/n4, move(1)*lap(not469),
      t(7)*lap(6),NS10,NS11),
    addNorm(odelbom/n5, move(1)*lap(4),
      t(7)*lap(3),NS11,NS12),
    addNorm(odelbom/n6, move(1)*lap(6),
      t(5)*lap((4;6;9)),NS12,NS13),
    addNorm(odelbom/n7, move(1)*diff,
      t(7)*lap(9),NS13,NS14),
    addNorm(odelbom/n8, move*true,
      or(t(1)*lap(0),or(t(2)*lap(0),or(t(3)*lap(0),
         t(5)*lap(0)))),NS14,NormSystem).
```

11.4.5 Example: 2-event Run of WASTE-COLLECTORS

Figure 4 illustrates an initial situation for a 3-agent WASTE-COLLECTOR system. An example of a 2-event run (with some output removed to facilitate reading) of this system is shown below.

Fig. 4. Initial Situation



If the ground of a norm is satisfied but not its consequence, the corresponding output is printed in *italics*. If both the ground and the consequence of a norm are satisfied, the corresponding output is printed in **boldface**. Note that two acts are prohibited for the moving agent (ω_2) in the last step of the run.

```
- w3 is at pos(3,3)
- w2 is at pos(2,2)
- w1 is at pos(1,1)
- I have visited pos(1,1)
- My set of feasible actions [w1]: [pass,grab,go(east),go(north),go(west),go(south)]

- Trying odelbom/n1a [pass]:
- Trying odelbom/n1b [pass]:
- Trying odelbom/n2a [pass]:
- Trying odelbom/n2b [pass]:
- Trying odelbom/n3a [pass]:
 atMove(w1) AND overlap[1][w1,w3] -->
- Trying odelbom/n3b [pass]:
- Trying odelbom/n4 [pass]:
 atMove(w1) AND overlap[not469][w1,w3] -->
- Trying odelbom/n5 [pass]:
 atMove(w1) AND overlap[4][w1,w2] -->
- Trying odelbom/n6 [pass]:
- Trying odelbom/n7 [pass]:
 atMove(w1) AND diff[][w1,w2] -->
 atMove(w1) AND diff[][w1,w3] -->
- Trying odelbom/n8 [pass]:
 true AND atMove(w1) -->
- Trying odelbom/n1a [grab]:
- Trying odelbom/n1b [grab]:
- Trying odelbom/n2a [grab]:
- Trying odelbom/n2b [grab]:
- Trying odelbom/n3a [grab]:
 atMove(w1) AND overlap[1][w1,w3] -->
- Trying odelbom/n3b [grab]:
- Trying odelbom/n4 [grab]:
 atMove(w1) AND overlap[not469][w1,w3] -->
- Trying odelbom/n5 [grab]:
 atMove(w1) AND overlap[4][w1,w2] -->
- Trying odelbom/n6 [grab]:
- Trying odelbom/n7 [grab]:
 atMove(w1) AND diff[][w1,w2] -->
 atMove(w1) AND diff[][w1,w3] -->
- Trying odelbom/n8 [grab]:
 true AND atMove(w1) -->
- Trying odelbom/n1a [go(east)]:
- Trying odelbom/n1b [go(east)]:
- Trying odelbom/n2a [go(east)]:
- Trying odelbom/n2b [go(east)]:
- Trying odelbom/n3a [go(east)]:
 atMove(w1) AND overlap[1][w1,w3] -->
```

