A Functional Program for Agents, Actions, and Deontic Specifications*

Adam Zachary Wyner

King's College London
London, UK
adam@wyner.info

Abstract. We outline elements of the *Abstract Contract Calculator*, a prototype language implemented in Haskell (a declarative programming language) in which we simulate agents executing abstract actions relative to deontic specifications. The deontic specifications are *prohibition*, *permission*, and *obligation*. The concepts of deontic specifications are derived from Standard Deontic Logic and Dynamic Deontic Logic. The concepts of abstract actions are derived from Dynamic Logic. The logics are declarative, while the implementation is operational. In contrast to other implementations, we have articulated and productive violation and fulfillment markers. Our actions are given with explicit action preconditions and postconditions, and we have deontic specification of complex actions. We implement inference in the *Contrary-to-Duty Obligations* case, which has been a central problem in Deontic Logic. We also distinguish *Contrary-to-Duty Obligations* from obligations on sequences, which has not previously been accounted for in the literature. The language can be used to express a range of alternative notions of actions and deontic specification. We use it to to model and simulate multi-agent systems in which the behavior of an agent is guided by deontic specifications on actions.

1 Introduction

We present an overview of the *Abstract Contract Calculator* (ACC) written in Haskell, which is a functional programming language (cf. Wyner (2006) for the code and documentation for the ACC). The ACC processes the deontic notions of *prohibition*, *permission*, and *obligation* applied to complex, abstract actions. As an intuitive example, suppose *Bill is obligated to leave the room*. We have a *deontic specification* "obligated" applying to an *agentive action* "Bill's leaving the room". We call sets of such expressions *Contract States*. Informally, were Bill to leave the (given) room, he would have violated the obligation to leave the

 This paper was prepared while the author was a postgraduate student at King's College London under the supervision of Tom Maibaum and Andrew Jones, which was funded by a studentship from Hewlett-Packard Research Labs, Bristol, UK. The author thanks Tom, Andrew, and HP Labs for their support and advice. Errors rest with the author.

room. Consequences may follow from the fact that he has violated his obligation. For example, he may then be obligated to pay a fine. The objective of the implementation is to abstractly model deontic specification of agentive actions as well as to simulate the behavior of agents executing actions relative to a contract state.

It is outside the scope of this paper to provide the complete implementation or formalization. Rather, we introduce basic, crucial elements and invite the reader to investigate the language further. In addition, actions and deontic notions have been extensively discussed in the Deontic Logic and Dynamic Logic literature (cf. Lomuscio and Nute (2004) and Wieringa and Meyer (1993), Harel (2000), Meyer (1988), and Khosla and Maibaum (1987)). We indicate some of our key sources. However, in contrast to these declarative works, we focus on operationalizing the notions. Therefore, we do not outline the logics. Indeed, in Wyner (2006), we have identified a range of fundamental questions with some of these logics.

The outline of the paper is as follows. We first discuss the context of our research as well as central issues along with how we address them. We turn to the implementation, which is largely presented conceptually and with fragments of Haskell code. The implementation has two aspects. First, it is a *programming tool* in that it allows alternative notions of deontic specification on agentive actions to be systematically examined and animated. Thus, one can develop and focus on a preferred interpretion of the deontic concepts. Second, having fixed an interpretation, the tool enables one to abstractly simulate environments in which agents behave relative to actions, sets of deontic specifications on actions, and how such sets change. It is intended to be used for simulation and modelling of Multi-Agent systems where deontic specifications govern the behavior of individuals or collectives of agents (see Gilbert and Troitzsch (2005) for a discussion of social science simulations). In the final two sections, we touch on other proposals for implementing deontic notions, and then we mention several aspects of the implementation which were not discussed in this paper as well as mention several aspects left for future research.

2 Background Context

The intial objective of our study was to define a formal language which is suitable for the formation, execution, and monitoring of legal contracts in a multi-agent system. Thus, our approach keeps in mind applied and empirical issues. Among the key issues, we wanted to simulate the behavior of agents with respect to deontic specifications on actions. In addition, we wanted to model how deontic specifications can *change* over time. For instance, if we have a deontic specification such as *Bill is obligated to deliver five pizzas*, we want to be able to determine the conditions under which Bill *violates* or *fulfills* this obligation. In addition, we want to determine what follows in either case. Furthermore, we want to define under what conditions can we eliminate Bill's obligation. In general, how could we operationalize deontic specifications on actions such that they

could be used to *guide* agentive behavior? Thus, the research is an application of deontic reasoning.

For a theoretical underpinning, we focussed on the analyses of deontic notions, particularly those with a dynamic component (cf. Meyer (1988), Khosla and Maibaum (1987), but cf. Carmo and Jones (2001) for a non-dynamic theory). Strictly put, the implementation does not implement a particular deontic logic. We found available logics to be unsuitable for a variety or reasons (cf. Wyner (2006)). Instead, we provide a language in which different logics could be operationalized, though we make some specific suggestions.

We have implemented our system in Haskell, which is a functional programming language. Speaking broadly, functional programming languages implement the *Lambda Calculus*. It is a programming language which is particularly well suited to computational semantics (cf. Doets and van Eijck (2004) and van Eijck (2004). For a comparison to Prolog, see Blackburn and Bos (2005)).

