Actor-Oriented Specification of Deontic Integrity Constraints

R.J. Wieringa
J.-J.Ch. Meyer

Department of Mathematics and Computer Science
Free University
De Boelelaan 1081a
1081 HV Amsterdam

uucp: roelw@cs.vu.nl, jules@cs.vu.nl

ABSTRACT

The logic of norms, called deontic logic, has been used to specify static, dynamic, and deontic integrity constraints for databases. For example, one can specify in deontic logic that a book borrowed from a library should be returned within three weeks, and that if it is not returned, the library should send a reminder. Intuitively, deontic logic presupposes the concept of an actor who undertakes actions and is responsible for fulfilling obligations. Until now, this concept has not been formalized. We present a formalization that increases the expressivity of database specifications, as well as allows us to solve a long-standing paradox of deontic logic.

1. Introduction

Deontic logic is the logic of permissions, prohibitions, and obligations. Surveys of several deontic logics that have been devised in the past have been given by Al-Hibri [2], Føllesdal [10], Kalinowski [18], and Åqvist [30]. Recently, deontic logic has been applied to the specification of software systems, in particular to the specification of information systems. Lee [23] applies traditional deontic logic as developed by Von Wright [36] to system specification, but others have developed new branches of deontic logic that are more suitable to software specification than the more traditional one. Fiadeiro and Maibaum [9] extend temporal logic with deontic operators, Khosla and Maibaum [20, 21], Van der Meyden [24] as well as Meyer [6, 26, 27] extend Harel’s dynamic logic [13] with deontic operators. In earlier papers, we applied Meyer’s logic to the specification of conceptual models of information systems [28, 33, 32]. We take this application as the point of departure in this paper. The approach is extended with the concept of an actor, and we start with listing some of the reasons why we want to do this.

1.1. The system as actor in the UoD

In [33], a library is specified in which an administration of books and library members is maintained. Members can borrow a book for three weeks, and are then obliged to return it. If they do not return it, the library will send a reminder. This is specified in the current version of the logic as

$$\forall p, b [\text{borrow}(p; b)]O(\text{return}(p; b)(\leq 21d))$$

(1)
\( \forall p, b \left[ \text{borrow}(p; b) \right] \left[ \text{clock}^{(21)} \right] \left( \text{PERF: borrow}(p; b) \rightarrow O(\text{remind}(\text{self}; p, b)) \right) \).

(2) says that after a person \( p \) borrowed a book \( b \), there is an obligation that the action \( \text{return}(p; b) \) occurs before the clock ticks 21 days. (2) says that after \( p \) borrowed \( b \), it is the case that after the clock ticks 21 days, if \( b \) is still borrowed by \( p \), then there is an obligation on the library (\text{self}) to send \( p \) a reminder. The intention of the formulas is that the object executing an action is the first argument of an action, separated from the other arguments by a semicolon. Thus, in (1), \( p \) executes \( \text{return} \) and in (2), \( \text{self} \) executes \( \text{remind} \). However, in the formal semantics there is nothing that expresses this intention. What is expressed by (1) and (2) is just that after certain events occur, certain obligations exist, without any formal indication of who does the action or who has the obligation. (We identify actions and events for the moment.) Still, this information is relevant. If (1) and (2) are used as integrity constraints in an automated information system, then (2) can be automated in such a way that the obligation specified in it is always fulfilled in a valid run of the system, because \text{self} is the automated system. But (1) cannot be so automated, because the obligation to return the book rests on the library member, and he or she cannot be specified to fulfill this obligation with certainty.

To be able to express that the system itself is an actor in the universe of discourse, we must first be able to express that there are actors at all. In this paper, we concentrate on the last idea. Every event occurrence will be labeled by an actor who initiates it. We will write \( t:a \) if the actor \( t \) initiates atomic event \( a \), and call \( t:a \) an \textit{action} (rather than an event). If we don't care who is the actor in an action is, we write \( a \) instead of \( t:a \).

1.2. The paradox of free choice permission

Traditionally, deontic logic has been plagued by numerous paradoxes. Castañeda and von Wright [5, 36] have proposed that a number of these paradoxes can be resolved by distinguishing actions from states. This approach has been formalized by Meyer [26, 27] using dynamic logic. The basic idea is to label the set of possible states as either forbidden or permitted, and to define any action that leads to a forbidden state as forbidden. Permission and obligation can then be defined in a standard way in terms of prohibition.

One paradox still remains, however, called the paradox of free choice permission, described, among others by Hilpinen [14] and Kamp [19]. This is that the following formula is derivable: (\( P(a) \) says that event \( a \) is permitted and + is a choice of events):

\[
(3) \quad P(\text{buy chewing gum}) \rightarrow P(\text{buy chewing gum + shoot the president}).
\]

Meyer [25] showed that this paradox can be resolved using the distinction between internal and external choice introduced by Hoare in CSP [15]. Briefly, CSP considers a world consisting of two actors, the machine \( m \) and its environment \( o \) (for observer). A choice made by \( m \) is called internal and one made by \( o \) is called external. Now, in Meyer's formalization of deontic logic, permission to do \( a \) means that there is a way of doing \( a \) that leads to a permitted state of the world. One reading of (3) is therefore intuitively plausible, viz. if there is a way to chew gum that leads to a permitted state, then there is a way to perform the process (buy chewing gum + shoot the president) that leads to a permitted world (viz. by performing the permitted way to chew gum). This is intuitively plausible only if someone else increases the set of options open to me. On the other hand, it is counterintuitive to conclude that I can make that choice. From \( P(\text{I chew gum}) \) I cannot conclude that I am permitted to choose between chewing gum and shooting the president.

