Annals of Mathematics and Artificial Intelligence 7(1993)289–346

Actors, actions, and initiative in normative system specification

R.J. Wieringa
Department of Mathematics and Computer Science, Vrije Universiteit, De Boelelaan 1081a, NL-1081 HV Amsterdam, The Netherlands

J.-J.Ch. Meyer
Vrije Universiteit, Amsterdam and University of Nijmegen, The Netherlands

Abstract

The logic of norms, called deontic logic, has been used to specify normative constraints for information systems. For example, one can specify in deontic logic the constraints 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. Thus, the notion of obligation to perform an action arises naturally in system specification. Intuitively, deontic logic presupposes the concept of an *actor* who undertakes actions and is responsible for fulfilling obligations. However, the concept of an actor has not been formalized until now in deontic logic. We present a formalization in dynamic logic, which allows us to express the actor who initiates actions or choices. This is then combined with a formalization, presented earlier, of deontic logic in dynamic logic, which allows us to specify obligations, permissions, and prohibitions to perform an action. The addition of actors allows us to express *who* has the responsibility to perform an action. In addition to the application of the concept of an actor in deontic logic, we discuss two other applications of actors. First, we show how to generalize an approach taken up by De Nicola and Hennessy, who eliminate τ from CCS in favor of internal and external choice. We show that our generalization allows a more accurate specification of system behavior than is possible without it. Second, we show that actors can be used to resolve a long-standing paradox of deontic logic, called the paradox of free-choice permission. Towards the end of the paper, we discuss whether the concept of an actor can be combined with that of an object to formalize the concept of active objects.

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 [1], Føllesdal [14], Kalinowski [25], and Åqvist [2]. Recently, deontic logic has been applied to the specification of software systems, in particular to the specification of information systems. Lee [31] applies traditional deontic logic as developed by

Von Wright [52] 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 [13] extend temporal logic with deontic operators, Khosla and Maibaum [28,29], Van der Meyden [32], as well as Meyer [10,34] extend dynamic logic [20,30] with deontic operators. In earlier papers, we applied Meyer's logic to the specification of conceptual models of information systems [48,51]. 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.

This paper is a revision and an extension of two abstracts that appeared earlier [37,49]. Those abstracts contained a formalization of actors, active choice, that we have replaced, in the current paper, by a formalization in terms of passive and active choices. This allows us to simplify the approach at some points, while at the same time making it more expressive.

1.1. THE SYSTEM AS ACTOR IN THE UoD

In an earlier paper [48], we specified a library 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

$$\forall p, b[borrow(p; b)]\mathbf{O}(return(p; b)_{(\leq 21d)}) \tag{1}$$

$$\forall p, b[borrow(p; b)][\mathbf{clock}^{(21)}](PERF : borrow(p; b) \rightarrow \mathbf{O}(remind(self; p, b))). \tag{2}$$

Formula (1) says that after occurrence of the event $borrow(p; b)$, the obligation predicate $\mathbf{O}(return(p; b)_{(\leq 21d)})$ holds. The meaning of this predicate is that there is an obligation that the action $return(p; b)$ occurs before the clock ticks 21 days. The meaning of formula (1) is that after a person p borrowed a book b, he should return the book within 21 days. Formula (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 (*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 *return* and in (2), *self* executes *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 *self* is the automated system. However, (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 $\iota:a$ if the actor ι initiates atomic event a, and call $\iota:a$ an *action* (rather than an event). If we do not care who the actor in an action is, we write a instead of $\iota:a$.

1.2. INTERNAL AND EXTERNAL CHOICE AND INTERNAL EVENTS

In order to explain the difference between internal and external events, Milner [38] sketches an intuitive picture of black box M equipped with buttons that can be pushed by an observer O. Buttons may or may not be blocked, depending upon the internal state of M. If O tries to push a button, he makes an observation, viz. that the button is blocked or that the button can be pushed.

If M has two buttons a and b that are unblocked, then O is in a position to choose whether to push one or the other button. Using CSP-like notation, the process executed is then $a[\,]b$, where $[\,]$ stands for external choice, i.e. a choice made by O. If, on the other hand, M chooses to block one or the other button, then the process executed is $a \sqcap b$, where $\sqcap$ stands for internal choice, i.e. a choice made by M. The difference between the two processes is that in $a \sqcap b$, O just finds that he can push one of the two buttons and that the other is blocked, whereas in $a[\,]b$, O has the freedom to choose between pushing either button.

This vivid example can be generalized to the case of n actors for any $n > 1$, by allowing any actor to make a choice. The distinction between internal and external then loses its meaning, for we will not identify with any actor in the system. Thus, actors allow a generalization of the concept of internal and external choice to the concept of an *active choice*, i.e. a choice labeled by an actor who *makes* the choice. We will write $\iota:(x \oplus y)$ for the active choice made by actor ι.

This seems to reduce the number of different kinds of choices from two (internal and external choice) to one (active choice). However, we need another kind of choice, which we will call *passive choice* or *alternative occurrence*. A passive choice between processes x and y is an underspecification, because it is the process x or the process y, but it is not specified which. We write $+$ for passive choice. Thus, $x + y$ is the process in which x or y occurs, but it is not specified which. Note, incidentally, that $+$ is not really a choice but absence of choice on the part of the specifier.

Active and passive choice differ, as can be seen from the axioms that we could set up for them. For example, $(x + y) + z = x + (y + z)$, but active choice is not associative. The order of choices made by ι in $\iota:(\iota:(x \oplus y) \oplus z)$ differs from that in $\iota:(x \oplus \iota:(y \oplus z)$, and it is not a priori the case that these two processes are equal. If the choices are made by different actors, equality is even less obvious.

Note that + is not CSP's internal choice [ ]. We would write $x[\,]y$ as $M:(x \oplus y)$. Our approach makes clear that in CSP, internal choice [ ] is overloaded with two meanings, underspecification and internal choice by the machine. We distinguish these as passive and active choices.

CCS and ACP [4] use the idea that a choice is made by performing a first event of one of the branches. They use τ to stand for 0 or more events that are initiated by M and invisible to O. Using these two conventions, $\tau x + \tau y$ represents a choice made by M. We would represent $\tau x + \tau y$ by $M:(x \oplus y)$. As De Nicola and Hennessy [39] note, this approach of using τ gives problems with the interpretation of $x + \tau y$ if x does not begin with a τ, because it does not give a clue about who makes the choice. This ambiguity does not exist in our approach. We would represent $x + \tau y$ by $x + M:a;y$ for a machine action a. This is a passive choice in which one branch is known to start with an action initiated by M. This also shows that CCS and ACP choice + is overloaded with active and passive meanings.

The idea of making a choice by performing a first event of one of the branches can be explicated by introducing atomic actions $\underline{\iota : (x \leftarrow\!\oplus\ y)}$ and $\underline{\iota : (x \oplus\!\rightarrow y)}$, which represent the event that ι chooses the left branch and the right branch, respectively. We call these actions *internal choices*. They can be made invisible in a separate abstraction step. Naming them explicitly allows us to solve a paradox of deontic logic, discussed below.

Note that τ combines the ideas of *initiative* by the machine and *invisibility* to O. We separate these two ideas and explicitly write the initiator of every event in front of the event. An action initiated, i.e. performed, by actor ι is obviously visible to ι, but the converse is not true in general. We are not concerned with visibility of events and will not represent which event is visible to which actor. (Doing this would probably lead to different equality relations on processes, one for every actor/observer.)

1.3. THE PARADOX OF FREE CHOICE PERMISSION

Traditionally, deontic logic has been plagued by numerous paradoxes. Castañeda [8] and Von Wright [52] have proposed that a number of these paradoxes can be resolved by distinguishing actions from states. The definition of deontic logic in dynamic logic we use is one formalization of this approach. One paradox still remains, however, called the *paradox of free choice permission* described, among others, by Hilpinen [21] and Kamp [26]. This is that the following formula is derivable (P(a) says that event a is permitted):

$$\mathbf{P}(\textit{buy chewing gum}) \rightarrow \mathbf{P}(\textit{buy chewing gum or kill the queen}). \qquad (3)$$

This is counterintuitive if we interpret the right-hand side as saying that you are permitted to choose between *buy chewing gum* and *kill the queen*. Meyer [33] observed that we can get out of this paradox if we use the CSP distinction between

internal and external choice. We now show that this idea can be generalized by using the distinction between active and passive choice.

First, we must remark that in our formalization of deontic logic, an action **a** is permitted if it may lead to a permitted state of the world. In general, an action a is nondeterministic and may lead to one out of a set of possible states of the world. If at least one of these states is permitted, then there is a way of doing **a** that is permitted, and we say that **a** itself is permitted. Thus, we should read "**a** may be beneficial" for **P(a)**.

Second, connecting this with the two kinds of choices, note that the nondeterminism of an action **a** is of the underspecification kind. **a** reprsents a passive choice among a set of state transitions, which have in common only that they contain the action **a**. Thus, we must read for **a** "**a** occurs, and possibly other things as well".

Connecting these two remarks, we think the formula

$$\mathbf{P(a)} \rightarrow \mathbf{P(a + b)} \tag{4}$$

is valid. **a** and **b** are actions in which we do not care who the actors are. **a + b** is a transition to a next world, such that the transition contains an occurrence of **a** or of **b** (or both). Now, if among the transitions represented by **a** there is one that leads to a permitted state of the world, then among the transitions represented by **a + b** there is one that leads to a permitted state of the world. If **a** may be beneficial, then **a + b** may be beneficial.