```
- Trying odelbom/n3b [go(east)]:
- Trying odelbom/n4 [go(east)]:
 atMove(w1) AND overlap[not469][w1,w3] -->
- Trying odelbom/n5 [go(east)]:
 atMove(w1) AND overlap[4][w1,w2] -->
- Trying odelbom/n6 [go(east)]:
- Trying odelbom/n7 [go(east)]:
 atMove(w1) AND diff[][w1,w2] -->
 atMove(w1) AND diff[][w1,w3] -->
- Trying odelbom/n8 [go(east)]:
 true AND atMove(w1) -->
- Trying odelbom/n1a [go(north)]:
- Trying odelbom/n1b [go(north)]:
- Trying odelbom/n2a [go(north)]:
- Trying odelbom/n2b [go(north)]:
- Trying odelbom/n3a [go(north)]:
 atMove(w1) AND overlap[1][w1,w3] -->
- Trying odelbom/n3b [go(north)]:
- Trying odelbom/n4 [go(north)]:
 atMove(w1) AND overlap[not469][w1,w3] -->
- Trying odelbom/n5 [go(north)]:
 atMove(w1) AND overlap[4][w1,w2] -->
- Trying odelbom/n6 [go(north)]:
- Trying odelbom/n7 [go(north)]:
 atMove(w1) AND diff[][w1,w2] -->
 atMove(w1) AND diff[][w1,w3] -->
- Trying odelbom/n8 [go(north)]:
 true AND atMove(w1) -->
- Trying odelbom/n1a [go(west)]:
- Trying odelbom/n1b [go(west)]:
- Trying odelbom/n2a [go(west)]:
- Trying odelbom/n2b [go(west)]:
- Trying odelbom/n3a [go(west)]:
 atMove(w1) AND overlap[1][w1,w3] -->
- Trying odelbom/n3b [go(west)]:
- Trying odelbom/n4 [go(west)]:
 atMove(w1) AND overlap[not469][w1,w3] -->
- Trying odelbom/n5 [go(west)]:
 atMove(w1) AND overlap[4][w1,w2] -->
- Trying odelbom/n6 [go(west)]:
- Trying odelbom/n7 [go(west)]:
 atMove(w1) AND diff[][w1,w2] -->
 atMove(w1) AND diff[][w1,w3] -->
- Trying odelbom/n8 [go(west)]:
 true AND atMove(w1) -->
- Trying odelbom/n1a [go(south)]:
- Trying odelbom/n1b [go(south)]:
- Trying odelbom/n2a [go(south)]:
- Trying odelbom/n2b [go(south)]:
- Trying odelbom/n3a [go(south)]:
 atMove(w1) AND overlap[1][w1,w3] -->
- Trying odelbom/n3b [go(south)]:
```

```
- Trying odelbom/n4 [go(south)]:
 atMove(w1) AND overlap[not469][w1,w3] -->
- Trying odelbom/n5 [go(south)]:
 atMove(w1) AND overlap[4][w1,w2] -->
- Trying odelbom/n6 [go(south)]:
- Trying odelbom/n7 [go(south)]:
 atMove(w1) AND diff[][w1,w2] -->
 atMove(w1) AND diff[][w1,w3] -->
- Trying odelbom/n8 [go(south)]:
 true AND atMove(w1) -->

- My deontic structure [w1]: [pass,grab,go(east),go(north),go(west),go(south)]
- There is a good position to the east
- There is a good position to the north
- There is a good position to the west
- There is a good position to the south

- My preference structure [w1]:
[p(10,go(east)),p(10,go(north)),p(10,go(south)),p(10,go(west)),p(30,pass),p(40,grab)]

- My choice set [w1]: [go(east),go(north),go(south),go(west),pass,grab]
- I choose action go(east)

*** Action[w1]: go(east) to pos(2,1)

- w3 is at pos(3,3)
- w2 is at pos(2,2)
- w1 is at pos(2,1)
- I have visited pos(2,2)
- My set of feasible actions [w2]: [pass,grab,go(east),go(north),go(west),go(south)]

- Trying odelbom/n1a [pass]:
- Trying odelbom/n1b [pass]:
- Trying odelbom/n2a [pass]:
- Trying odelbom/n2b [pass]:
- Trying odelbom/n3a [pass]:
- Trying odelbom/n3b [pass]:
- Trying odelbom/n4 [pass]:
- Trying odelbom/n5 [pass]:
 atMove(w2) AND overlap[4][w2,w3] -->
- Trying odelbom/n6 [pass]:
 atMove(w2) AND overlap[6][w2,w1] -->
- Trying odelbom/n7 [pass]:
 atMove(w2) AND diff[][w2,w1] -->
 atMove(w2) AND diff[][w2,w3] -->
- Trying odelbom/n8 [pass]:
 true AND atMove(w2) -->
- Trying odelbom/n1a [grab]:
- Trying odelbom/n1b [grab]:
- Trying odelbom/n2a [grab]:
- Trying odelbom/n2b [grab]:
- Trying odelbom/n3a [grab]:
```