In addition to allowing actors to initiate atomic events like buying chewing gum or shooting the president, we therefore allow actors to make a choice. \( t:(a + b) \) denotes the choice made by actor \( t \) between actions \( a \) and \( b \). Thus, (4) is a formalization of our intuition of the validity of (3) and (5) formalizes the intuition of the invalidity of (3):

\[
(4) \quad P(t_1:a_1) \rightarrow P(t_2:(t_1:a + t_1:b))
\]

\[
(5) \quad P(t_1:a_1) \rightarrow P(t_1:(t_1:a + t_1:b))
\]

(4) says that there is a possibility that \( t_2 \) makes the choice in such a way that a permitted world will
ensue after performing the chosen action. (5) makes the incorrect statement that if I am permitted to do something, then I am permitted to choose to do something else as well. The choice in (4) is called imposed choice, in (5) it is called free choice. The force of the permission $P(t_2:(t_1:a + t_1:b))$ is stronger than that of $P(t_2:(t_1:a + t_1:b))$, because in addition to saying that there is a possibility that $t_1$ chooses a permitted action, it says that $t_1$ is permitted to choose between the actions. In our system, (4) is a theorem and (5) is not. In fact, we have the theorems

\begin{align*}
(6) & \quad P(t_2:(t_1:a + t_1:b)) \iff P(t_1:a) \lor P(t_1:b) \\
(7) & \quad P(t_1:(t_1:a + t_1:b)) \rightarrow P(t_1:a) \land P(t_1:b),
\end{align*}

which agrees with our intuitions. (7) blocks the paradox of free choice permission.

1.3. Active objects

Formalization of the concept of an object is now receiving increasing attention [1, 4, 7, 22, 35]. Although there is as yet no formal definition of what an object is that is accepted by the database research community, there is increasing consensus that at least the concepts of encapsulation of state and behavior, and the use of globally unique object identifiers are essential [3]. We will show that our formalization of actors can be used comfortably to specify objects with at least these two characteristics, and in addition allows the formalization of the concept of an active object as an object that is capable of initiating actions. All objects then have a local state that can change by local events, but these events may be initiated themselves or by other objects. The object initiating an action must have been declared as an actor. This view of active objects has the advantage of agreeing with the intuitive concept of actions and their initiators that is prevalent in some system development methods, notably JSD [17].

1.4. Plan of the paper

In section 2 we show how to specify a set of possible DB states sharing a set of common abstract data types (ADT’s). In section 3 we show how to specify events, actions, and steps consisting of actions. In particular, we show how to specify the actors who initiate actions. In section 4 we specify dynamic integrity constraints (IC’s) which allow us to state what the effect of actions and steps is on the DB. In section 5 we show how to specify deontic IC’s that allow us to state who is permitted, forbidden, or obligated to do what. Section 6 sketches how to specify objects in this formalism, and in particular how active objects are specified. Finally, in section 8 we summarize the paper.

2. Static constraints and ADT’s

We use the equational paradigm of specifying ADT’s [8, 12, 11, 31]. A simple example is

\begin{verbatim}
datatype spec Persons
  import Booleans
  sorts PERSON
  operations
    p0 : PERSON
    next : PERSON -> PERSON
    eq : PERSON x PERSON -> BOOL
  variables
    p : PERSON
  equations
    [E1] p0 eq p0 = true
    [E2] next(p) eq next(p) = p eq p
    [E3] p eq next(p) = false
    [E4] next(p) eq p = false
end spec Persons
\end{verbatim}

Booleans is a specification of the set of Boolean values, called BOOL, and their operations. Sets of
data elements are called sorts. They may be partially ordered as defined in [12]. For example, we may declare EMPLOYEE to be a subsort of PERSON, by saying EMPLOYEE \( \leq \) PERSON. All operations declared for a supersort are inherited by its subsorts. The declarations of the sorts, their ordering, and the operations on the sorts form an ADT signature, and an ADT signature with a number of equations is an ADT specification. The equations are all universally quantified formulas, consisting of an equals sign flanked by two terms of compatible sorts. Below, we will use terms of sort PERSON as actor identifiers. The specification Persons of this type is omitted for reasons of space. The equations in Persons say that two PERSON identifiers are eq iff they are syntactically the same identifier. Identifiers carry no information except that of equality or difference with other identifiers, and that of type (an identifier has fixed type(s)).

We use the initial semantics of ADT’s as defined in the references listed above, which is the equational analogon of Herbrand models of logic specifications [12]. In the initial semantics, all data elements of any sort are denoted by closed (i.e. variable-free) terms of that sort. Two closed terms are equal precisely when they can be proved to be equal from the equations, using the rules of equational logic. (We may think of the usual inference rules of first-order logic with equality for the moment, applied only to formulas that are equations. These are given below.) By the close correspondence between closed terms of an actor sort like PERSON and what they denote, we will call a closed actor term as well as its denotation in the initial semantics actor identifiers.

In the initial semantics, elements of PERSON are denoted by the closed terms \( p_0 \), \( \text{next}(p_0) \), \( \text{next} \left( \text{next}(p_0) \right) \), etc. It can be proved that the equations in Persons reduce any closed term of the form \( p_1 \text{eq} p_2 \) to true or false. The eq operation does therefore not add "junk" to \( \text{BOOL} \) in the form of terms \( p_1 \text{eq} p_2 \) that are not true or false. The subset relation on sort names is interpreted as the subset relation on sets of data elements. For example, the set of EMPLOYEE data elements is a subset of the set of PERSON data elements.

Equational logic is extremely simple, since an equational language is just the set of all equations over a signature, and its inference relation basically uses the properties of the = sign. We extend this to first-order logic with equality to specify static IC’s as follows. A static constraint signature \( \text{Sig} \) is a triple \( ((S, \leq), \text{IF}, \text{IP}) \), where \( S \) is a set of sort names partially ordered by \( \leq \), \( \text{IF} \) is a set of function declarations over \( S \), and \( \text{IP} \) is a set of predicate symbols with their arity taken from \( S \).