The formula is invalid if we read *active* choice in it. Thus, we want

$$\mathbf{P(a)} \rightarrow \mathbf{P}(\iota{:}(\mathbf{a} \oplus \mathbf{b})) \tag{5}$$

to be invalid. From the fact that **a** is permitted, it should not follow that any actor is permitted to choose between doing **a** or something else. $\iota{:}(\iota{:}a \oplus \iota{:}b)$ is called a *free choice* in deontic logic, and $\mathbf{P}(\iota{:}(\iota{:}a \oplus \iota{:}b))$ is called *free choice permission*. We will call $\iota_0{:}(\iota_1{:}a \oplus \iota_2{:}b)$ *imposed choice* if ι_0 differs from at least one of ι_1 and ι_2 and we call $\mathbf{P}(\iota_0{:}(\iota_1{:}a \oplus \iota_2{:}b))$ *imposed choice permission*.

We will show that the concept of actor and the distinction between active and passive choice allows us to eliminate the paradox of free choice permission in several ways. To illustrate this, we present one system in which (4) is a theorem and (5) is not. In fact, in that system, we have the theorems

$$\mathbf{P(a + b)} \leftrightarrow \mathbf{P(a)} \vee \mathbf{P(b)}, \tag{6}$$

$$\mathbf{P}(\iota{:}(\iota{:}a \oplus \iota{:}b)) \rightarrow \mathbf{P}(\iota{:}a) \wedge \mathbf{P}(\iota{:}b). \tag{7}$$

Formula (7) blocks the paradox of free choice permission. There is no such theorem as (7) in the case that at least two of the three actors in an active choice are different.

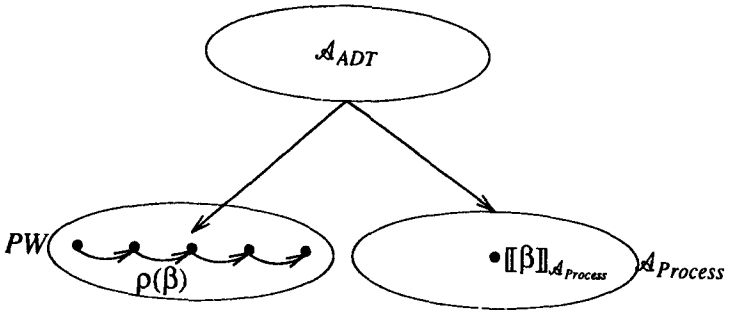

Fig. 1. Overall structure of the model.

We also present another system, using the internal choices $\iota : (x \leftarrow\!\oplus\ y)$ and $\iota : (x \oplus\!\rightarrow y)$, in which neither (4) nor (5) is a theorem, but in which the conditions under which we grant an actor permission to make a choice can be specified precisely. This system has more axioms than the first one, but has a simpler semantics and agrees more closely with our intuition. These two systems are not the only axiomatizations of free choice permission, but at least they suffice to show that the paradox of free choice permission can be eliminated by using the concept of an actor.

1.4. PLAN OF THE PAPER

The paper has the following structure. We want to be able to specify a system as a set of possible states which all contain an underlying abstract data type (ADT) as reduct. Thus, for example, if the natural numbers are part of the underlying ADT, then 1 + 1 = 2 is valid in all possible states of the system. Events and processes can take one from one state to another state, but will leave the validity of equations in the underlying ADT intact. Events and processes will be specified as functions on the set of possible states. Thus, the system is a Kripke structure with multiple accessibility relationships, one for each event and process.

In more detail, we model any system as shown in fig. 1. Each world in the set *PW* of possible worlds contains the underlying ADT, called $\mathcal{A}_{ADT}$, as reduct. In addition, there is an algebra $\mathcal{A}_{Process}$ of processes, of which the elements are events and processes, and the operators are choices, sequence, synchronous execution, etc. The process algebra also contains the underlying ADT as reduct. Finally, to interpret events and processes as functions on possible worlds, there is a function ρ which, for each event and process, yields a function on *PW* that states the effect of the event or process on the possible worlds.

In section 2, we rehearse some definitions relevant to equational specification of ADTs and treat two kinds of equational specifications that we need, the equational specification of abstract data types (ADTs) and the equational specification of process algebras. ADT specifications are needed for, among others, the specification

of actor identifiers, and process algebras are needed to axiomatize the concepts of active and passive choice. In addition, the concept of action negation is axiomatized, which is needed to define deontic operators later on.

In section 3, we extend equational logic to full first-order logic with equality, and show how to specify static integrity constraints in it. This is a brief section, but some fine points concerning the semantics of static constraint specifications are treated.

Section 4 extends first-order logic with equality to a particular brand of dynamic logic with equality. Dynamic logic is a logic for reasoning about post-conditions of terminating processes, and the processes reasoned about are the processes defined in the process algebra specification given in section 2. A sound and complete inference system for this system of dynamic logic is given. Furthermore, section 4 defines what dynamic constraint specifications are.

Section 5 extends dynamic logic to deontic logic. This again is a brief section in which we introduce only the ideas necessary for our purpose.

Throughout the paper, we give a specification of a library as example specification. For convenience, this is given in the appendix. In section 6, we give two models of this library specification, both of which solve the paradox of free choice permission. The structure of both models is shown in fig. 1, but they differ in that the second model uses the atomic actions $\underline{\iota : (x \leftarrow\!\oplus\ y)}$ and $\underline{\iota : (x \oplus\!\rightarrow y)}$ to represent internal choices.

Finally, section 7 contains a discussion of the relations between nondeterminism and initiative as formalized in CSP, CCS and in this paper, and of a possible extension of this work to the concept of active object in object-oriented specification. Section 8 summarizes and concludes the paper.

2. Specification in equational logic

In any system specification, we will need some abstract data types (ADTs) such as natural numbers, Booleans or strings. We simply assume these given, but point out that we will specify actors as an ADT as well. First, we rehearse some definitions relevant to algebraic specification of ADTs, and then we give some examples of specifications of actor ADTs. We use order-sorted algebraic specification [11,19,44].

DEFINITION 1 (ORDER-SORTED SIGNATURE)

An *order-sorted signature* is a triple $((S, \leq), \mathbb{F}, \mathbb{P})$, where $(S, \leq)$ a poset of sort names, $\mathbb{F}$ a set of function declarations over S of the form $f : s_1 \times \ldots s_n \rightarrow s_0$ for $s_0, \ldots, s_n \in S$, and $\mathbb{P}$ a set of predicate declarations over S of the form $P : s_1 \times \ldots \times s_n$. If $\mathbb{P} = \emptyset$, the signature is called *equational*.

Although it would be more accurate to call a signature without predicate declarations *functional*, we call it equational because it is the kind of signature used in equational specifications.

For each sort $s \in S$ of a signature, we assume an infinite set X_s of *variables* of sort s. All X_s for all $s \in S$ are assumed to be mutually disjoint. So $X_{s_1} \cap X_{s_2} = \emptyset$ even if $s_1 \leq s_2$. The set of all variables is $X = \cup_{s \in S} X_s$.

A *Sig*-term of sort s is either a variable of sort $s' \leq s$ or a term of the form $f(t_1, \ldots, t_n)$, where f is declared as $f: s_1 \times \ldots s_n \rightarrow s_0$ with $s_0 \leq s$ and the sort of t_i is less than or equal to s_i, $i = 1, \ldots, n$ [19,44]. $T(Sig)_s(X)$ is the set of *Sig*-terms containing variables from X. The set of closed *Sig*-terms of sort s is $T(Sig)_s$. We omit *Sig* if the signature is understood or irrelevant. Goguen and Meseguer [19] show that under certain weak conditions, called *regularity*, the terms of an equational signature always have a unique least sort. In the definitions, we assume that all specifications satisfy these conditions.

DEFINITION 2 (EQUATIONAL LANGUAGE)

Let $Sig = ((S, \leq), \mathbb{F})$ be an equational signature.

- A *connected component* of S is an equivalence class with respect to the transitive symmetric closure of $\leq$.
- Two sort names in S are *compatible* if they are in the same connected component.
- A *Sig-equation* $(X, t_1 = t_2)$ consists of a pair of terms t_1, t_2 over Sig_{Eq} whose least sorts of t_1 and t_2 are compatible, and a set X of variable declarations such that all variables in t_1 and t_2 are declared in X.
- A *conditional* equation has the form $(X, t_{l_1} = t_{r_1} \wedge \ldots \wedge t_{l_k} = t_{r_k} \rightarrow t_{l_0} = t_{r_0})$, with $(X, t_{l_i} = t_{r_i})$ a *Sig*-equation.
- The *equational language* $L_{Eq}(Sig)$ is the set of all conditional equations over *Sig*.

Declarations are needed to be able to define inference rules that are sound and complete even if empty sorts are allowed [18,9]. For brevity, we gather the declaration together in one place in the examples. There is a relation $\vdash_{Eq}$ that defines *equational deduction* and a relation $\models_{Eq}$ that defines truth of a conditional equation in a structure. We do not go into detail about these relations here; definitions can be found elsewhere [17,19,44] and in section 4.1, we give inference rules and a truth definition for dynamic logic with equality, which includes equational logic. Whichever definition is used, we assume here that $\vdash_{Eq} \Rightarrow \models_{Eq}$.

DEFINITION 3 (EQUATIONAL SPECIFICATION)

Let Sig_{Eq} be an equational signature. A Sig_{Eq}-specification $Spec_{Eq}$ is a pair (Sig_{Eq}, E) where E is a set of conditional equations over Sig_{Eq}.