3 Driving Issues

The implementation is driven by four interlocking issues: compositional and productive flags which signal violation or fulfillment of a deontic specification; negation of an action as *antonym* or *opposite*; complex actions, particularly sequences; and the Contrary-to-Duty paradox. In the following, we briefly outline the problems and our solutions, which we find again in the implementation.

3.1 Violability

In our view, the key concept of the deontic notions is that of *violability*. Logical or operational representations ought then to have violation (or fulfillment) markers in the formal language such that one can reason further with them (either for recovery or other processes). Thus, bad behavior is *marked* and *reasoned with* rather than ruled out (cf. Anderson and Moore (1957), Meyer (1988), and Khosla and Maibaum (1987)). We do not adopt the approach of recent proposals which use the deontic notions to *filter out* or to *prioritize* actions (cf. Garcia-Camino et. al. (2005) and Aldewereld et. al. (2005)).

In recent proposals of Standard Deontic Logic (Carmo and Jones (2001)) or Dynamic Deontic Logic (Khosla and Maibaum (1987) and Meyer (1988)), we find a distinguished *proposition* which is used to mark that a deontic specification has been violated. We use the marker for further reasoning or reactive behavior. The richer the structure of the marker, the subtler the ways it can be used (for logical proposals along these lines, cf. van den Meyden (1996, Meyer and Wieringa (1993), and Kent, Maibaum, and Quirk (1993)).

In Wyner (2006), we have argued that the markers for deontic specification on complex actions have to be productively and compositionally derived from the agent, the deontic specification, the input actions, and the mode of combination of the actions. We also argued that in order to calculate the conditions under which an *obligation* is *violated*, we need a lexical semantic function to calculate

action negation. We discuss this in the next section. We should also mention that in Wyner (2006), we argue that *temporal* specifications are *not* essential to deontic specifications on actions. To our knowledge, these claims and supporting arguments are novel. They serve to distinguish our analysis and implementation.

3.2 Action Opposition and Deontic Specification

One key component of our analysis is the calculation of actions in opposition. Suppose *Bill is obligated to deliver pizzas for an hour*. It is intuitively clear that *some* actions count toward fulfilling the obligation and *some other* actions count towards violating the obligation. Furthermore, *not just any activity which is not itself an action of delivering pizzas counts toward a violation*. Indeed, some actions which Bill executes are *deontically underspecified*. If this were not the case, then *anything* Bill does other than delivering pizzas leads to a violation. More formally, set-theoretic complementation is not the appropriate notion for action negation *in our domain of application*, for it would imply that at any one time, the agent can either violate the obligation or fulfill it. There would be no actions which are deontically underspecified. This is unreasonable for agents executing contracts over time. Instead, we need some means to calculate the *opposite actions* with respect to the particular input action, leaving other actions underspecified.

In general, we want to be able to calculate the *relevant* opposite of an action, if there is one. While action opposition in natural language is rather unclear, we use *abstract actions* with respect to which we can define action opposition. Suppose α, β, and γ are abstract actions; we make these clearer in the implementation. For an action, say α, from the domain of actions, we can calculate the opposite action (given well-formedness conditions), say it is β. We can say that γ is not in any relation of opposition to either α or β. Thus, if a complex action is obligatory, we can determine what specific actions fulfill the obligation, what actions violate it, and what actions are deontically underspecified.

Other problems arise where we deontically specify complex actions. For instance, suppose we have the complex action combinator for *sequence*, where one action follows another. $(\alpha;\beta)$ represents α followed by an execution of β. Let us make this sequence obligatory: **Obligated**$(\alpha;\beta)$. Of this obligation, we want to know: What is the mark of violation or fulfillment of this obligation? Under what conditions do the marks appear? Intuitively, the mark of violation ought to indicate that the *sequence* per se has been violated (similarly for fulfillment). Thus, we need some means to define for any well-formed sequence of actions the violation marker for that sequence. Similar points can be made with respect to the other complex action combinators. In addition, we have to consider when the violation marker arises. For example, the sequence is violated where α is first executed, and then β is *not* executed, but not necessarily where β is executed before the execution of α.

In general, we have to be able to productively calculate, for any well-formed complex action, the compositional value of the violation (or fulfillment) marker. In turn, this implies that we have to calculate the *opposite* of any complex

action. Thus, the lexical semantic rules must apply productively; it is not feasible to have a listing of every complex action and its opposite. Productivity and compositionality are also crucial to handle *novel actions*, which are new basic action that we introduce to a particular system. We do not want reasoning and action execution to hang when it is fed novel input.

So far as we know, the importance of productivity, compositionality, or lexical semantic opposition have not been recognized in the deontic logic literature.

3.3 Contrary-to-Duty Obligations

Contrary-to-Duty (CTD) Obligations have been a central problem in Deontic Logic (cf. Carmo and Jones (2001), which claims that it is the defining problem). Thus, an implementation ought to provide for it. CTDs are those cases where a secondary obligation arises in a context where a primary obligation has been violated. In other words, having violated one obligation, one incurs another obligation. For example, if one is obligated to return a book by a specific time, then (given the rules of a particular library), one may be obligated to pay a fine. Such cases are key to legal reasoning and a case of context-dependent reasoning (cf. Carmo and Jones (2001)). We have argued (Wyner (2006)) that violation and fulfillment markers are key to distinguish a CTD case from the case where the primary obligation *changes*. In other words, it is key that the action introduces a violation marker. In virtue of this marker, the secondary obligation is introduced.