An atomic \( \text{Sig} \)-formula over a static constraint signature is either an equation over the signature \( ((S, \leq), \text{IF}) \) or a formula \( P(t_1, ..., t_n) \), where \( P \) is declared to have arity \( s_1 \times ... \times s_n \) and \( t_i \) is a term of sort \( s_i \), \( i = 1, ..., n \). We use \( P, Q \) as metavariables over the predicate symbols. An order-sorted static constraint language \( L_{\text{Stat}}(\text{Sig}) \) over a static constraint signature \( \text{Sig} \) is the set of formulas defined by the following BNF:

\[
\phi ::= t_1 = t_2 \mid P(t_1, ..., t_n) \mid \neg \phi \mid \phi \land \psi \mid \phi \lor \psi \mid \phi \rightarrow \psi \mid \phi \leftrightarrow \psi \mid \forall x(\phi) \mid \exists x(\phi),
\]

where the first two alternatives are atomic \( \text{Sig} \)-formulas. We drop the qualification \( \text{Sig} \) when \( \text{Sig} \) is irrelevant or understood.

A static constraint specification \( \text{SpecStat} \) is a static constraint signature with a set of formulas over that signature. From the syntax, we can see that this subsumes ADT specifications. To distinguish functions declared in an ADT signature from those introduced in a static constraint signature, we call the first operations and the second functions. So function names can be interpreted differently in different possible worlds, but operations are interpreted everywhere the same, i.e. as they are interpreted in the underlying ADT. An example static constraint specification is

static constraint spec StaticLibraryConstraints
import
  Books, Queues, Persons
functions
  reservations : BOOK \rightarrow QUEUE
predicates
  Reserved : BOOK
Books and Persons are specifications of the ADT's of book and person identifiers. Queues is a specification of the ADT of queues of person identifiers. reservations(b) returns a queue of person identifiers. [S0] defines a book to be available iff it is present and there are no reservations for it. [S1] defines Reserved(b) as a shorthand for the test whether the queue of reservations is empty. This will be extended with dynamic constraints below, which say when a book becomes or ceases to be present, and that a person is added to the queue of reservations of a book when he reserves it.

The inference rules of $L_{Stat}$ are the usual ones of first-order logic, and rules concerning the substitution of equals for equals shown in table 1.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[Con1]</td>
<td>$t_1 = t_2$</td>
</tr>
<tr>
<td></td>
<td>$t[x \mapsto t_1] = t[x \mapsto t_2]$</td>
</tr>
<tr>
<td>[Sub1]</td>
<td>$t_1 = t_2$</td>
</tr>
<tr>
<td></td>
<td>$t_1[x \mapsto t] = t_2[x \mapsto t]$</td>
</tr>
<tr>
<td>[Con2]</td>
<td>$t_1 = t_2$</td>
</tr>
<tr>
<td></td>
<td>$P(t_1) \iff P(t_2)$</td>
</tr>
</tbody>
</table>

Table 1. Inference rules for first-order logic with equality

The intended model of the underlying $Spec_{ADT}$ is the initial semantics as explained briefly above. We call this semantics the underlying ADT. The intended model of $Spec_{Stat}$ is a Kripke structure consisting of all possible models of $Spec_{Stat}$ (in the first-order logic sense) that contain the underlying ADT as $Sig_{ADT}$-reduct. By this we mean that each sort and operation name declared in $Sig_{ADT}$ denotes the same sort and operation, respectively, in each possible state of the Kripke structure. Thus, not only is $2 + 3 = 5$ derivable from $Spec_{Stat}$ iff it is derivable from its underlying $Spec_{ADT}$ alone. This means, for example, that $2 + 3 = 5$ is derivable from a static IC specification iff it is derivable from its underlying ADT specification. If we want to know, on the other hand, what the value of reservations(p) is for a person p, we must do a DB lookup.
3. Processes

To define dynamic constraints, we go back to the underlying ADT specification and extend it with a specification of a set of possible processes. We call this specification the process theory. The process theory has an intended model, which we call the process algebra. Each element in the process algebra is a possible process. In this section, we define this without formal connection with the set of DB states, and in the next section, we define such a connection.

Our process theory SpecProcess is just an equational specification that extends the underlying ADT specification with sort and operation names. There must be one selected sort of interest called the process sort. Terms of this sort denote processes. The extension is required to be conservative, i.e. no new identifications are added to the underlying ADT specification. We also require it to contain the underlying ADT as a SigADT-reduct, i.e. all sort- and operation names defined in the underlying ADT specification keep their meaning when used in the process theory.

We divide SpecProcess into three parts, the specification of actions, of steps, and of processes.

process spec LibraryActions
import
  Persons, Books, Library, Money
sorts
  PERSON_EVENT, LIBRARY_EVENT, ACTION
functions
  borrow : BOOK -> PERSON_EVENT
  return : BOOK -> PERSON_EVENT
  reserve : BOOK -> PERSON_EVENT
  notify : PERSON x BOOK -> LIBRARY_EVENT
  pay : MONEY -> PERSON_EVENT
  _:=_ : PERSON x PERSON_EVENT -> ACTION
  _:=_ : LIBRARY x LIBRARY_EVENT -> ACTION
end spec LibraryActions

Person, Books, Library and Money are ADT specifications. Library declares a sort LIBRARY and a constant 1 of sort LIBRARY. (We consider a world with only one library.) The term 1:notify (p, b) stands for the action of the library 1 notifying p that book b is overdue. a, b, and c are used as metavariables over closed terms of sort ACTION, l and k are metavariables over actors, and a, b, c are metavariables over events. We will often refer to a as an atomic event and to a as an atomic action.