2.1. EQUATIONAL ABSTRACT DATA SPECIFICATION

Abstract data types (ADTs) can be specified equationally [11, 12, 19, 44]. Our intended semantics of equational ADT specifications is the *initial algebra semantics* explained in the references above. Basically, in this semantics, a data element is a closed term modulo equality. This means that each data element of an ADT is named by at least one closed term, and that two closed terms denote equal data elements iff they can be proved equal from the specification, using the inference rules of equational logic. Goguen and Meseguer [19] call this the **no junk** and **no confusion** properties of the initial algebra semantics. Junk is unnameable data elements, and the more dignified term *mystical elements* may also be used.

Our running example contains the following ADT specifications.

value spec *PersonIdentifiers*
  import
    Booleans
  sorts
    PERSON
  functions
    $p_0 : PERSON$
    $next : PERSON \rightarrow PERSON$
    $eq : PERSON \times PERSON \rightarrow BOOL$
  variables
    $x, x_1, x_2 : PERSON$
  equations
[1] $x\ eq\ x = true$
[2] $p_0\ eq\ next(x) = false$
[3] $next(x)\ eq\ p_0 = false$
[4] $next(x_1)\ eq\ next(x_2) = x_1\ eq\ x_2$
end spec *PersonIdentifiers*

We assume that our example contains an ADT specification *Booleans* which declares sort of interest (SOI) *BOOL*, for which the usual Boolean operators are defined. *PersonIdentifiers* then adds a sort *PERSON*, which is populated with closed terms of the form $next^n(p_0)$ for $n \in \mathbb{N}$. The elements of *PERSON* are called *person identifiers.*

Without giving them, we assume a number of ADT specifications. The specification *PersonQueues* has SOI *PERSON_QUEUE* of queues of person identifiers. The *eq* function for book identifiers is needed in the *PersonQueues* specification. We also assume a specification *BookIdentifiers* with SOI *BOOK* of book identifiers, a specification *Money* with SOI *MONEY*, and a specification *LibraryIdentifiers* with

SOI *LIBRARY*. This last sort contains only one element, l_0, identifying the library we are interested in.

To specify a system, we assume that there is an ADT specification containing a distinguished sort of *actor identifiers*, called the *actor sort* of the ADT specification. An actor identifier is just a data element of a sort that will be used in a certain way in the rest of the system specification, as illustrated below. We will use ι, κ as metavariables for terms of the actor sort of a specification. In our running example, we declare the actor identifier sort *ACTOR* as follows.

value spec *ActorIdentifiers*
  import
    PersonIdentifiers, LibraryIdentifiers
  sorts
    PERSON $\leq$ *ACTOR*
    LIBRARY $\leq$ *ACTOR*
end spec *ActorIdentifiers*

These will be the data elements that identify actors in the system. Terms ι, κ of the sort *ACTOR* will receive special treatment in the process algebra.

In general, the symbol $\leq$ used in the example specification is not the $\leq$ of the signature determined by the specification. The above specification determines a signature in which $(\{PERSON, LIBRARY, ACTOR\}), \leq)$ is a poset of sort names. The partial ordering $\leq$ in this poset is the reflexive transitive closure of the relation $\leq$ declared above. Thus, in the poset we have, for example, $PERSON \leq PERSON$, but this is not declared in the specification.

In what follows, the concept of a conservative extension of an ADT specification will be essential. In the next definition, we write $(S_0, \leq_0) \subseteq (S_1, \leq_1)$ for $S_0 \subseteq S_1$ and $\leq_0 \subseteq \leq_1$.

DEFINITION 4 (EXTENSION)

Let $Spec_0 = ((S_0, \leq_0), \mathbb{F}_0, \mathbb{P}_0, E_0)$ and $Spec_1 = ((S_1, \leq_1), \mathbb{F}_1, \mathbb{P}_1, E_1)$ be two equational specifications.

- $Spec_1 = ((S_1, \leq_1), \mathbb{F}_1, \mathbb{P}_1, E_1)$ is an *extension* of $Spec_0$ iff $(S_0, \leq_0) \subseteq (S_1, \leq_1)$, $\mathbb{F}_0 \subseteq \mathbb{F}_1$, $\mathbb{P}_0 \subseteq \mathbb{P}_1$, and $E_0 \subseteq E_1$.
- $Spec_1$ is an *enrichment* of $Spec_0$ iff it is an extension with $(S_0, \leq_0) = (S_1, \leq_1)$.

DEFINITION 5 (CONSERVATIVE EXTENSION)

Let $Spec_1$ be an extension of $Spec_0 = ((S_0, \leq_0), \mathbb{F}_0, \mathbb{P}_0, E_0)$.

(1) $Spec_1$ is a *complete* extension of $Spec_0$ iff for any $s \in S_0$ and any $t \in T(Spec_1)_s$ there is a $t' \in T(Spec_0)_s$ such that $Spec_1 \vdash_{Eq} t = t'$.

(2) $Spec_1$ is a *consistent* extension of $Spec_0$ iff for any $s \in S_0$ and any $t_1, t_2 \in T(Spec_0)_s$ we have $Spec_1 \vdash_{Eq} t_1 = t_2$ iff $Spec_0 \vdash_{Eq} t_1 = t_2$.

(3) $Spec_1$ is a *conservative* extension of $Spec_0$ iff it is a complete and consistent extension of $Spec_0$.

In this definition, we follow the definition of conservativeness given by Ehrig and Mahr [12, p. 153]. Note that "conservative extension" may also be used in the meaning of "complete extension" as defined above. Thus, a complete extension of $Spec_0$ adds no data elements to the ADT specified by $Spec_0$, and a consistent extension identifies no more data elements than were already identified by $Spec_0$. Jointly, these two properties defined conservative extensions. If we choose the initial semantics of the extended specification $Spec_0$, then any conservative extension of $Spec_0$ preserves the no junk and no confusion properties of the initial semantics of $Spec_0$.

2.2. EQUATIONAL PROCESS ALGEBRA SPECIFICATION

The behavior of a system will be specified in process algebra [4–7]. It is not customary to give a purely equational specification of this, so we give one in this subsection. New elements in this specification are the inclusion of an underlying ADT specification, the presence of actors, and an axiomatization of action negation. First, we define the relation between a process specification and the underlying ADT specification.

DEFINITION 6 (PROCESS SIGNATURE)

Let Sig_{ADT} be an ADT signature. A *process signature* $Sig_{Process}$ over Sig_{ADT} is an extension of Sig_{ADT} with declarations of sort and function names. One sort added in the process specification will be called the *process sort.* Terms of that sort are called *process terms.*

DEFINITION 7 (PROCESS SPECIFICATION)

Let $Spec_{ADT}$ be an ADT specification. A *process specification* $Spec_{Process}$ over $Spec_{ADT}$ is a conservative extension of $Spec_{ADT}$ whose signature is a process signature over the signature of $Spec_{ADT}$.

Any conservative extension of an ADT specification, with a distinguished sort called the sort of processes, is a process specification. This is because at the level of equational specification, there is nothing special about the fact that a specification is a process specification rather than a specification of natural numbers or stacks. Calling a conservative extension of $Spec_{ADT}$ a *process* specification is an expression of the intention of the specifier and is not something visible from the syntax of the specification alone.

Note that for us, a *process specification* is the same thing as a *process theory*. We will use the two terms as synonymous. A process specification is for us a set of axioms for processes, and not a process *definition*, i.e. a set of equations that defines a single process.

There are many possible models of process specifications, such as process graphs, event structures, or Petri nets [4,16], each with many different equality relations between processes. These models differ in the way they model concurrency, nondeterminism, and in general in their discriminating power between processes. We allow the specifier to use his or her preferred intended process model but require that, whichever model is chosen, the process specification is a *conservative* extension of the ADT specification. Thus, no data elements must be added to the ADT sorts, and no data elements in those sorts must be identified that were not already identified in the ADT specification. This preserves the no junk and no confusion properties of $Spec_{ADT}$.

We now show an example extension of an underlying ADT specification to a process specification and give an intended semantics of this specification afterwards. Our example process specification is given in three parts: a specification of atomic events, of single-step processes, and of processes containing the sequence operator.

process spec *LibraryEvents*
  import
    PersonIdentifiers, BookIdentifiers, LibraryIdentifiers, Money
  sorts
    PERSON_EVENT
    LIBRARY_EVENT
  functions
    borrow : $BOOK \rightarrow PERSON\_EVENT$
    return : $BOOK \rightarrow PERSON\_EVENT$
    reserve : $BOOK \rightarrow PERSON\_EVENT$
    pay : $MONEY \rightarrow PERSON\_EVENT$
    notify : $PERSON \times BOOK \rightarrow LIBRARY\_EVENT$
end spec *LibraryEvents*

Terms of the sort *PERSON_EVENT* and *LIBRARY_EVENT* are called *atomic events*. We use a, b as metavariables over atomic *events*, i.e. as variables over T_{PERSON_EVENT} and $T_{LIBRARY\_EVENT}$.

Because $Spec_{ADT}$ is imported in $Spec_{Process}$, we have that $pay(\$2) = pay(\$1 + \$1)$ exactly when $\$2 = \$1 + \$1$ in the underlying ADT specification. If the process specification would not be a consistent extension of $Spec_{ADT}$, then we could have $pay(\$2) = pay(\$3)$, even though $\$2 \neq \$3$ in $Spec_{ADT}$, which is undesirable. If $Spec_{Process}$ would not be a complete extension of $Spec_{ADT}$, then the process specification could

introduce new data elements. We think that would be undesirable, and that all data elements should be declared in $Spec_{ADT}$. Further discussion of the use of conservative extensions follows in section 3.