3.4 Obligations on Sequences versus Sequences of Obligations

In Wyner (2006), we have argued for a distinction between obligations on sequences and sequences of obligations, contra Meyer (1988) who conflates them (Khosla and Maibaum (1987 mention the distinction, but do not elaborate). For example, a sequence of obligations is: one is obligated to do α and then one is obligated to do β. In contrast, one could be obligated to do α followed by β. The difference is in terms of the *violation* conditions. For a sequence of obligations, each obligatory action introduces its own violation marker. For an obligation on a sequence, failure to execute part of the sequence introduces a violation marker on the sequence *per se*. This highlights the crucial role of productive, compositional markers.

To create these richer markers, we provide a richer structure for complex actions. For example, given a sequence of $\alpha;\beta$, the structure distinguishes the input actions α and β, the resultant action (suppose) γ, and the mode of formation, which is the sequence operator. Given the definitions of basic actions, the complex action operators are given functional definitions. With this, we may define a deontic specification to apply to different *parts* of the complex action relative to the complex action operator. This allows us to define *families* of deontic specifications. For example, we can define three versions of obligations on sequences: in one, the obligation distributes to each component action; in another, the obligation applies to the *collective* action; in another, the obligation

applies so as to allow interruptable obligation specifications. We can map out the logical space of possibilities. This allows us a very fine-grained, more accurate analysis. From these alternatives, we can chose that which best suits the purposes of the implementation. In our domain of application, the latter notion seems most important, and it depends on complex markers which arise in a given order.

4 An Overview of the Implementation

In the following subsections, we present highlights of the modules, necessarily skipping many details. *States of Affairs* are lists of propositions along with indices for worlds and times. *Basic Actions* are essentially functions from States of Affairs to States of Affairs. *Lexical Semantic Functions* allow us to *calculate* actions in specified lexical semantic relations such as *opposite*. These functions help us define the consequences of deontically specified actions. *Deontic Operators* apply to actions to specify what actions lead to States of Affairs in which fulfillment or violation is marked relative to the action and agent. We call such a specification a *Contract Flag State*. We implement reasoning for *Contrary-to-Duty Obligations* by modifying contract states relative to violation or fulfillment *flags*. We end with a presentation of complex actions.

4.1 States of Affairs

We construct many of our expressions from basic Haskell types for strings `String`, integers `Int`, and records, which are labels associated with values of a given type. In terms of these, we have several derived types.

Definition 1.
```
type PropList = [String]
type World    = Int
type Time     = Int
type SOA      = Rec (properties :: PropList, time :: Time,
                     world :: World)
type DBSoas   = [SOA]
```

Our atomic propositions are of type *String* such as *prop1* and *prop2*. Prefixing a string with *neg-* forms the negation of a proposition, and we have a *double negation elimination* rule. Lists of propositions, of type *PropList*, form the properties which define the properties which hold of a state of affairs. We can filter the lists for consistency. This means that we remove from the model any list of properties which has a proposition and its negation such as *[prop1, neg-prop1]*. Filtering serves to *constrain* the logical space of models under consideration and used for processing. For our purposes, we do not have complex propositions other than negation. Nor do we address inference from propositions at the level of contexts.

States-of-Affairs, which are of type *SOA*, are records comprised of a list of properties along with indices for world and time. An example SOA is:

Example 1. (properties = [prop1, prop7, prop5, neg-prop3],
time = 2, world = 4)

Lists of expressions of type *SOA* are of type *DBSoas*. These can be understood as *alternative states of affairs* or *possible worlds*.

4.2 Basic Actions

An *action* is of a record of type *Action*, which has fields for a *label* of type *String*, preconditions *xcond* of type *PropList*, and postconditions *ycond* of type *PropList*. An action is used to express *state transitions* from SOAs where the preconditions hold to SOAs where the postconditions hold. An action with an agent is of type *AgentiveAction*, which is a record with fields for an action and an *Agent* of type *String*. A list of agentive actions is of type *DBAgentiveAction*.

Definition 2.
type Action = Rec (label :: String,
xcond :: PropList,
ycond :: PropList)
type DBAction = [Action]
type Agent = String
type AgentiveAction = Rec (action :: Action,
agent :: Agent)
type DBAgentiveAction = [AgentiveAction]

An example of an agentive action is:

Example 2. (action = (label = Action6,
xcond = [prop1, prop7, prop5],
ycond = [prop3, neg-prop4, neg-prop6]),
agent = Jill)

This represents an *abstract agentive action*, which contrasts with agentive actions found in natural language such as *Jill leaves*. We work exclusively with abstract agentive actions since we can explicitly work with the properties which exhaustively define them. It is harder to do so with natural language expressions since it is not clear that we can either explicitly or exhaustively define them in terms of component properties. Nonetheless, we can refer to the natural language examples where useful.

The function *doAgentiveAction* in **Definition 3** takes expressions of type *SOA* and *AgentiveAction* and outputs an expression of type *SOA*.