By requiring the SpecProcess to be a conservative extension of SpecADT, we know that pay ($2) = pay ($1 + $1). The proof is simply that in the underlying ADT of money, $2 = $1 + $1 is provable, and the result follows by the inference rule [Con1] applied to the term pay (x).

To be able to refer to persons and libraries as actors, we simply assume that the underlying ADT specification contains the specification

datatype spec LibraryActors
import
  Persons, Library
sorts
  PERSON <= ACTOR
  LIBRARY <= ACTOR
end spec LibraryActors

By the initial semantics, actors are either persons or libraries, but they are never books. By the reduct requirement, we know that each closed term of sort ACTOR denotes the same actor in each possible world, and that different closed terms (on which eq delivers false) denote different actors.

The next specification specifies steps, processes that contain no sequence operator.
process spec StepAxioms
import
LibraryActions, LibraryActors
sorts
SCHEDULING_STEP
ACTION \leq ~STEP
functions
any : SCHEDULING_STEP
fail : SCHEDULING_STEP
+ : STEP x STEP \rightarrow SCHEDULING_STEP
& : STEP x STEP \rightarrow SCHEDULING_STEP
+ : ACTOR x SCHEDULING_STEP \rightarrow STEP
- : STEP \rightarrow STEP
variables
i, i0, i1, i2: ACTOR
s, s1, s2: STEP
e1, e2: SCHEDULING_STEP
equations
[S1] i:(s1 + s2) = i:(s2 + s1)
[S2] i:(s1 & s2) = i:(s2 & s1)
[S3] --s = s
[S4] (not ((10 eq i1) and (10 eq i2)) = true) \rightarrow
     ((10:(i1:e1 + i2:e2)) = 10:(-i1:e1 & -i2:e2))
[S5] (not ((10 eq i1) and (10 eq i2)) = true) \rightarrow
     ((-i0:(i1:e1 + i2:e2)) = 10:(-i1:e1 + -i2:e2))
end spec StepAxioms

We use the convention that n-ary operators bind stronger than m-ary operators, n<m, but if we wish we
can add brackets to emphasize operator binding. We use \alpha as metavariable over terms of sort STEP
and \alpha as metavariable over SCHEDULING_STEP. It is easily shown that \alpha either is a negated action
or has the form t: \alpha. In general, t: \alpha says that actor t takes the initiative to the scheduling step \alpha, or
schedules \alpha. \alpha may be an atomic event a. In t: any, t executes any atomic event, and in t: fail, t stag-
nates eternally. t:(\alpha_1 + \alpha_2) denotes the choice of t between \alpha_1 and \alpha_2, t:(\alpha_1 & \alpha_2) denotes the choice
of t to let \alpha_1 and \alpha_2 occur synchronously, and \neg \alpha denotes the non-performance of \alpha.

[S4–S5] are conditional axioms. We assume that a Boolean function eq with infix notation is
defined for each actor sort, which returns true iff its arguments are equal. The axioms require multi-
actor steps to be a Boolean algebra. The intuition behind this is that the effect of not making a choice
between \alpha_1 and \alpha_2 is the same as the effect of not doing \alpha_1 and \alpha_2. We call this the “extensional
reading” of choice, by which we mean that they make De Morgan’s laws valid in the semantics. In a
single-actor step, we use what we will call an “intensional reading”, and look at the choice of the sin-
gle actor as an event in itself. On that reading, choice and synchronization are simply different actions
and the axioms [S4–S5] do not hold. We come back to this when the difference between the two
choices is made more precise.

For reasons of space, the specification of processes (sequences of steps) is omitted. Elsewhere
[29] we give this specification as well, together with a detailed discussion of the different kinds of non-
deterministic processes one can express using actors, and a comparison with CCS and CSP.

4. Dynamic integrity constraints
Next, we define the effect of each possible step (in general, of each possible process) on the DB state as
a set-valued function. In general, each step (process) will take us from a DB state to one out of a set of
possible next states. This definition is part of the semantics of processes and is called the behavioral
semantics of processes, to distinguish it from the process algebra, which is called the algebraic seman-
tics.

A dynamic constraint signature SigDyn consists of a triple of signatures (SigADT, SigStat, SigProcess) such that SigStat and SigProcess
are static IC and process signatures, respectively, that share SigADT as their underlying ADT signature. Other than that, SpecStat and SpecProcess
have no sort or operation names in common.

The language $L(S_{IgDyn})$ of dynamic constraints, with typical elements $\Phi$ and $\Psi$, is an extension of $L(S_{IgProc})$ with elements given by the BNF:

$$\Phi ::= \phi \mid \Phi_1 \land \Phi_2 \mid \neg \Phi \mid \Phi_1 \land \Phi_2 \mid \Phi_1 \rightarrow \Phi_2 \mid \Phi_1 \leftrightarrow \Phi_2 \mid [\beta]\Phi$$

where $\phi$ is a formula of $L(S_{IgStat})$ and $\beta$ is a process term over $S_{IgProc}$. We use $\langle \beta \rangle \Phi$ as an abbreviation of $\neg [\beta] \neg \Phi$. Thus, a formula in $L(S_{IgDyn})$ is either an equation between process terms or a formula of the above form, of which $L(S_{IgStat})$ is a special case (containing in its turn equations from $L(S_{IgADT})$ as a special case). Note that by our definition of $L(S_{IgDyn})$-formulas, process terms can only occur inside the modal operator or else as part of a process equation.