The next specification adds non-atomic events. All events, atomic and non-atomic, need actors to occur. An *action* is an event initiated by an actor. Just like we have atomic and non-atomic events, we have atomic and non-atomic actions.

process spec *Actions*

import

LibraryActions, LibraryActors

sorts

$PERSON\_EVENT \leq EVENT$

$LIBRARY\_EVENT \leq EVENT$

$CHOICE \leq EVENT$

$ATOMIC\_ACTION \leq ACTION$

functions

$any : EVENT$

$fail : EVENT$

$\_\oplus\_ : ACTION \times ACTION \rightarrow CHOICE$

$\_:\_ : PERSON \times PERSON\_EVENT \rightarrow ATOMIC\_ACTION$

$\_:\_ : LIBRARY \times LIBRARY\_EVENT \rightarrow ATOMIC\_ACTION$

$\_:\_ : ACTOR \times EVENT \rightarrow ACTION$

$\_+\_ : ACTION \times ACTION \rightarrow ACTION$

$\_\,\&\,\_ : ACTION \times ACTION \rightarrow ACTION$

$-\_ : ACTION \rightarrow ACTION$

variables

$\iota, \iota_0, \iota_1, \iota_2 : ACTOR$

$\alpha : EVENT$

$\alpha, \alpha_1, \alpha_2, \alpha_3 : ACTION$

equations

[PC1] $\alpha_1 + \alpha_2 = \alpha_2 + \alpha_1$

[PC2] $(\alpha_1 + \alpha_2) + \alpha_3 = \alpha_1 + (\alpha_2 + \alpha_3)$

[PC3] $\alpha + \alpha = \alpha$

[N1] $- - \alpha = \alpha$

[N2] $-(\alpha_1 + \alpha_2) = -\alpha_1 \,\&\, -\alpha_2$

[N3] $-(\alpha_1 \,\&\, \alpha_2) = -\alpha_1 + -\alpha_2$

[D] $\alpha \,\&\, (\alpha_1 + \alpha_2) = (\alpha \,\&\, \alpha_1) + (\alpha \,\&\, \alpha_2)$

[ANY1] $\iota : \alpha + \iota : any = \iota : any$
[ANY2] $\iota : \alpha \,\&\, \iota : any = \iota : \alpha$
[AC1] $\iota : (\alpha_1 \oplus \alpha_2) = \iota : (\alpha_2 \oplus \alpha_1)$
end spec *Actions*

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. Just as we use a, b as metavariable over atomic events, we use **a**, **b** as metavariable over atomic *actions*, i.e. closed terms of sort ATOMIC_ACTION. All atomic actions have the form $\iota : a$ for an actor ι and an atomic event a. The intuitive interpretation of $\iota : a$ is that ι does (initiates, performs) a. For example, the term $l : notify(p, b)$ stands for the action of the library l notifying p that book b is overdue.

We use α as metavariable over *EVENT* terms and α as metavariable over *ACTION* terms. By **any** and **fail**, we mean *any* and *fail* events initiated by any actor. So **any** is $\iota : any$ for any ι.

An event is a transition to a next state that needs an actor to be performed. An event α may be

- an atomic event a (*PERSON_EVENT* or *LIBRARY_EVENT*),
- *any* or *fail*,
- or a choice $\oplus$ between actions.

An event cannot occur on its own but must be initiated by an actor. An event initiated by an actor is called an *action*. Apart from actions of the form $\iota : \alpha$, there are actions composed of more elementary actions by alternative occurrence (+), synchronous occurrence (&), and action negation (–). We now explain the axioms in the specification.

(1) Passive choice has the usual properties of choice in process algebras: it is commutative, associative, and idempotent. It can be proven in equational logic from the above axioms that & has the same properties as passive choice, i.e. it is commutative, associative, and idempotent. Synchronization also distributes over choice, as is usual in process algebra.

(2) $-\alpha$ means that it is not the case that α occurs. Action negation is required to be a Boolean algebra with passive choice and synchronous occurrence. If we assume [N1], then [N2] and [N3] are equivalent.

(3) [ANY1] defines $\iota : any$ as the local zero of additions for ι. It enforces the interpretation on **any** that it is a passive choice over any action. Saying that ι does anything or α thus gives no extra information over the statement that ι does anything.

(4) [ANY2] defines $\iota : any$ as the local unit of multiplication for ι. It enforces the interpretation on $\iota : \alpha$ in which to say that ι does α is synonymous with the

statement that ι does α, and possibly anything besides as well. This interpretation is convenient for the interpretation of action negation, as we will see, but it is not necessary. It could be omitted from the system if necessary.

(5) We finally add a minimal axiom for active choice, viz. that it is commutative. This much is uncontroversial about active choice. We will in a moment see that other axioms, which seem reasonable at first sight, are questionable.

We now discuss the axioms we considered, but did not include in the example specification, for *fail*, active choice, and action negation.

Failure. It would have been neat to include the axioms

$$\iota : \alpha + \iota : \mathit{fail} = \iota : \alpha \tag{8}$$

$$\iota : \alpha \,\&\, \iota : \mathit{fail} = \iota : \mathit{fail} \tag{9}$$

to complement `[ANY1]` and `[ANY2]`. Axiom (8) defines $\iota : \mathit{fail}$ to be the zero of addition. It is one of the standard deadlock axioms of process algebra, viz. when given the choice to fail or do something, an actor will always do something. However, since we use *passive* choice in (8), the axiom should be read as

> the statement that ι fails or does α, but we do not say which,
> is synonymous with the statement that ι does α.

This is valid only if $\iota : \alpha$ is a choice over a set of options that includes $\iota : \mathit{fail}$ and that is highly questionable. In the models we give later on, it is false. We therefore omitted the axiom from the specification.

Formula (9) defines $\iota : \mathit{fail}$ to be the local zero of parallel composition. It says that when ι fails, it cannot do anything besides fail. This seems reasonable, but it is false if we interpret & as the synchronous occurrence of two actions and allow different actors to perform actions independently of each other. The right-hand side of (9), $\iota : \mathit{fail}$, expresses that ι fails but the rest of the world can still continue performing actions. The left-hand side of (9), $\iota : \alpha \,\&\, \iota : \mathit{fail}$, denotes a situation in which ι fails to do anything *and* does α. However, it is inconsistent to say that ι does something and does nothing at the same time. So $\iota : \alpha \,\&\, \iota : \mathit{fail}$ specifies some kind of impossible action, which would be identified with the failure event $\iota : \mathit{fail}$. It is possible to add this axiom, but since we have different intuitions about what the left- and right-hand side mean, intuitively, we chose to be careful and omitted the axiom.

Active choice. Looking for other axioms for active choice, we may try idempotence,

$$\iota : (\kappa : \alpha \oplus \kappa : \alpha) = \kappa : \alpha.$$

As it stands, this is questionable, for on the left-hand side it is ι who does something, whereas on the right-hand side it is κ who does something. However,

$$\iota : (\iota : \alpha \oplus \iota : \alpha) = \iota : \alpha \tag{10}$$

seems reasonable and we could add it. The axiom is valid in our first example model below, because we there treat the left-hand side of the equation as an atomic action that turns out to have the same effect as the action on the right-hand side. The axiom is not valid in the second model we give, because there we make the choice explicit and the left-hand side is not an atomic action anymore (even though the net effect of the left-hand side equals the effect of the action on the right-hand side). We therefore omitted it from the example specification.

Associativity of active choice is not valid, in general, for take the equation

$$\iota : (\kappa : (\alpha_1 \oplus \alpha_2) \oplus \alpha_3) = \kappa : (\alpha_1 \oplus \iota : (\alpha_2 \oplus \alpha_3)).$$

This is not a priori true intuitively, because ι and κ make completely different choices on the left-hand side and on the right-hand side. In addition, they make their choices in a different order on the left- and right-hande sides. This is also true in

$$\iota : (\iota : (\alpha_1 \oplus \alpha_2) \oplus \alpha_3) = \iota : (\alpha_1 \oplus \iota : (\alpha_2 \oplus \alpha_3)).$$

By omitting associativity, actors can only make binary choices in our setting. Given the current axioms, an actor can only make a choice between n alternatives, $n > 2$, by making a sequence of binary choices, and these sequences are not equivalent. What is needed here is an n-ary choice for $n \geq 0$. In the case of $n = 0$, we have failure, and in the case of $n > 0$, we have an active choice between n options. However, such a choice is not axiomatizable in first-order equational logic.

It is possible to reduce active to passive choice by adding the axiom

$$\iota_0 : (\alpha_1 \oplus \alpha_2) = \iota_0 : \tau^1; \alpha_1 + \iota_0 : \tau^1; \alpha_2, \tag{11}$$

where τ^1 is a single invisible action performed by ι_0, and ; the sequence operator axiomatized below. An active choice would then be a term of the sort *PROCESS* declared below and not of sort *ACTION*. This is similar to the CCS interpretation of internal choice, which says that a choice is made by doing the first event of one of the branches. However, it is not the same as CCS internal choice, for τ^1 is one invisible action initiated by ι_0, and not zero, one or more invisible actions initiated by a machine M, as it is in CCS.

If we add axiom (11), then idempotence of active choice for a single actor (10) is false, because we would have

$$\iota : (\iota : \alpha \oplus \iota : \alpha_2) = \iota : \tau^1; \iota : \alpha$$

and this cannot be reduced to $\iota : \alpha$.