Definition 3. *type doAgentiveAction :: SOA → AgentiveAction → SOA*

In the definition of the function (not provided), an action can be executed so long as the preconditions of the action are a subset of the properties of the SOA with respect to which the action is to be executed. Following execution of the action, the postconditions of the action hold in the subsequent context, and the time index of the resultant SOA is incrementally updated (in this paper,

we do not manipulate the world index). Further constraints on the execution of the well-formed transitions are that the properties of the resultant SOA must be *consistent* (no contradictions) and *non-redundant* (no repeat propositions). In addition, we *inertially maintain* any properties of the input *SOA* which are not otherwise changed by the execution of the action.

In (3), we have an example.

Example 3.
input> doAgentiveAction
(properties = [prop1, neg-prop3, prop5, prop7],
time = 2, world = 4)
(action = (label = Action6,
xcond = [prop1, prop5, prop7],
ycond = [prop3, neg-prop4, neg-prop6]),
agent = Jill)
output> (properties = [prop1, prop3, neg-prop4,
prop5, neg-prop6, prop7],
time = 3, world = 4)

4.3 Lexical Semantic Functions

For the purposes of deontic specification on agentive actions, we define lexical semantic functions. These functions allow us to *functionally* (in the mathematical sense) determine actions in specified relationships. This is especially important for the definition of *obligation*, where we want to determine which *specific alternatives* of a given action induce violation. One observation we want to account for is the following. Informally, if it is obligatory for Jill to leave the room, then Jill would violate the obligation by remaining in the room. On the other hand, if it is obligatory for Jill to remain in the room, then Jill would violate the obligation by leaving the room. In other words, we see a *reciprocal* relationship between actions in opposition. Furthermore, notice that if Jill's leaving the room is obligatory, then the action which fulfills the obligation and the action which violates the obligation *must both be executable* in the same *SOA*. This means that the actions have *the same precondition properties*. While the natural language case provides the intuitions behind the functions, we implement them with respect to our abstract actions. We only provide a sample of the lexical semantic functions (see Wyner 2006 for further discussion).

Let us suppose a (partial) lexical semantic function *findOpposites*, which is essentially a function from *Action* to *Action*. For processing, it takes a lexicon and some constraints. For example, suppose *findOpposites* applied to the action labelled *Action6* yields *Action7* and vice versa. While there are many potential implementations of action opposition, we have defined the function *findOpposites* such that it outputs an action which is the same as the input action but for the negation of one of the postcondition propositions. This closely models the natural language example of the opposition between *leave* and *remain*. As an illustration, we have the following:

Example 4. input> findOpposites (label = Action6,
xcond = [prop1, prop7, prop5],
ycond = [prop3, neg-prop4, neg-prop6])
output> (label = Action7, xcond = [prop1, prop7, prop5],
ycond = [prop3, neg-prop4, prop6])

Three things are important about the function *actionOpposites* for our purposes. First, we can calculate *specific alternative actions* which give rise to violations. As discussed earlier, it is unintuitive that just *any action* other than the obligated action should give rise to violation. Second, as a calculation, we can find an opposite for any action *where the lexical structure allows one*. For the purposes of deontic specification, it need *not* be the case that every action *has* an antonym (although one could define a function and lexical space to allow this). Crucially, this holds for atomic as well as complex actions. And finally, the function *actionOpposites* is defined so as to provide reciprocal actions; that is, the opposite of *Action6* is *Action7* and vice versa. Thus, the function closely models the natural language case discussed above.

4.4 Deontic Specifications

The previous three subsections are components of deontic specifications on actions, which we model on the following intuition. Suppose an agent *Jill* is obligated to delivery a pizza. This implies that were she to deliver the pizza, in the context after the delivery of the pizza, we would want to indicate that *Jill* has delivered the pizza. Moreover, by doing so, she has fulfilled her obligation *with respect to her obligation to deliver the pizza*. On the other hand, suppose *Jill* were not to deliver the pizza, which is the opposite of delivering the pizza. In this case, we should indicate in the subsequent context that Jill that has not delivered the pizza. Furthermore, by doing so, she has *violated her obligation with respect to delivering the pizza*. We assume there are *deontically underspecified* actions as well. For example, if *Jill* eats an apple, which she could do concurrently over the course of delivering the pizza or not delivering the pizza, it may be that she does not incur a violation or fulfillment flag relative to that action. While it is possible that we use a *fixed* list for some cases to determine when violation markers arise, this will not work for complex actions or novel actions, which are those actions that are not already prelisted in a lexicon.

To define the deontic specifications, we provide a type *ContractFlag*. This type is a record having fields for: the action which is executed (indicated by the label), the deontic specification on the action (i.e. *obligated*, *permitted*, or *prohibited*), the action which is deontically specified (indicated by the label and which can be distinct from the action that is executed), whether execution of the action flags for violation or fulfillment, and the agent which executes the action. Lists of contract flags are of type *ContractFlagState*.

Definition 4. *type ContractFlag = Rec (actionDone::String,*
deonticSpec::String, onSpec:: String,

valueFlag::String, agent::Agent)
type ContractFlagState = [ContractFlag]

The violation and fulfillment flags, which are *String* types that are values of *valueFlag*, are key in reasoning what follows from a particular flag. In other words, that an agent has violated an obligation on an action may imply that the agent *incurs* an additional obligation. Indeed, such reasoning is central to legal reasoning. This is further developed in the section below on *Contrary-to-Duty Obligations*.