```
- Trying odelbom/n3b [grab]:
- Trying odelbom/n4 [grab]:
- Trying odelbom/n5 [grab]:
 atMove(w2) AND overlap[4][w2,w3] -->
- Trying odelbom/n6 [grab]:
 atMove(w2) AND overlap[6][w2,w1] -->
- Trying odelbom/n7 [grab]:
 atMove(w2) AND diff[][w2,w1] -->
 atMove(w2) AND diff[][w2,w3] -->
- Trying odelbom/n8 [grab]:
 true AND atMove(w2) -->
- Trying odelbom/n1a [go(east)]:
- Trying odelbom/n1b [go(east)]:
- Trying odelbom/n2a [go(east)]:
- Trying odelbom/n2b [go(east)]:
- Trying odelbom/n3a [go(east)]:
- Trying odelbom/n3b [go(east)]:
- Trying odelbom/n4 [go(east)]:
- Trying odelbom/n5 [go(east)]:
 atMove(w2) AND overlap[4][w2,w3] -->
- Trying odelbom/n6 [go(east)]:
 atMove(w2) AND overlap[6][w2,w1] -->
- Trying odelbom/n7 [go(east)]:
 atMove(w2) AND diff[][w2,w1] -->
 atMove(w2) AND diff[][w2,w3] -->
- Trying odelbom/n8 [go(east)]:
 true AND atMove(w2) -->
- Trying odelbom/n1a [go(north)]:
- Trying odelbom/n1b [go(north)]:
- Trying odelbom/n2a [go(north)]:
- Trying odelbom/n2b [go(north)]:
- Trying odelbom/n3a [go(north)]:
- Trying odelbom/n3b [go(north)]:
- Trying odelbom/n4 [go(north)]:
- Trying odelbom/n5 [go(north)]:
 atMove(w2) AND overlap[4][w2,w3] -->
- Trying odelbom/n6 [go(north)]:
 atMove(w2) AND overlap[6][w2,w1] -->
 e(5)*overlap[(4;6;9)][w2,w1]
 --> PROHIBITED(go(north),w2)
- Trying odelbom/n1a [go(west)]:
- Trying odelbom/n1b [go(west)]:
- Trying odelbom/n2a [go(west)]:
- Trying odelbom/n2b [go(west)]:
- Trying odelbom/n3a [go(west)]:
- Trying odelbom/n3b [go(west)]:
- Trying odelbom/n4 [go(west)]:
- Trying odelbom/n5 [go(west)]:
 atMove(w2) AND overlap[4][w2,w3] -->
- Trying odelbom/n6 [go(west)]:
 atMove(w2) AND overlap[6][w2,w1] -->
- Trying odelbom/n7 [go(west)]:
 atMove(w2) AND diff[][w2,w1] -->
 atMove(w2) AND diff[][w2,w3] -->
- Trying odelbom/n8 [go(west)]:
 true AND atMove(w2) -->
- Trying odelbom/n1a [go(south)]:
- Trying odelbom/n1b [go(south)]:
- Trying odelbom/n2a [go(south)]:
- Trying odelbom/n2b [go(south)]:
- Trying odelbom/n3a [go(south)]:
- Trying odelbom/n3b [go(south)]:
- Trying odelbom/n4 [go(south)]:
- Trying odelbom/n5 [go(south)]:
 atMove(w2) AND overlap[4][w2,w3] -->
- Trying odelbom/n6 [go(south)]:
 atMove(w2) AND overlap[6][w2,w1] -->
- Trying odelbom/n7 [go(south)]:
 atMove(w2) AND diff[][w2,w1] -->
 e(7)*overlap[9][w2,w1]
 --> PROHIBITED(go(south),w2)

- My deontic structure [w2]: [pass,grab,go(east),go(west)]

- There is a good position to the east
- There is a good position to the west

- My preference structure [w2]: [p(10,go(east)),p(10,go(west)),p(30,pass),p(40,grab)]

- My choice set [w2]: [go(east),go(west),pass,grab]
- I choose action go(east)

*** Action[w2]: go(east) to pos(3,2)
```

Kopior av publikationer kan beställas från:
Forskningssekretariatet
Högskolan i Gävle
801 76 Gävle

mail: registrator@hig.se

Förteckning över samtliga FoU-rapporter finns på:
www.hig.se/forskning/forskning/publicerat.htm

Tidigare FoU-rapporter:

Tryck
Reproavdelningen
Högskolan i Gävle

ISSN 1403-8749

Utgiven av:
Utbildnings- och forskningskansliet
Högskolan i Gävle

Februari 2008

HÖGSKOLAN
I GÄVLE

801 76 Gävle
Besöksadress: Kungsbäcksvägen 47
Telefon: 026-64 85 00 Fax: 026-64 86 86
www.hig.se