$\iota : \tau^1$ expresses the fact that ι performs one action, but it is not said which. A refinement of this is to name the action explicitly. In $\iota_0:(\alpha_1 \oplus \alpha_2)$, ι_0 actively chooses between α_1 and α_2. This active choice can be modeled as a passive choice between two actions performed by ι_0. One action is

$$\underline{\iota_0:(\alpha_1 \leftarrow\!\oplus\ \alpha_2)},$$

pronounced "ι_0 chooses α_1 out of the possibilities α_1 and α_2", and one is

$$\underline{\iota_0:(\alpha_1 \oplus\!\rightarrow\ \alpha_2)},$$

in which ι_0 chooses α_2 out of the possibilities α_1 and α_2. As said before, these actions are called *internal choices* and they are atomic actions. ι_0 can perform these internal choices irrespective of who are the actors in α_1 and α_2, i.e. they may be equal to or different from ι_0. For every triple ι_0, α_1 and α_2, there are such atomic actions. Using internal choices, we can then define active choice in terms of passive choice by

$$\iota_0:(\alpha_1 \oplus \alpha_2) = \underline{\iota_0:(\alpha_1 \leftarrow\!\oplus\ \alpha_2)};\alpha_1 + \underline{\iota_0:(\alpha_1 \oplus\!\rightarrow\ \alpha_2)};\alpha_2. \qquad (12)$$

This definition has the advantage that it requires no extra machinery beyond the well-understood passive choice to interpret and that it makes clear why $\oplus$ lacks a number of properties that $+$ has. In addition, it will allow us to specify precisely where the problem lies with the paradox of free choice permission, as we will see in section 6.2. However, it does not solve the problem that active choice should really be n-ary rather than binary.

Action negation. $-\alpha$ is the same kind of underspecification as α_1 and α_2. Thus, $-\alpha$ means "α does not occur, and it is not specified what does occur". This is similar to the meaning of negation in propositional logic, and this meaning explains why negation forms a Boolean algebra with passive choice and synchronization. We call this interpretation of action negation *passive action negation.*

There is another meaning of action negation, which we call *active action negation,* in which $-\iota : \alpha$ means "ι does not initiate α". The difference with α is that we now explicitly mention the actor who does not do something. This can be interpreted in a *local* way as the statement that ι does something other than α, or in a global way as the statement that another actor than ι does α. A passive disjunction between these two interpretations is also possible. The local interpretation of active negation is enforced by the axiom

$$\iota : \alpha + -\iota : \alpha = \iota : any.$$

The global interpretation of action negation cannot be axiomatized in the example specification, for it requires a constant that denotes the process "any actor does something".

Note that a seemingly plausible axiom like

$$-\iota : fail = \iota : any$$

is only reasonable if we assume local active negation. It defines negation to be a complement operator.

We now turn to the specification of processes that contain a sequence operator. The sort *PROCESS* below is the process sort of our process specification.

process spec *Processes*
import
Actions
sorts
$ACTION \leq PROCESS$
functions
$\_;\_ : PROCESS \times PROCESS \rightarrow PROCESS$
$\_+\_ : PROCESS \times PROCESS \rightarrow PROCESS$
$\_\&\_ : PROCESS \times PROCESS \rightarrow PROCESS$
$\_\oplus\_ : PROCESS \times PROCESS \rightarrow CHOICE$
variables
$\iota : ACTOR$
$\alpha : ACTION$
$\beta, \beta_1, \beta_2, \beta_3 : PROCESS$
equations

[S1] $(\beta_1; \beta_2); \beta_3 = \beta_1; (\beta_2; \beta_3)$

[PC4] $\beta_1 + \beta_2 = \beta_2 + \beta_1$
[PC5] $(\beta_1 + \beta_2) + \beta_3 = \beta_1 + (\beta_2 + \beta_3)$
[PC6] $\beta + \beta = \beta$

[SYN1] $(\alpha_1; \beta_1) \,\&\, (\alpha_2; \beta_2) = (\alpha_1 \,\&\, \alpha_2); (\beta_1 \,\&\, \beta_2)$
[SYN2] $\alpha_1 \,\&\, (\alpha_2; \beta_2) = (\alpha_1 \,\&\, \alpha_2); \beta_2$
[SYN3] $(\alpha_1; \beta_1) \,\&\, \alpha_2 = (\alpha_1 \,\&\, \alpha_2); \beta_1$

[D1] $\beta_1; \beta_3 + \beta_2; \beta_3 = (\beta_1 + \beta_2); \beta_3$
[D2] $\beta_3; \beta_1 + \beta_3; \beta_1 = \beta_3; (\beta_1 + \beta_2)$
[D3] $\iota : (\beta_1; \beta_3 \oplus \beta_2; \beta_3) = \iota : (\beta_1 \oplus \beta_2); \beta_3$

[S2] $\iota : (\beta_1 \oplus \beta_2) = \iota : (\beta_2 \oplus \beta_1)$

[FAIL1] $\iota : fail; \iota : \beta = \iota : fail$
[FAIL2] $\iota : fail; \beta_1; i : \beta_2 = \iota : fail; \beta_1$

[AC2] $\iota : (\beta_1 \oplus \beta_2) = \iota : (\beta_2 \oplus \beta_1)$

end spec *Processes*

We use α as metavariable over terms of sort *ACTION* and β as metavariable over terms of sort *PROCESS*. We use the convention that : binds stronger than ;. *PROCESS* is the process sort of our example process theory, as mentioned in definition 6.

Remarks

(1) [Syn1-3] define synchronization between processes as the synchronous performance of each of their steps. When the shortest process runs out of steps to do, the other continues. This clearly differs from interleaving. Meyer and De Vink [36] call this *synchronous start semantics.*

(2) [D2] defines determinism for passive choice. This motivated by the observation that the statement

β_3 occurs, followed by β_1 or β_2, but it is not specified which

gives the same information as

β_3 occurs, followed by β_1, or β_3 occurs followed by β_2, but it is not specified which.

If we define active choice in terms of passive choice, as in (12), then we should drop the declaration of $\oplus$ from the specification *Actions*, because it then contains a sequence. [D1] and [D3] then imply

$$\begin{aligned} \iota : (\beta_1; \beta_3 \oplus \beta_2; \beta_3) &= \iota : (\beta_1 \oplus \beta_2); \beta_3 \\ &= \underline{(\iota : (\beta_1 \leftarrow\!\oplus\ \beta_2); \beta_1} + \underline{\iota : (\beta_1 \oplus\!\rightarrow\ \beta_2); \beta_2}); \beta_3 \\ &= \underline{\iota : (\beta_1 \leftarrow\!\oplus\ \beta_2); \beta_1; \beta_3} + \underline{\iota : (\beta_1 \oplus\!\rightarrow\ \beta_2); \beta_2; \beta_3}, \end{aligned}$$

which agrees with our intuitions.

(3) In the case of active choice, we have the well-known non-equivalence of

$\iota : (\beta_3; \beta_1 \oplus \beta_3; \beta_2)$ "ι chooses between $\beta_3; \beta_1$ and $\beta_3; \beta_2$"

and

$\beta_3; (\iota : (\beta_1 \oplus \beta_2)$ "β_3 occurs, and then ι chooses between β_1 and β_2".

Thus, there is no left-distributivity of active choice. On the other hand, there is right-distributivity, as shown by axiom [D3], which states that ι makes a choice between the first events of two branches and cannot see anything that occurs after that. This could be generalized to actors with n-step lookahead, as in **LR**(n) parsers.

(4) [FAIL1,2] say that if ι deadlocks, ι cannot do anything anymore but other actors can still display initiative as if nothing happened.

In CSP, there is an axiom in which external choice equals an internal choice, viz.

$$O:(O:a;\ \beta_1 \oplus O:a;\ \beta_2) = O:a;\ M:(\beta_1 \oplus \beta_2).$$

O loses initiative to the machine because O cannot look ahead more than one step. A possible formalization of this with actors is

$$\iota : (\beta_1;\ \beta_2 \oplus \beta_1;\ \beta_3) = \beta_1;\ \beta_2 + \beta_1;\ \beta_3,$$

where ι loses initiative, but it is not stated who gets it, if anyone gets it. This would not hold when active choice is defined in terms of passive choice, as in (11) or (12), so we omitted it.

To keep matters simple, we do not allow negation of processes, although this has been formalized in an earlier version of the language [10,34], and can easily be added here.

To keep matters even more simple, we assume the initial semantics of our example process specification. Thus, $\mathcal{A}_{Process}$ in fig. 1 is the initial algebra of *Processes*. Thus, *Processes* is surely a conservative extension of the underlying ADT. Other models could be chosen, but this matter is orthogonal to the problem of initiative upon which we concentrate in this paper.

3. Specification of static constraints in first-order logic

To specify static system properties, we use first-order logic with equality. This is in itself nothing special, but there are some subtleties concerning the import of ADT specifications and concerning the semantics of static system specifications, which we treat in this section. We explain this in the next few paragraphs.

DEFINITION 8 (STATIC CONSTRAINT SIGNATURE)

Let Sig_{ADT} be an ADT signature. A *static constraint signature* Sig_{Stat} is an enrichment of an ADT signature Sig_{ADT} with declarations of function and predicate symbols. Sig_{ADT} is called the *underlying* ADT signature of Sig_{Stat} and Sig_{Stat} is called a static signature *over* Sig_{ADT}.