A deontic specifier such as *obligated* is essentially a function from an *AgentiveAction* to a *ContractFlagState*. A list of actions *DBAction* and propositions *PropList* are also input for the purposes of code development.

Definition 5. *type obligatedCompFlag :: AgentiveAction →*
DBAction → [PropList] → ContractFlagState

In **Definition 6**, we give a sample of Haskell code which calculates a *ContractStateFlag* relative to an input agentive action *inAgentiveAction* (along with a lexicon and compatibility constraints). Expressions of the form *#label list* return the *value* associated with given the *label* found in the *list*. Expressions of the form [x | x ← P] are *list comprehensions* in Haskell; they are analogous to the set-builder notation of set theory, where for S = {x + 2 | x ∈ {1,. . . ,5} ∧ odd(x)}, the result is S = {3, 5, 7}. List comprehension works much the same way, but using lists rather than sets.

We discuss the code relative to the line numbers in **Definition 6**. Lines 1-2 constitute a *guard* on the function: if the action from the input agentive action *has* an opposite (i.e. is a non-empty list), only then do we return a non-empty *ContractStateFlag* list. Otherwise, we return the empty list (line 14). This reflects the conceptual point that there can only be obligations on an action where the obligation can be violated (cf. Wyner 2006). Thus, where we return a non-empty list, there is some action in opposition to the input action. In lines 3-7, we create a list of type *ContractState* which represents the *fulfillment* of the obligation on the action. In lines 7-13, we find the opposite to the input action and use it to create a list of type *ContractState* which represents the *violation* of the action. We use *++* to conjoin these to lists to produce a list of type *ContractFlagState*.

Definition 6. *obligatedCompFlag inAgentiveAction inDBAction inComp*
```
1     | ((findOpposites (#action inAgentiveAction)
2       inDBAction inComp) /= []) =
3         ([ (actionDone=(#label (#action inAgentiveAction)),
4           deonticSpec="Obligated",
5           onSpec=(#label (#action inAgentiveAction)),
6           valueFlag="Fulfilled",
7           agent=(#agent inAgentiveAction))] ++
8         [(actionDone=(#label x), deonticSpec="Obligated",
9           onSpec=(#label (#action inAgentiveAction)),
10          valueFlag="Violated",
```

```
11                    agent=(#agent inAgentiveAction))
12                      | x ← (findOpposites
13                      (#action inAgentiveAction) inDBAction [])])
14              | otherwise = []
```

To illustrate, let us assume that when we apply *obligatedCompFlag* to an agentive action labelled *Action6* with agent *Jill*. The output is:

Example 5. [(actionDone = Action6, agent = Jill, deonticSpec = Obligated,
onSpec = Action6, valueFlag = Fulfilled),
(actionDone = Action7, agent = Jill, deonticSpec = Obligated,
onSpec = Action6, valueFlag = Violated)]

This is of type *ContractStateFlag*. It indicates that were *Jill* to execute *Action6*, then *Jill* would have fulfilled her obligation on *Action6*. On the other hand, were *Jill* to execute *Action7*, then *Jill* would have violated her obligation on *Action6*.

As lists of records, we can manipulate them. For example, we can add to or subtract from contract states. For example, the following represents Jill's obligation with respect to *Action6* and Bill's prohibition with respect to *Action9*.

Example 6. [(actionDone = Action6, agent = Jill, deonticSpec = Obligated,
onSpec = Action6, valueFlag = Fulfilled),
(actionDone = Action7, agent = Jill, deonticSpec = Obligated,
onSpec = Action6, valueFlag = Violated),
(actionDone = Action9, agent = Bill, deonticSpec = Prohibited,
onSpec = Action9, valueFlag = Violated)]

Manipulations of *ContractStateFlag* expressions are crucial for *modelling* contract change, which is key to the analysis and implementation of Contrary-to-Duty Obligations.

4.5 Contrary-to-Duty Obligations

To model reasoning for CTDs, we enrich our States-Of-Affairs to include expressions of type *contractFlagState* as well as *histories* of type *history*. Histories are lists of records of what was done, when, by whom, and whether it counts as a fulfillment or violation relative to a deontic specification. Such records are of type *HistoryFlag*. They are much like *ContractState* expressions, but record the world and time at which the action is executed. An important difference between *HistoryFlag* and *ContractStateFlag* expressions is in *how they are processed*. This is further developed below.

Definition 7.
```
type HistoryFlag = Rec (actionDone::String,
                        deonticSpec::String, onSpec:: String,
                        valueFlag::String, agent::Agent,
                        world::World, time::Time)
type History = [HistoryFlag]
```

Our SOAs are enriched with both a *ContractFlagState* and a *History*.

Definition 8. *type SOAHistorical = Rec (properties::PropList, actionDone::String, history::History, contractFlagState::ContractFlagState, world::World, time::Time)*

Actions are executed relative to a *SOAHistorical*. Action execution *doAgentiveActionSOAHist* is essentially a function from *SOAHistorical* to *SOAHistorical*. We illustrate this informally below.

Let us suppose the following is the input *SOAHistorical* to *doAgentiveActionSOAHist*. Notice that the history is empty, which means that there is no evidence that an action has been executed.