The intuitive semantics of $[\beta]\Phi$ is "after execution of $\beta$, $\Phi$ holds necessarily", and the intuitive semantics of $\langle \beta \rangle \Phi$ is "after execution of $\beta$, $\Phi$ may hold". The language can thus be used to express pre- and postconditions of events.

The inference relation $\vdash_{Dyn}$ is an extension of the inference rules for first-order logic with equality with the rules in table 2.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[N]</td>
<td>$\Phi$</td>
</tr>
<tr>
<td>[Sub2]</td>
<td>$\beta_1 = \beta_2$</td>
</tr>
<tr>
<td>[DL1]</td>
<td>$[\beta](\Phi_1 \rightarrow \Phi_2) \rightarrow ([\beta]\Phi_1 \rightarrow [\beta]\Phi_2)$</td>
</tr>
</tbody>
</table>

Table 2. Inference rules for dynamic logic

This monotonically extends $\vdash_{Stat}$ and $\vdash_{Eq}$, so we can drop the subscript.

$Spec_{Dyn} = (S_{IgDyn}, C_{Dyn})$ is a dynamic constraint specification over the static constraint specification $Spec_{Stat}$ and the process specification $Spec_{Proc}$ if it is a conservative extension of both $Spec_{Stat}$ and $Spec_{Proc}$, i.e. such that

$$Th(Spec_{Stat}) = Th(Spec_{Dyn}) \cap L_{Stat}(S_{IgStat})$$
$$Th(Spec_{Proc}) = Th(Spec_{Dyn}) \cap L_{Proc}(S_{IgProc}).$$

An example dynamic constraint specification fragment is

dynamic constraint spec DynamicLibrary
  import
    StaticLibraryConstraints, ProcessAxioms
  variables
    p : PERSON
    b : BOOK
    q : QUEUE
    l : LIBRARY
  dynamic constraints
    [D0] (p:borrow(b)) Borrowed(b, p) \land \neg Present(b)
    [D1] Reserved(b) \rightarrow (p eq first(reservations(b)) = true \rightarrow
         (p:borrow(b)) reservations(b) = tail(reservations(b)))
    [D2] Reserved(b) \land p eq first(reservations(b)) = false \rightarrow
         (p:borrow(b)) false
    [D3] (p:reserve(b)) Present(b) \land \neg Borrowed(b, p)
    [D4] Present(b) \rightarrow (p:reserve(b)) false
    [D5] \neg Present(b) \rightarrow (p:borrow(b)) false
    [D6] reservations(b) = q \rightarrow
         (p:reserve(b)) reservations(b) = add(p, q)
    [D7] Present(\ell) \rightarrow (p:reserve(b)) false
    [D8] reservations(b) = q \rightarrow
         (\ell:notify(p, b)) [clock(\ell)]
         (front(q) = p \rightarrow reservations(b) = tail(q))
end spec DynamicLibrary
[D0] says that the effect of borrowing a book is that it is not present and is borrowed. [D1] says that in the case of borrowing reserved books, we in addition remove the borrower from the queue of reservations if he is the first in the queue. If the borrower is not first in queue, [D2] says the borrowing event is blocked. [D3] defines the effect of returning a book, and [D4–5] say that a book alternates between being borrowed and returned. [D6–7] define the effect of reserving a book, and [D8] cancels a reservation 7 days after a reserver is notified of its presence.

The possible models of $Spec_{Dyn}$ have the structure shown in figure 1.

$Spec_{Dyn}$ extends a static IC specification $Spec_{stat}$ and a process specification $Spec_{process}$, that have models $PW$ and $A_{Process}$, respectively. The underlying ADT is a reduct of each possible world (black dot) in $PW$ as well as of the process algebra $A_{Process}$. Each process term $\beta$ denotes an element in $A_{Process}$, called the algebraic semantics of the process term. The behavioral semantics of $\beta$ is given by a function $\rho$, which assigns an accessibility relation on $PW$ to $\beta$. Assuming $\beta$ is closed, $\rho(\beta)$ applied to $w \in PW$ yields the set of possible next worlds that can result from executing $\beta$. Execution of $\beta$ will lead to one of these worlds. More precisely,

$$\rho: T_{Process}(X) \longrightarrow (\Sigma \longrightarrow (PW \longrightarrow \mathcal{P}(PW))),$$

where $X$ is a set of variables, $T_{Process}(X)$ is the set of process terms of $Sig_{Process}$ with variables from $X$, and $\Sigma$ is the set of all sort-preserving assignments $\sigma$ to variables. $\rho$ accepts, in sequence, a process term, an assignment to variables in the term, and a possible world, and then delivers the set of possible next worlds that can be reached by assigning this process term (given this assignment to variables). In order to enforce soundness and completeness of the inference rules, we require of $\rho$ that it satisfies the requirement that for $\beta_1, \beta_2 \in T_{Process}(X),$

$$\llbracket \beta_1 \rrbracket_{\sigma, A_{Process}} = \llbracket \beta_2 \rrbracket_{\sigma, A_{Process}} \implies \rho(\beta_1)(\sigma) = \rho(\beta_2)(\sigma).$$

All sort names have a fixed interpretation in all possible worlds, which is defined in the underlying ADT specification or in the process specification. So if $x$ is declared of sort $s$ and $\sigma$ is a sort-preserving assignment, then $\sigma(x)$ is an element of $\llbracket s \rrbracket_d$, where $d$ is either $d_{ADT}$ or $d_{Process}$.

If $M_{Dyn} = (A_{ADT}, PW, A_{Process}, \rho)$ is a model of $Sig_{Dyn}$ and $\sigma: X \longrightarrow M_{Dyn}$ is an assignment to all variables, then truth in $w \in PW$ under assignment $\sigma$ of a dynamic logic formula is defined by:

- for each $\beta \in T_{Process}(X)$, we have $w, \sigma \models_{Dyn} [\beta] \Phi$ iff for all $w' \in \rho(\beta)(\sigma)(w)$, we have $w', \sigma \models_{Dyn} \Phi$.