DEFINITION 9 (STATIC CONSTRAINT LANGUAGE)

The *order-sorted constraint language* $L(Sig_{Stat})$ over a static constraint signature Sig_{Stat} is the set of all first-order formulas that can be built from the signature according to the following BNF:

$$\phi ::= t_1 = t_2 \mid P(t_1, \ldots, t_n) \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \phi \leftrightarrow \psi \mid \forall x(\phi) \mid \exists x(\phi).$$

Note that if Sig_{Stat} is a static constraint signature over Sig_{ADT}, then

$$L_{ADT}(Sig_{ADT}) \subseteq L_{Stat}(Sig_{Stat})$$

and each Sig_{ADT} specification is a Sig_{Stat} constraint specification. $L_{ADT}(Sig_{ADT})$-equations are to be treated as universally quantified formulas in $L_{Stat}(Sig_{Stat})$.

There are definitions of an inference relation $\vdash_{Stat}$ in $L_{Stat}(Sig_{Stat})$ and a truth relation $\models_{Stat}$ for formulas in this language such that $\vdash_{Stat} \Leftrightarrow \models_{Stat}$. They extend the inference and truth relations in equational logic [17]:

$$Spec_{Stat} \models_{Stat} t_1 = t_2 \Leftrightarrow Spec_{Stat} \models_{ADT} t_1 = t_2,$$

$$Spec_{Stat} \vdash_{Stat} t_1 = t_2 \Leftrightarrow Spec_{Stat} \vdash_{ADT} t_1 = t_2.$$

We give an inference relation and a truth definition for dynamic logic with equality below (section 4.1) that extends these relations.

DEFINITION 10 (STATIC CONSTRAINT SPECIFICATION)

Let $Spec_{ADT} = (Sig_{ADT}, E)$ be an ADT specification. Then $Spec_{Stat} = (Sig_{Stat}, C)$ is a *static constraint specification over* $Spec_{ADT}$ iff it is a conservative enrichment of $Spec_{ADT}$.

Our running example contains the following static constraint specification.

static constraint spec *StaticLibraryConstraints*
  import
    Persons, BookIdentifiers, Queues
  functions
    reservations : *BOOK* → *QUEUE*
  predicates
    Reserved : *BOOK*
    Available : *BOOK*
    Present : *BOOK*
  variables
    b : *BOOK*
  static constraints
[C1] *Available*(*b*) ← *Present*(*b*) ∧ ¬*Reserved*(*b*)
end spec *StaticLibraryConstraints*

The underlying ADT specification in the example is the specification of persons, books, and queues. In the specification of dynamic library constraints to

be given later, *Reserved(b)* will be set to *true* if a person reserves a book and is set to *false* if the queue of reservations is emptied. We use the convention that the leftmost quantifier can be omitted if it is a universal quantifier. All others must be shown.

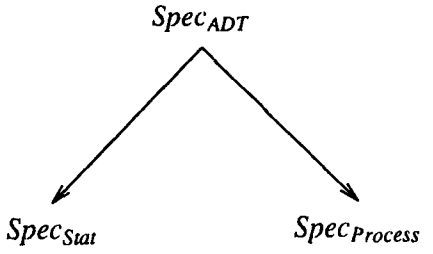

Fig. 2. Two conservative extensions of the underlying ADT specifications.

Figure 2 shows the import relations between specifications. The underlying ADT specification is conservatively extended in two directions, which will not be related until we introduce dynamic specifications below.

Since $Spec_{Stat}$ is an *enrichment* of $Spec_{ADT}$, it does not add any sorts. Since it is a *conservative* enrichment of $Spec_{ADT}$, it does not add any data elements to the ADT sorts and it does not identify any data elements that were not already identified in $Spec_{ADT}$. This is important in a database context, as we will see in a moment.

DEFINITION 11 (POSSIBLE WORLDS)

Let $Spec_{Stat}$, be a static constraint signature over $Spec_{ADT}$. A *possible world* of $Spec_{Stat}$ is a model of $Spec_{Stat}$. The set of all possible worlds of $Spec_{Stat}$ is called $PW(Spec_{Stat})$, or PW if the specification is understood or irrelevant. We use w as metavariable over PW.

Each possible world provides an interpretation of the function and predicate symbols in $Spec_{Stat}$ that satisfies the constraints. Due to the requirement that $Spec_{Stat}$ be a conservative enrichment of $Spec_{ADT}$, the interpretation of the sort and operation symbols, all declared in $Spec_{ADT}$, is the same in all possible worlds. Thus, the underlying ADT $\mathcal{A}_{ADT}$ is a $Spec_{ADT}$-reduct of the intended process algebra and the intended model of $Spec_{Stat}$, as illustrated in fig. 3, where w is any possible world of $Spec_{Stat}$. If $w \in PW$, it is a possible model of Sig_{Stat}, so we can write $w \models_{Stat} \phi$. The requirement that $Spec_{Stat}$ is a conservative enrichment of $Spec_{ADT}$ means that $Spec_{Stat}$ is a complete and consistent extension of $Spec_{ADT}$, and this is a generalization of Reiter's [40,41] domain closure and unique name axioms.

Reiter's *domain closure* axioms say that each element in the model must have a constant as name. The initial semantics generalizes this by requiring each element

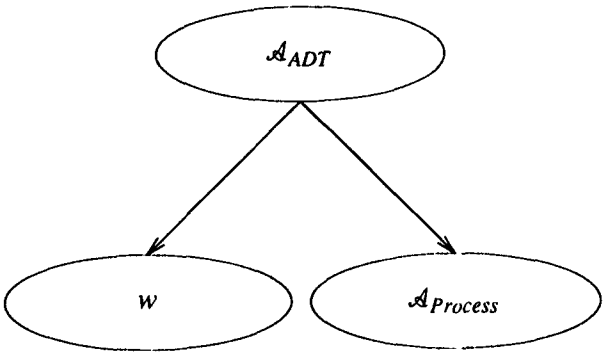

Fig. 3. The underlying ADT is a reduct of two different algebras.

to have a closed term as name. By allowing only *enrichments* of the ADT specification, no data sorts are added in $Spec_{Stat}$, and by the completeness requirement, the domain closure property is preserved in each w. Thus, all sorts have the same extension in all possible $w \in PW$, and there are no unnamed elements in any sort.

In addition, by the consistency requirement, the names given in $Spec_{Stat}$ to data elements in w are equal to the names given in $Spec_{ADT}$. For example, in every state of the library and for every closed term *reservation*(b), there will be a data element q of *QUEUE* such that *reservation*(b) = q is true. Without this requirement, *reservation*(b) could be a data element of *QUEUE* that would not be equal to any data element specified in *QUEUES*.

Reiter's *unique name* axiom says that different constants denote different elements of the model. Initial semantics generalizes this to the property that two closed terms are equal if and only if they can be proved equal from $Spec_{ADT}$. The conservativeness requirement preserves this property in $Spec_{Stat}$.

Because all data sorts contain the same elements in all possible worlds, we must add a mechanism to distinguish, in each possible world, between those elements that actually exist and those that only have possible existence. We do this by assuming a special unary predicate E that is applicable to terms of all sorts. This is a standard solution to existence problems in modal logic [15]. If $E(x)$ is true in w, then by definition, x actually exists in w. If $s_1 \leq s_2$, then by the construction of order-sorted algebras, in any world w we then have that the set of existing objects of sort s_1 is a subset of the set of existing objects of sort s_2.

We use the abbreviations

$$\exists^E x\phi(x) \stackrel{\text{def}}{\Leftrightarrow} \exists x(E(x) \wedge \phi(x))$$

and

$$\forall^E x\phi(x) \stackrel{\text{def}}{\Leftrightarrow} \forall x(E(x) \rightarrow \phi(x)).$$

We have no requirements for the unary existence predicate E in our definitions. We may want to avoid "dangling pointers" by adding axioms like

$$P(x_1, \ldots, x_n) \rightarrow E(x_1) \wedge \ldots \wedge E(x_n)$$

to the specification. Nothing hinges on these axioms in this paper, and we just ignore them.

4. Specification in dynamic logic

In this section, we temporarily turn our attention to a more general logic. In section 4.1, we define the syntax, inference rules and semantics of a first-order logic with multiple modalities and equality. In section 4.2, we turn to the more special case of specification in dynamic logic.

4.1. LANGUAGE, SEMANTICS AND INFERENCE RULES

Dynamic logic is a logic to reason about postconditions of terminating processes [20,30]. Usually, the terms denoting these processes are not defined algebraically and the focus is on the logic of postconditions. In this section, we define a particular version of dynamic logic that explicitly defines a logic of processes in addition to the logic of postconditions.

DEFINITION 12 (SIGNATURE OF DYNAMIC LOGIC)

A *dynamic logic signature* Sig_{Dyn} is an order-sorted signature consisting of two (possibly overlapping) subsignatures Sig_{Stat} and $Sig_{Process}$, with

$$Sig_{Dyn} = Sig_{Stat} \cup Sig_{Process}.$$

This is a minimal definition of a dynamic logic signature. In the definition, we have made no assumptions about the signatures Sig_{Stat} and $Sig_{Process}$ and until further notice, these are just arbitrary order-sorted signatures. (Of course, these names foreshadow the use to which they will be put later on.) Sig_{Stat} and $Sig_{Process}$ are arbitrary order-sorted signatures and therefore generate languages $L(Sig_{Stat})$ and $L(Sig_{Process})$. There is an inference relation $\vdash_{Stat}$ and a truth definition $\models_{Stat}$ for these languages, that collapse to the equational versions $\vdash_{Eq}$ and $\models_{Eq}$ for an equational signature.