Example 7.
(contractFlagState =
[(actionDone = Action6, agent = Jill,
deonticSpec = Obligated, onSpec = Action6,
valueFlag = Fulfilled),
(actionDone = Action7, agent = Jill,
deonticSpec = Obligated, onSpec = Action6,
valueFlag = Violated),
(actionDone = Action9, agent = Bill,
deonticSpec = Prohibited, onSpec = Action9,
valueFlag = Violated)],
history = [],
properties = [prop1, prop7, prop5, neg-prop4, neg-prop6],
time = 2, world = 7)

Suppose that Jill does execute *Action7* with respect to this *SOAHistorical*. This means that we should indicate that Jill has violated her obligation. Thus, in the *history* of the subsequent *SOAHistorical*, we record that Jill executed *Action7*. We also record that this action violates Jill's obligation to execute *Action6*, as well as the world and time stamp where the violation occurred. We also see that the time of the *SOAHistorical* is updated. The properties are updated as well.

Example 8.
(contractFlagState =
[(actionDone = Action6, agent = Jill,
deonticSpec = Obligated, onSpec = Action6,
valueFlag = Fulfilled),
(actionDone = Action7, agent = Jill,
deonticSpec = Obligated, onSpec = Action6,
valueFlag = Violated),
(actionDone = Action9, agent = Bill,
deonticSpec = Prohibited, onSpec = Action9,
valueFlag = Violated)],
history = [(actionDone = Action7, agent = Jill,

deonticSpec = obligated, onSpec = Action6,
time = 2, valueFlag = Violated, world = 7)],
properties = [prop1, prop7, prop5, prop3, neg-prop4, prop6],
time = 3, world = 7)

The next step in the implementation of CTDs is to allow contract state modification *relative to actions which have been executed in the history*. Recall from the discussion of CTDs that we only want a secondary obligation to arise *in a context where some other obligation has been violated*. In other words, if a particular violation of an obligation is marked in the *History*, we want a secondary obligation to be introduced into (or subtracted from) the *ContractStateFlag* of the *SOAHistorical*. For example, suppose Jill is obligated to leave the room. If Jill violates this obligation (by remaining in the room), then she incurs a secondary obligation to pay £5 to Bill. On the other hand, if Jill fulfills her obligation, then she incurs a secondary permission to eat an ice cream. The secondary obligations or permissions only arise in cases where a primary obligation has been violated or fulfilled.

To implement this, we have to examine whether a particular violation marker appears in the history. Second, we have to make that violation marker *trigger ContractStateFlag* modification. For instance, suppose that it is marked in the *History* that Jill has violated her obligation to do *Action6* by doing *Action7*. As a consequence of that, we modify the current contract state by removing her previous obligation and introducing an obligation on *Action11*. In such an operation, only the *ContractStateFlag* is modified. This gives the appearance of inference in a state, for there is no state change marked by temporal updating.

We have a function *doRDS*, which implements action execution for *relativized deontic specifications*; it is a function from *AgentiveActions* and *SOAHistorical* to *SOAHistorical*. It incorporates modification of the *ContractStateFlag*. Where we assume the steps just outlined to the *ContractStateFlag* in (7), a result is along the following lines:

Example 9. (contractFlagState =
[(actionDone = Action11, agent = Jill,
deonticSpec = Obligated, onSpec = Action11,
valueFlag = Fulfilled),
(actionDone = Action15, agent = Jill,
deonticSpec = Obligated, onSpec = Action11,
valueFlag = Violated),
(actionDone = Action9, agent = Bill,
deonticSpec = Prohibited, onSpec = Action9,
valueFlag = Violated)],
history = [(actionDone = Action7, agent = Jill,
deonticSpec = obligated, onSpec = Action6,
time = 2, valueFlag = Violated, world = 7)],
properties = [prop1, prop7, prop5, prop3, neg-prop4, prop6],
time = 3, world = 7)

The implementation captures the essence of the CTD problem. It models how the execution of an action relative to a *ContractFlagState* induces a modification of the *ContractFlagState*.

4.6 Deontic Specification on Complex Actions

We implement complex actions as records. Complex Actions have fields for the input actions, the complex action operator, and the result of the application of the operator to the input actions. We discuss here only the sequence operator, as it raises the more complex and interesting problems for deontic specification. We represent sequences schematically as follows.

Example 10. (inActionA = ActionA, inActionB = ActionB,
operator = SEQ, outAction = ActionC)

The *outAction* is, in this case, *function composition* of the input actions (*pace* several restrictions on well-formedness): the preconditions of *ActionC* are the preconditions of *ActionA*; the postconditions of *ActionC* are those of *ActionB* together with those of *ActionA* which remain by *inertia*; the preconditions of *ActionB* must be a subset of the postcondition properties of *ActionA*; and the postcondition properties of *ActionC* must otherwise be consistent. Our decomposition of actions into explicit preconditions and postconditions as well as our explicit construction of complex actions relative to those conditions distinguishes our approach from Dynamic Logic approaches, where there are basic actions.