- Truth of the other dynamic logic formulas is defined as usual.

Truth of a formula in a world, in a model, and of a specification in a model are defined as usual. This truth definition coincides with the standard truth definition for $\phi$ in $L(Sig_{Stat})$, which we denote $\models_{Stat}$, and with the standard truth definitions for equations in $L(Sig_{ADT})$, which we denote $\models_{ADT}$. It can be shown easily that

$$w \models_{Stat} \phi \iff w \models_{Dyn} \phi \text{ and }$$

$$w \models_{ADT} t_1 = t_2 \iff w \models_{Stat} t_1 = t_2.$$

We will therefore omit the subscript from $\models$ from now on.
Theorem 1.
$Spec_{Dyn} \models \Phi$ iff $Spec_{Dyn} \models \Phi$.
The proof of this is sketched in [29]. A detailed proof is forthcoming in [34].

Theorem 2.
The following theorems hold in $Spec_{Dyn}$.

- [Dyn1] $[\beta]true$
- [Dyn2] $[\beta]false$
- [Dyn3] $[\beta](\Phi_1 \land \Phi_2) \leftrightarrow ([\beta]\Phi_1 \land [\beta]\Phi_2)$
- [Dyn4] $[\beta](\Phi_1 \vee \Phi_2) \leftrightarrow ([\beta]\Phi_1 \vee [\beta]\Phi_2)$
- [Dyn5] $[\beta](\Phi_1 \vee \Phi_2) \leftrightarrow ([\beta]\Phi_1 \land [\beta]\Phi_2)$

Out of all possible models of a $Spec_{Dyn}$, we choose one as intended model, which we think lies close to the way we normally think of databases, and which solves the paradox of free choice permission as well. The choices for the underlying ADT specification (initial semantics), the static IC specification (set of possible worlds containing the underlying ADT as reduct) and the process specification (initial algebra), have already been made. It remains to define a behavioral semantics $p$ for the processes.

We do this by assuming a function

\[ effect: T_{EVENT}(X) \rightarrow (\Sigma \rightarrow (PW \rightarrow PW)) \]

which defines the effect of an atomic event on the state of the world. EVENT is a supersort of the event sorts. In general, a dynamic logic specification does not determine the effect of events exhaustively. Several effect functions remain possible with respect to a given specification, and we need a kind of frame assumption to choose between these possibilities. For example, one can stipulate that whatever is not specified to change does not change when an atomic event is applied. We leave open how effect is chosen, and require only that the function satisfies

\[ \text{if } [[a]]_\sigma, \mathcal{A}_{proc} = [[b]]_\sigma, \mathcal{A}_{proc} \text{ then } effect(a)(\sigma) = effect(b)(\sigma). \quad (E) \]

Thus, the effect of pay ($\$$2) and pay ($\$$1 + $\$$1) on the state of the DB will be the same, because the two terms are semantically equal in the process algebra.

Next, we define a step as a set of the form

\[ \{ \{a_1, a_2, \ldots\}, \{b_1, b_2, \ldots\}, \ldots\} \]

where the elements are finite sets of (possibly open) terms of sort STEP called synchronization sets. We require that if $t:fail$ is in a synchronization set, then $t$ does not participate with other events in the synchronization set. A synchronization set $s$ is called compatible if for all $w \in PW$ and all $\sigma \in \Sigma$

\[ effect(a_1)(\sigma) \cdots effect(a_n)(\sigma)(w) = effect(a_1)(\sigma) \cdots effect(a_n)(\sigma)(w) \]

for all permutations $\{i_1, \ldots, i_n\}$ of $\{1, \ldots, n\}$ of all non-failure events in the set. effect is extended to compatible synchronization sets by defining $effect(s)(\sigma)$ to be the composition of $effect(a_j)(\sigma)$ as shown above. Compatible synchronization sets contain events that can be executed synchronously. We may think of these as DB transactions, that take one in a single atomic step to a next DB state but may be executed as a sequence of sub-atomic updates. We define step($a$) as the set of all compatible synchronization sets containing $a$ as element. So step($a$) has the form

\[ \{ \{\ldots,a,\ldots\}, \{\ldots,a,\ldots\}, \ldots\} \]

Each step determines an accessibility relation on possible worlds, defined as follows.

1. $p(a)(\sigma)(w) = \{effect(s)(\sigma)(w) \mid \text{for all } s \in \text{step}(a)\}$.
2. $p(t:fail)(\sigma)(w) = \text{as in 1, with } a \text{ replaced by } t:fail$. 


3. \( \rho(t:\text{any})(\sigma)(w) = \rho(t:\text{any})(\sigma)(w) \) as in 1, with \( t:\text{any} \) replaced by \( t:\text{any} \).

4. \( \rho(-\alpha)(\sigma)(w) = \{\text{effect}(s)(\sigma)(w) \mid s \in \text{SYN}\backslash \text{step}(\alpha)\} \).

5. \( \rho(t_0:(t_1:\alpha_1 + t_2:\alpha_2))(\sigma)(w) = \rho(t_1:\alpha_1)(\sigma)(w) \cup \rho(t_2:\alpha_2)(\sigma)(w) \), where \( t_0, t_1, \) and \( t_2 \) are not all equal.

6. \( \rho(t:1)(\alpha_1 + t:2))(\sigma)(w) = \rho(t:1)(\alpha_1)(\sigma)(w) \cap \rho(t:2)(\alpha_2)(\sigma)(w) \).