DEFINITION 13 (LANGUAGE OF DYNAMIC LOGIC)

Let $Sig_{Dyn} = Sig_{Stat} \cup Sig_{Process}$ be a dynamic logic signature. The dynamic logic language $L(Sig_{Dyn})$ generated by Sig_{Dyn}, with typical elements Φ and Ψ, is given by the BNF:

$$\Phi ::= \phi \mid \Phi_1 \vee \Phi_2 \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \rightarrow \Phi_2 \mid \Phi_1 \leftrightarrow \Phi_2 \mid [\beta]\Phi,$$

where $\phi \in L(Sig_{Stat})$ and β is a term over $Sig_{Process}$. We use

$$\langle\beta\rangle\Phi$$

as an abbreviation of $\neg[\beta]\neg\Phi$.

The intention is that terms over $Sig_{Process}$ are process terms, and that the intuitive semantics of $[\beta]\Phi$ is

"after execution of β, Φ holds necessarily".

This is partial correctness of nondeterministic programs β, for it says that if β terminates, i.e. leads to a next world, then in all next worlds to which β can lead, Φ holds. The intuitive semantics of $\langle\beta\rangle\Phi$ is, dually

"there is an execution of β after which Φ holds".

This is total correctness, for it says that β terminates, and that in at least one state reachable by β, Φ holds.

Note that by our definition of $L_{Dyn}(Sig_{Dyn})$-formulas, terms over $Sig_{Process}$ can only occur inside the modal operator. A dynamic logic formula containing no modal operators is just a formula from $L(Sig_{Stat})$.

We now assume that for $L(Sig_{Stat})$ and $L(Sig_{Process})$, the concept of a semantic structure has been defined, and use these to define the concept of a structure for $L(Sig_{Dyn})$.

DEFINITION 14 (INTERPRETATION STRUCTURES FOR $L(Sig_{Dyn})$)

An *interpretation structure* for $L(Sig_{Dyn})$ is a triple

$$\mathcal{M}_{Dyn} = (PW, \mathcal{A}_{Process}, \rho),$$

where

- PW is a non-empty set of non-empty structures for $L(Sig_{Stat})$;
- $\mathcal{A}_{Process}$ is a non-empty structure for $L(Sig_{Process})$;
- if Σ is the set of all sort-preserving assignments σ to variables, then ρ is a function

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

satisfying the requirement that for $\beta_1, \beta_2 \in T_{Process}(X)$,

$$[\![\beta_1]\!]_{\sigma,\mathcal{A}_{Process}} = [\![\beta_2]\!]_{\sigma,\mathcal{A}_{Process}} \Rightarrow \rho(\beta_1)(\sigma) = \rho(\beta_2)(\sigma).$$

ρ is called the *state transition semantics* of processes.

The definition is illustrated in fig. 4.

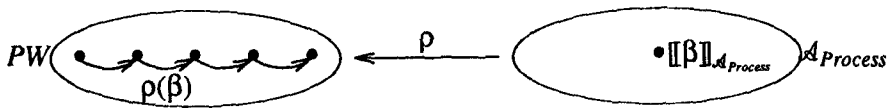

Fig. 4. Structures for $L(Sig_{Dyn})$.

Remarks

(1) *PW* functions as a Kripke structure for $L(Sig_{Dyn})$, and each $w \in PW$ interprets $L(Sig_{Stat})$.

(2) The Kripke structure *PW* is extended with an algebra $\mathcal{A}_{Process}$ in which process terms are interpreted. There are no requirements on the relation between *PW* and $\mathcal{A}_{Process}$.

(3) There are requirements, however, on the relation between $\mathcal{A}_{Process}$ and the accessibility relations on *PW*. Roughly, each process $p \in \mathcal{A}_{Process}$ must define an accessibility relation $\rho(p)$ on *PW*, which may be nondeterministic, i.e. several worlds are accessible from one world by following $\rho(p)$ in one step. This is because in general, processes are nondeterministic in the sense that the state reached by performing the process may not be defined uniquely.

(4) To explain the definition of ρ more precisely, for each process term $\beta \in T_{Process}(X)$ and each variable assignment $\sigma \in \Sigma$, an accessibility relation $\rho(\beta)(\sigma)$ is defined in such a way that terms that are equal in $\mathcal{A}_{Process}$ are assigned equal accessibility relations. The denotation function $[\![\cdot]\!]_{\sigma, \mathcal{A}_{Process}}$ is the usual interpretation function of equational logic.

(5) If we think of $Sig_{Process}$ as a process signature, $L(Sig_{Dyn})$-structures assign two kinds of semantics to process terms. One is an *algebraic semantics of uninterpreted process terms*, in which the operations on processes (choice, sequence, parallel composition) are given a semantics, without looking at the effect of the processes on the state of the world. The other kind of semantics is a kind of *labeled transition system* that defines the effect of each process on the state of the world. $\rho(p)$ defines the effect p has on the extension of the function and predicate symbols in $L(Sig_{Dyn})$.

DEFINITION 15 (TRUTH OF $L(Sig_{Dyn})$-FORMULAS)

Let $\mathcal{M}_{Dyn} = (PW, \mathcal{A}_{Process}, \rho)$ be a structure for $L(Sig_{Dyn})$ and $\sigma: X \to \mathcal{M}_{Dyn}$ be an assignment to all variables, then *truth of a dynamic logic formula* in $w \in PW$ under assignment σ 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.

For each σ, we define $\mathcal{M}_{Dyn}, \sigma \models_{Dyn} \Phi$ iff $w, \sigma \models_{Dyn} \Phi$ for all $w \in PW$. Truth of a formula in a world ($w \models_{Dyn} \Phi$) and in a structure ($\mathcal{M}_{Dyn} \models_{Dyn} \Phi$) are defined as usual.

This truth definition coincides with the standard truth definition for ϕ in $L(Sig_{Stat})$, which we denote $\models_{Stat}$, and with the standard truth definitions for equations in $L(Sig_{ADT})$, which we denote $\models_{ADT}$. We have for all $w \in PW$ that

$$w \models_{Stat} \phi \Leftrightarrow w \models_{Dyn} \phi;$$

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

We will therefore omit the subscript from $\models$ from now on.

Next to a truth definition, we need an inference relation for $L(Sig_{Dyn})$. We use the inference rules [N] and [DL1] of modal logic, and the rules for equational reasoning [Ref], [Sym], [Tran], [Con1], [Sub1], and [Con2], extended with one extra rule for equality, [Sub2], as given in table 1. We call the set of rules in table 1 $DYN^{=}$.

Table 1

The set $DYN^{=}$ of inference rules for synamic logic with equality.

All axioms and theorems of first-order logic.		
[MP] $\dfrac{\Phi, \Phi \rightarrow \Psi}{\Psi}$	[G] $\dfrac{\Phi}{\forall x(\Phi)}$	[N] $\dfrac{\Phi}{[\beta]\Phi}$
[Ref]	[Sym]	[Tran]
$t = t$	$t_1 = t_2 \rightarrow t_2 = t_1$	$(t_1 = t_2 \wedge t_2 = t_3) \rightarrow t_1 = t_3$
[Con1] $\dfrac{t_1 = t_2}{t\{x \mapsto t_1\} = t\{x \mapsto t_2\}}$		[Sub1] $\dfrac{t_1 = t_2}{t_1\{x \mapsto t\} = t_2\{x \mapsto t\}}$
[Con2] $\dfrac{t_1 = t_2}{P(t_1) \leftrightarrow P(t_2)}$		[Sub2] $\dfrac{\beta_1 = \beta_2}{([\beta_1]\Phi \leftrightarrow [\beta_2]\Phi)}$
[DL1] $[\beta](\Phi_1 \rightarrow \Phi_2) \rightarrow ([\beta]\Phi_1 \rightarrow [\beta]\Phi_2)$		

An inference rule has the form $\frac{H}{\Phi}$, with H a set of formulas. If $H = \varnothing$, the rule is also called an *axiom*. The equality axioms [Ref], [Sym] and [Tran] hold for all terms, including process terms. It is extremely important to note that there are two kinds of inferences we can do in $L(Sig_{Dyn})$:

- inferences in $L(Sig_{Process})$ in which we reason about equality of process terms, and
- dynamic logic inferences on $L(Sig_{Dyn})$ formulas. These include inferences in $L(Sig_{Stat})$.

Note that $DYN^{=}$ restricted to equations is just equational logic.

If Φ is derivable from a set H of $L(Sig_{Dyn})$ formulas, we write $H \vdash_{Dyn} \Phi$. To reiterate what we just said, these inference rules monotonically extend the corresponding rules $\vdash_{Stat}$ and $\vdash_{ADT}$ for $L(Sig_{Stat})$ and $L(Sig_{ADT})$. We have

$$H \vdash_{Stat} \phi \Leftrightarrow H \vdash_{Dyn} \phi$$

$$H \vdash_{ADT} t_1 = t_2 \Leftrightarrow H \vdash_{Dyn} t_1 = t_2.$$

We drop the subscript from $\vdash$ from now on.

THEOREM 1 (SOUNDNESS AND COMPLETENESS)

$$\vdash \Phi \Leftrightarrow \models \Phi.$$

Proof

Soundness is easy to prove. For completeness, the [Ref], [Sym], [Tran], [Con1], [Con2], and [Sub1] axioms are known to completely axiomatize first-order logic with equality. It is essential that we look at all models here; if we restrict ourselves to initial models, completeness would only hold with respect to ground equations [9].