In Meyer (1988), obligations on sequences are reduced to sequences of obligations on the component actions. In Khosla and Maibaum (1987), obligations on sequences are irreducible to sequences of obligations, but rather are obligations on the sequence *per se*. In Wyner (2006), we have further discussion of the significance of the difference, particularly the CTD problem. Here, we simply point out that the implementation provides ways to articulate these differences. For example, suppose *Jill* is the agent of the sequence and *ActionD* is the opposite of *ActionA* and *ActionE* is the opposite of *ActionB*. To provide the distributive interpretation of obligation in Meyer (1988), Obl_{dist}, we need two components. First, we have an initial contract state for the obligation on the first action:

Example 11. [(actionDone = ActionA, agent = Jill, deonticSpec = Obligated,
onSpec = ActionA, valueFlag = Fulfilled),
(actionDone = ActionD, agent = Jill, deonticSpec = Obligated,
onSpec = ActionA, valueFlag = Violated)]

In addition, we have a ContractStateModTrigger record which specifies that in the context where the first action has been executed (checked in the history), then the obligation on the second action of the sequence is introduced. This results in the following contract state, which specifies the fulfillment and violation cases for each of the *component* actions:

Example 12. [(actionDone = ActionA, agent = Jill, deonticSpec = Obligated,
onSpec = ActionA, valueFlag = Fulfilled),
(actionDone = ActionD, agent = Jill, deonticSpec = Obligated,
onSpec = ActionA, valueFlag = Violated),
(actionDone = ActionB, agent = Jill, deonticSpec = Obligated,
onSpec = ActionB, valueFlag = Fulfilled),
(actionDone = ActionE, agent = Jill, deonticSpec = Obligated,
onSpec = ActionB, valueFlag = Violated)]

We might say that the obligated sequence has been fulfilled where the obligations on each action have been fulfilled and in the right order.

In contrast, we could represent Khosla and Maibaum's interpretation (1987) by applying the operator to *ActionC* with a collective interpretation of obligation, Obl_{coll}. We suppose that *ActionF* is the opposite of *ActionC*:

Example 13. [(actionDone = ActionC, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Fulfilled),
(actionDone = ActionF, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Violated),

The most interesting case is the interruptable notion of obligation on a sequence. In this case, there is a violation and fulfillment flag with respect to the *whole sequence*, and the actions must apply in a given order. We assume the following initial contract state, where we emphasize that the marker for violation is relative to the complex action *per se* and there is no marker for fulfillement of the sequence:

Example 14. [(actionDone = ActionD, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Violated),

The second component is the ContractStateModTrigger, which specifies that *after* execution of the first action *ActionA*, an obligation to execute the second action arises such that fulfillment of this obligation marks fulfillment of the obligation of the sequence, while violation of this obligation marks violation of the obligation on the sequence. The resulting contract state looks like:

Example 15. [(actionDone = ActionD, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Violated),
(actionDone = ActionB, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Fulfilled),
(actionDone = ActionE, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Violated),

It is in such cases that a productive and compositional analysis comes to the fore.

These examples show that there are alternative definitions which can be used to define deontic specification on complex actions. The particular definitions may be designed to suit particular purposes and interpretations. The language

is thus very expressive and can be used to implement different notions of values applied to actions for the purposes of simulation in a multi-agent system. Further discussion appears in Wyner (2006).

5 Some Comparisons

There have been several recent efforts to operationalize deontic specifications. Some we have already discussed. For example, Garcia-Camino et. al. (2005) and Aldewereld et. al. (2005) appear to use deontic specifications to filter out or sort actions. We do not believe that this represents the essence of the deontic notions. Sergot (2006) uses the event calculus and only considers permissions. While we may eventually want to integrate deontic specifications into an event calculus, we would want to be clear about deontic specifications themselves; it does not seem necessary to add the additional and potentially obscuring components of the event calculus. In addition, Sergot (2006) has neither complex actions nor an analysis of the CTD problem. Boella and van der Torre (2006 present an architecture for normative systems which is similar in that deontic specifications *add* information to basic information. However, it is unclear how they implement their design, integrate complex actions, or account for the CTD problem.

6 Other Elements of the Implementation and Future Research

One key aspect of the implementation which we have not discussed here are *consistency* constraints and *implicational* relations between deontic specifications. For this, we define a notion of the *negation* of a deontic specification. We also introduce lexical relations between positive and negative deontic specifications. Further discussion appears in Wyner (2006).

We plan to enrich the structure of agents to give them some capacity to *reason* with respect to their goals, preferences, and relationships to other agents. As we want to model organizational behavior, we want to add *roles*, *powers*, a *counts as* relation between actions, and *organizational struture* to the implementation. The jural relations of *rights* and *duties* can also be incorporated into the language. While the implementation provides a significant and novel advance in the field, much yet remains to be done.

References

Aldewereld, et. al.: Designing Normative Behaviour by the Use of Landmarks. In G. Lindeman, et. al. (eds.) *Proceedings of AAMAS-05 International Workshop on Agents, Norms and Institution for Regulated Multi Agent Systems.* Utrecht, (2005), 5-18.