7. \( \rho(t_0:(t_1:\alpha_1 + t_2:\alpha_2))(\sigma)(w) = \rho(t_1:\alpha_1)(\sigma)(w) \cap \rho(t_2:\alpha_2)(\sigma)(w) \).

Remarks:

1. \( \text{step}(\alpha)(\sigma) \) is the set of all compatible synchronization sets in which \( \alpha \) participates, so the order of applying the \text{effect} function in 1 is immaterial.

2. \( \rho(t:\text{fail})(\sigma)(w) \) is the set of all compatible synchronization sets containing \( t:\text{fail} \). By the definition of compatibility, it will not contain any \( t:a \).

3. \( \rho(t:\text{any})(\sigma)(w) \) contains all compatible synchronization sets in which \( t \) participates, but not with \( t:\text{fail} \).

4. Actions are negated with respect to all possible steps. Thus, \( -t:a \) is the set of all possible synchronization sets in which \( t \) does not participate with \( a \). These are the synchronization sets in which \( t \) participates with another event, or with \( \text{fail} \), or in which \( t \) does not participate at all.

5. Choice makes \( \rho \) deliver a function \( PW \rightarrow \mathcal{P}(PW) \) rather than \( PW \rightarrow PW \). In general, each set has a set of possible next worlds, one of which will be actually reached when executing the step. The intuitive notion captured by this semantic definition is that if \( t_0 \) chooses between two steps, the set of possible next world that may result from his choice is the union of the sets of possible next worlds reachable by the branches. Note that, if \( \rho(\alpha_i)(\sigma)(w) \) for \( i = 1, 2 \) consists of compatible synchronization sets, then so does \( \rho(t:(\alpha_1 + \alpha_2))(\sigma)(w) \).

6. If the three actors involved are the same, then we capture a quite different intuition with the semantics, viz. what the effect of \( t \)'s choice itself is, rather than what the effect of the chosen branches is. It says that the effect of the making the choice itself is the intersection of the effects of the branches. This agrees with our stronger interpretation of \( P( (t:(1:a + 1:b), \) which says that \( t \) is permitted to do \( a \) and \( b \) (just as in the multi-actor case), and in addition that \( t \) is permitted to make the choice.

7. There are no special considerations for synchronization that distinguish the single-actor from the multi-actor case. \( \alpha_1 \) & \( \alpha_2 \) is the synchronous execution of two steps, and the effect will be brought about by those functions on \( PW \) that bring about the effect of \( \alpha_1 \) and \( \alpha_2 \). Hence, we intersect the steps. Note that the effect of this synchronous execution is the same as the effect of the choice event in a single-actor choice between \( t:\alpha_1 \) and \( t:\alpha_2 \). This is compatible with the process terms whose effect this is, being unequal. Note that \( \alpha_1 \) and \( \alpha_2 \) may be \textit{incompatible}, so that there is no synchronization set in which both participate. For incompatible steps \( \alpha_1 \) and \( \alpha_2 \), we have \( \rho(t:(\alpha_1 & \alpha_2))(\sigma)(w) = \emptyset \). This has an important consequence for the logic, which we will see below.

We call this model a \textit{free choice} model because it allows us to differentiate a free from imposed choice.

Theorem 3.

\( \mathcal{M}_f \) is a model of \textit{SpecDyn}.

Theorem 4.

If \( t_0, t_1 \) and \( t_2 \) are not all equal, then the following formulas are true in \( \mathcal{M}_f \).

\[ [F1] (t_0:(t_1:\alpha_1 + t_2:\alpha_2))(\Phi) \leftrightarrow (t_1:1 \alpha_1)(\Phi) \land (t_2:1 \alpha_2)(\Phi) \]

\[ [F2] (t_0:(t_1:\alpha_1 + t_2:\alpha_2))(\Phi) \leftrightarrow (t_1:1 \alpha_1)(\Phi) \lor (t_2:1 \alpha_2)(\Phi) \]
Remarks:

1. [F2] represents our "extensional" reading of a multi-actor choice. \( \Phi \) may be true after \( t_0 \)'s choice iff it may be true as an effect of \( \alpha_1 \) or \( \alpha_2 \). By the duality \( [a] \Phi \equiv \neg [\neg a] \neg \Phi \), we have [F1], saying that \( \Phi \) will be true after \( t_0 \)'s choice iff it will be true after \( \alpha_1 \) and \( \alpha_2 \).

2. In a single-actor choice, [F4] reflects our "intensional" reading that if \( \Phi \) may be the effect of \( t \)'s choice, then it may be the effect of each of the branches. The arrow goes only one way, because there may not be any joint effect of \( \alpha_1 \) and \( \alpha_2 \) at all, viz. when they are incompatible. However, if there is an effect, i.e. if there is a \( \Phi \) with \( \langle t:\langle \alpha_1 \rangle \Phi \rangle \Phi \), then we can conclude that \( \langle t:\langle \alpha_1 \rangle \Phi \rangle \Phi \). To understand the dual formula in [F3], we must realize that choice is an event that occurs before the branches are executed. Choice does not bring us to a next possible world, but it does occur at a point in time preceding the execution of \( \alpha_1 \) and \( \alpha_2 \). The left-hand side of [F4] then says

"after \( t \)'s choice, the system is in a state where \( \Phi \) can be brought about",

which implies the right-hand side, both branches can bring about \( \Phi \). Applying the duality, the left-hand side becomes

"it is not the case that after \( t \)'s choice, the system is in a state where \( \neg \Phi \) can be brought about",

which is equivalent to

"after \( t \)'s choice, the system can be in a state where \( \Phi \) will be brought about".

This is the correct reading of the left-hand side of [F3], and it is implied by the right-hand side, which says that one of the branches will bring about \( \Phi \).