Anderson, A., Moore, O.: The Formal Analysis of Normative Concepts. The American Sociological Review. **22** (1957) 9-17

Boella, G., and v. d. Torre, L.: An Architecture of a Normative System. Proceedings of AAMAS'06, May 8-12, 2006, Hakodate, Hoddaido, Japan (2006)

Garcia-Camino, et. al.: A Distributed Architecture for Norm-Aware Agent Societies. In M. Baldoni et. al. (eds.) *Declarative Agent Languages and Technologies III, Third InternationalWorkshop, DALT 2005 Utrecht, The Netherlands, July 25, 2005*. London: Springer (2006)

Blackburn, P., Bos, J.: Representation and Inference for Natural Language: A First Course in Computational Semantics, Palo Alto, CA: CSLI Publications, (2005)

Carmo, J., Jones, A.: Deontic Logic and Contrary-to-duties. In D. Gabbay and Franz Guenthner (eds.) Handbook of Philosophical Logic, Dordrecht: Kluwer Academic Publishers, (2001)

Carmo, J., Jones, A.: Deontic Database Constraints, Violation, and Recovery. Studia Logica. **57** (1996) 139-165

d'Altan, P., Meyer, J.-J.Ch., Wieringa, M.: An integrated framework for ought–to–be and ought–to–do constraints. Artificial Intelligence and Law. **4** (1996) 77–111

Doets, K., van Eijck, J.: The Haskell Road to Logic, Maths and Programming, London: King's College Publications, (2004)

Dowty, D.: Word Meaning and Montague Grammar. Dordrecht, Holldand: Reidel Publishing Company (1979)

van Eijck, J.: Computational Semantics and Type Theory. Website download – http://homepages.cwi.nl/ jve/cs/, (2004)

Gilbert, N., Troitzsch, K.: Simulation for the Social Scientist, London, UK: Open University Press, (2005)

Harel, D., Kozen, D., and Tiuryn, J.: Dynamic Logic. Cambridge, MA: The MIT Press (2000)

Jones, A., Sergot, M.: On the Characterisation of Law and Computer Systems: the Normative Systems Perspective. In J.-J.Ch Meyer and R.J. Wieringa (eds.) Deontic Logic in Computer Science – Normative System Specification. Wiley (1993), 275-307

Kent, S., Maibaum, T., and Quirk, W.: Formally Specifying Temporal Contraints and Error Recovery. In Proceedings of the IEEE International Symposium on Requirements Engineering, IEEE C.S. Press, 208-215

Khosla, S., Maibaum, T.: The Prescription and Description of State-Based Systems. In B. Banieqbal, H. Barringer, and A. Pneuli (eds.) Temporal Logic in Specification. Springer-Verlag (1987) 243-294

Lomuscio, A. and D. Nute (eds.): Deontic Logic in Computer Science: Proceedings of the 7th International Workshop on Deontic Logic in Computer Science. R. Thomason (ed.), London, Springer, (2004)

Makinson, D.: On a Fundamental Problem of Deontic Logic. In P. McNamara and H. Prakken (eds.) Norms, Logics, and Information Systems. New Studies in Deontic Logic and Computer Science. IOS Press 1999 29-53

Meyden, R. v. d.: The Dynamic Logic of Permission. Journal of Logic and Computation. **6** (1996) 465-479

Meyer, J.-J.Ch.: A Different Approach to Deontic Logic: Deontic Logic Viewed as a Variant of Dynamic Logic. Notre Dame Journal of Formal Logic. **1** (1988) 109-136

Montague, R.: Formal Philosophy: Selected Papers of Richard Montague. R. Thomason (ed.), New Haven, Yale University Press, (1974)

Meyer, J.-J.Ch., Wieringa, R.J.: Actors, Actions, and Initiative in Normative System Specification. Annals of Mathematics and Artificial Intelligence. **7** (1993) 289-346

Penner, J., Schiff, D., Nobles, R. (eds.): Introduction to Legal Theory and Jurisprudence: Commentary and Materials. London, Buttersworth Law (2002)

Royakkers, L.: Representing Legal Rules in Deontic Logic. Ph.D. Thesis, Katholieke Universiteit Brabant, Tilburg (1996)

Sergot, M.: A Brief Introduction to Logic Programming and its Applications in Law. In C. Walter (ed.) Computer Power and Legal Language. Quorum Books (1988) 25-39

Sergot, M.: The Representation of Law in Computer Computer Programs. In T.J.M. Bench-Capon (ed.) Knowledge-Based Systems and Legal Applications. Academic Press (1991) 3-67

Sergot, M. and Richards, R.: On the Representation of Action and Agency in the Theory of Normative Positions. Fundamenta Informaticae. **48** (2001) 273-293

Sergot, M.: Normative Positions. In Henry Prakken and Paul McNamara (eds.) Norms, Logics and Information Systems. New Studies in Deontic Logic and Computer Science. IOS Press (1998) 289-310

Sergot, M.: A Computational Theory of Normative Positions. ACM Transactions on Computational Logic. **2** (2001) 581-622

Sergot, M.: $(C+)^{++}$: An Action Language for Modelling Norms and Institutions. technical report at http://www.doc.ic.ac.uk/research/technicalreports/2004/ DTR04-8.pdf

Wieringa, R.J., Meyer, J.: Deontic Logic in Computer Science: Normative System Specification. John Wiley and Sons (1993)

Wyner, A.Z.: Violations and Fulfillments in the Formal Representation of Contracts. ms King's College London, Department of Computer Science, submitted for the Ph.D. in Computer Science (2006)

Wyner, A.Z.: Maintaining Obligations on Stative Expressions in a Deontic Action Logic. In A. Lomuscio and D. Nute (eds.) Deontic Logic in Computer Science. Springer (2004), 258-274