3. Synchronous execution is impossible if the synchronized steps are incompatible, so there is a one-way arrow here as well. The logic is not able to express necessary conditions for two steps to be compatible. This is a general problem with the intersection of accessibility relations in a Kripke model with multiple accessibility relations, that can only be solved by strengthening the language. Meyer [27] did this by adding \textit{DONE} \( :a \) predicates to the language, but Van der Hoek and Meyer [16] show how to do this in general.

5. Deontic integrity constraints

\( L(\text{SigDyn}) \) can be used as a language for deontic constraints by introducing violation states \( V_i: t:\alpha \), one for each of the reasons why \( t \) would be forbidden to perform \( \alpha \). These are implicitly declared in any specification. Each \( V_i: t:\alpha \) is a predicate with 0 or more arguments. (If it has 0 arguments, it is a propositional atom.) Then deontic modalities are introduced by defining abbreviations as follows.

\[
\begin{align*}
\Delta \quad P(\alpha) & \iff \neg [\alpha] V_i: \alpha \text{ for an } i, \text{ (" \alpha \) is permitted"}, \\
\Delta \quad O(\alpha) & \iff \neg [\alpha] V_i: \alpha \text{ for an } i, \text{ (" \alpha \) is obligatory"}, \\
\Delta \quad F(\alpha) & \iff \neg P(\alpha) \text{ (" \alpha \) is forbidden"}.
\end{align*}
\]

The semantics of this formalization of deontic logic has been studied in dynamic logic without actors in [26, 27]. This logic has been applied to system specification in [33, 32]. With actors, the modalities express more. For example, \( P(t:a) \) says that \( t \) is permitted to do to \( a \), and \( P(t:(t:a + t:b)) \) says that \( t \) is permitted to choose between doing \( a \) or \( b \) (i.e. choosing brings him into a state where he can do a permitted action). A deontic IC specification is just a dynamic IC specification that uses the (implicitly defined) violation predicates. An example is
deontic constraint spec DeonticLibraryConstraints

import DynamicLibraryConstraints

variables
P : PERSON
b, b' : BOOK
l : LIBRARY

deontic constraints

[N0] Permitted(p: borrow(b)) ↔ Member(p) ∧ ¬ V:p:-return(b')

[N1] [p: borrow(b)] [clock(21)] (Borrowed(b, p) → V:p:-return(b))

[N2] V:p:-return(b) →

[p: return(b)] (V:p: return(b) ∧ ¬ V:p:-return(b))

[N3] [p: pay($2, b)] → V:p: return(b)

[N4] [p: borrow(b)] Obligated(p: return(b), < 21)

[N5] [p: borrow(b)] [clock(21)] {Borrowed(b, p) → Obligated(l: remind(p, b))}

[N6] Present(b) ∧ Reserved(p) →

(p = front(reservations(b)) → Obligated(l: notify(p, b)))

end spec DeonticLibraryConstraints

[N0] says that a book can be borrowed only by members who are not in violation of the constraint on returning a book within three weeks. [N1] says that this constraint is violated if the book is still borrowed after 21 days. clock(21) is the process in which the clock makes 21 ticks (defined in [28, 33]). [N2] says that returning the book too late cancels the violation of not returning it, but raises another one, which [N3] says can be canceled by paying $2. [N4] says that borrowing a book creates the obligation to return it within 21 days. Obligated(p: return(b), < 21) is an obligation to perform a choice of steps, viz. return the book, or let the clock tick once and return the book, etc. If a book is not returned within 21 days, [N5] says that the library is obligated to send a reminder. Finally, if a book is returned, the library should notify the person who is the first reserver of the book ([N6]). Reserved(p) shields the application of front to an empty queue.

Theorem 5.
If t₀, t₁ and t₂ are not all equal, then the following formulas are true in Μ_f.

[P1] F(t₀:(t₁: α₁ + t₂: α₂)) ↔ F(t₁: α₁) ∧ F(t₂: α₂)

[P2] P(t₀:(t₁: α₁ + t₂: α₂)) ↔ P(t₁: α₁) ∨ P(t₂: α₂)

[P3] F(t₁:(t₀: α₁ + t₂: α₂)) ↔ F(t₁: α₁) ∨ F(t₂: α₂)

[P4] P(t₁:(t₀: α₁ + t₂: α₂)) → P(t₀: α₁) ∧ P(t₂: α₂)

[P5] F(t₀:(t₁: α₁ & t₂: α₂)) ↔ F(t₁: α₁) ∨ F(t₂: α₂)

[P6] P(t₀:(t₁: α₁ & t₂: α₂)) → P(t₀: α₁) ∧ P(t₁: α₂)

[P2] says that t₀ is permitted to give two actors a choice iff at least one of the actors can do something permitted. t₀ is permitted to choose, because his choice may lead to a permitted state of the world. By duality, he is forbidden to do so iff both actors are given forbidden things to do ([P1]). [P4] says that if t is permitted to make a choice between α₁ and α₂, then he is permitted to perform either branch. By duality, if he is forbidden to do either branch, then he is forbidden to make the choice ([P3]). This resolves the paradox of free choice permission. If any actor is permitted to synchronize two steps, then both steps are permitted ([P5]). If at least one step is forbidden, then any actor is forbidden to synchronize them ([P6]).

6. Conclusion
We defined the syntax, semantics and sound and complete inference rules for a logical language to specify static, dynamic, and deontic integrity constraints for databases. We showed how the introduction of actors can resolve a long-standing paradox of deontic logic, the paradox of free choice permission. Current and future work will include the giving an operational semantics to a reduced version of the language, and studying the connections between object-oriented and deductive databases, using object identifiers as defined in this paper as bridging concept. In particular, the concept of an active object will be formalized.
7. References