We prove completeness of the model part of $DYN^{=}$ by using a Henkin-style proof that is standard in model logic [24]. First note that [N] and [DL1] characterize Kripke structures, with accessibility relations R_β for each $\beta \in T_{Process}(X)$. Moreover, the inference rule [Sub2] corresponds to the property that

$$\mathcal{A}_{Process} \models \beta_1 = \beta_2 \Rightarrow R_{\beta_1} = R_{\beta_2}. \tag{13}$$

Let us call the class of Kripke structures satisfying (13) $\mathcal{C}$. Completeness is then the statement that for all $\Phi \in L(Sig_{Dyn})$,

$$\text{if for all } \mathcal{M} \in \mathcal{C} \text{ we have } \mathcal{M} \models \Phi, \text{ then } DYN^{=} \vdash \Phi.$$

This is equivalent to the statement that for all $\Phi \in L(Sig_{Dyn})$, there is an $\mathcal{M} \in \mathcal{C}$ such that

$$\not\vdash \neg\Phi \Rightarrow \mathcal{M} \not\models \neg\Phi. \tag{14}$$

According to the standard argument, in order to prove (14), it is sufficient to prove that all maximally consistent sets of formulas $H \subseteq L(Sig_{Dyn})$ are satisfiable within $\mathcal{C}$, i.e. that there is an $\mathcal{M} \in \mathcal{C}$ and a world $w \in \mathcal{M}$ such that

$$\mathcal{M}, w \models \phi \text{ for all } \phi \in H.$$

To do this, the standard argument constructs a canonical model, which we call $\mathcal{M}_0$, in which the worlds w are maximally consistent sets of $L(Sig_{DYN})$ formulas, and the reachability relation is defined by

$$R_\beta(w_1, w_2) \stackrel{\text{def}}{\Leftrightarrow} \forall \Phi([\beta]\Phi \in w_1 \Rightarrow \Phi \in w_2).$$

This canonical model satifies the property that $\mathcal{M}_0, w \models \phi$ for every $\phi \in w$. In order to prove (14), it must be shown that this canonical model satisfies the defining property (13) for the class $\mathcal{C}$ of Kripke structures. However, this is easy to prove, for let $\mathcal{A}_{Process} \models \beta_1 = \beta_2$. Then with inference rule [SUB2], we infer

$$[\beta_1]\Phi \leftrightarrow [\beta_2]\Phi.$$

It is known that each $w \in \mathcal{M}_0$ contains all theorems (since it is a maximally consistent set), so we know that

$$([\beta_1]\Phi \leftrightarrow [\beta_2]\Phi) \in w_1.$$

Because worlds in $\mathcal{M}_0$ are maximally consistent sets, and hence are closed under logical consequence, we have that for all $\Phi \in L(Sig_{Dyn})$,

$$([\beta_1]\Phi \in w_1 \Rightarrow \Phi \in w_2) \Leftrightarrow ([\beta_2]\Phi \in w_1 \Rightarrow \Phi \in w_2).$$

We conclude that for all $w_1, w_2 \in \mathcal{M}_0$, $R_{\beta_1}(w_1, w_2) = R_{\beta_2}(w_1, w_2)$, and therefore $R_{\beta_1} = R_{\beta_2}$. This shows that $\mathcal{M}_0 \in \mathcal{C}$, and we have proven (14). □

THEOREM 2

The following theorems hold in $L(Sig_{Dyn})$ with inference relation $DYN^=$.

(1) $[\beta]true.$

(2) $\neg\langle\beta\rangle false.$

(3) $[\beta](\Phi_1 \wedge \Phi_2) \leftrightarrow ([\beta]\Phi_1 \wedge [\beta]\Phi_2).$

(4) $[\beta](\Phi_1 \vee \Phi_2) \leftarrow ([\beta]\Phi_1 \vee [\beta]\Phi_2).$

(5) $\langle\beta\rangle(\Phi_1 \vee \Phi_2) \leftrightarrow (\langle\beta\rangle\Phi_1 \vee \langle\beta\rangle\Phi_2).$

(6) $\langle\beta\rangle(\Phi_1 \wedge \Phi_2) \rightarrow (\langle\beta\rangle\Phi_1 \wedge \langle\beta\rangle\Phi_2).$

Proof

See [34]. The others follow from [Sub2] and the step axioms for our process theory. □

These are standard theorems in modal logic. They do not assume anything about the process terms β and they do not relate the structure of process terms to the structure of postconditions. To state truths about the process terms, we must assume a particular process specification. Using our example specification *Processes*, we have the following.

THEOREM 3

The following theorems hold in $L(Sig_{Dyn})$, using the specification *Process* to supply $Sig_{Process}$, and $DYN^{=}$ as inference relation.

(1) $[-(\alpha_1 + \alpha_2)]\Phi \leftrightarrow [(-\alpha_1 \,\&\, -\alpha_2)]\Phi$.

(2) $[-(\alpha_1 \,\&\, \alpha_2)]\Phi \leftrightarrow [(-\alpha_1 + -\alpha_2)]\Phi$.

(3) $[-(-\alpha_2)]\Phi \leftrightarrow [\alpha]\Phi$.

Proof

The proof is immediate with [Sub2]. □

These theorems hold for any modal logic and do not take into account the special nature of the *Process* specification. In the multimodal logic considered in this section, formulas like $[\alpha_1 + \alpha_2]\Phi \leftrightarrow [\alpha_1]\Phi \wedge [\alpha_2]\Phi$ are not valid because at this point, we do not relate the internal structure of Φ to the internal structure of β in $[\beta]\Phi$. The truth of such formulas depends upon the particular choice of ρ in a model, and to choose a particular ρ, we should choose particular specifications. We therefore turn to specifications in dynamic logic in the next section, and then define intended models of these specifications in which the desired validities hold. We start with defining what a specification in $L(Sig_{Dyn})$ should look like.

4.2. DYNAMIC CONSTRAINT SPECIFICATIONS

DEFINITION 16 (DYNAMIC CONSTRAINT SIGNATURE)

Let $Spec_{ADT}$ be an ADT signature and $Sig_{DYN} = Sig_{Stat} \cup Sig_{Process}$ be a dynamic logic signature. Sig_{DYN} is called a *dynamic constraint signature* over Sig_{ADT}, Sig_{Stat}, and $Sig_{Process}$ iff

- Sig_{Stat} is a static contraint signature over Sig_{ADT},
- $Sig_{Process}$ is a process signature over Sig_{ADT}, and
- the function symbols declared in both extensions are the function symbols declared in Sig_{ADT}, i.e. $\mathbb{F}_{Stat} \cap \mathbb{F}_{Process} = \mathbb{F}_{ADT}$.

From now on, we assume that our Sig_{Dyn} satisfies definition 16. Figure 5 shows the import relations between the signatures. The dynamic signature does not add any declarations to the static signature or the process signature. All machinery required to specify dynamic constraints is already present in those signatures.

DEFINITION 17 (DYNAMIC CONSTRAINT SPECIFICATION)

Let $Spec_{ADT}$ be an ADT specification with signature Sig_{ADT}, and let $Sig_{Dyn} = Sig_{Stat} \cup Sig_{Process}$ be a dynamic constraint signature. Then a *dynamic constraint specification* over $Spec_{ADT}$, $Spec_{Stat}$ and $Spec_{Process}$ is defined by

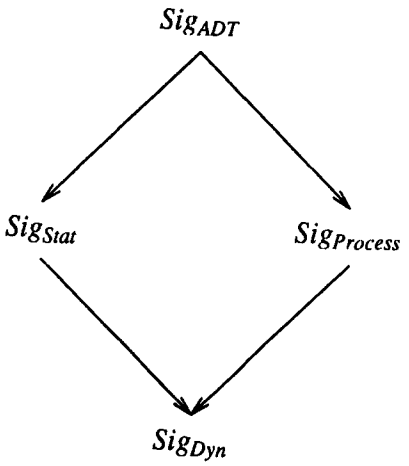

Fig. 5. Import relations between signatures.

$$Spec_{Dyn} = (Sig_{Dyn}, E_{Dyn}),$$

where

- $E_{Dyn} \subseteq L(Sig_{Dyn})$,
- $(Sig_{Stat}, E_{Dyn} \cap L(Sig_{Stat}))$ is a static constraint specification over $Spec_{ADT}$, and
- $(Sig_{Process}, E_{Dyn} \cap L(Sig_{Process}))$ is a process specification over $Spec_{ADT}$.

Thus, $Spec_{Dyn}$ contains a static constraint specification over $Spec_{ADT}$ and a process specification over $Spec_{ADT}$. Both extend $Spec_{ADT}$ conservatively, and only process specification is allowed to introduce any new sorts. All formulas added by $Spec_{Dyn}$ to these two specifications are modal formulas.

An example dynamic constraint specification is

dynamic constraint spec *DynamicLibraryConstraints*
import
StaticLibraryConstraints, Processes
variables
p : *PERSON*
b : *BOOK*
q : *QUEUE*
dynamic constraints
[D0] $[p : borrow(b)]Borrowed(b, p) \wedge \neg Present(b)$
[D1] $Reserved(b) \rightarrow$
$(p = head(q) \wedge q = reservations(b) \rightarrow$
$[p : borrow(b)]reservations(b) = tail(q))$

[D2] $Reserved(b) \rightarrow$
$(\neg(p = head(q) \wedge q = reservations(b)) \rightarrow [p : borrow(b)]false)$
end spec *DynamicLibraryConstraints*

[D0] says that the effect of borrowing a book is that it is not present and is borrowed. [D1,2] say that in the case of reserved books, we remove the borrower from the queue of reservations if he is the first in the queue, otherwise the event is blocked.

Because the process specification and the static contraint specification conservatively extend the same ADT specification, ADT terms that occur in process terms as well as in static constraint formulas have the same meaning and are subject to the same equalities. For example, suppose we have a predicate $Payed : PERSON \times MONEY$. Then it is easy to show that we have

$$\begin{aligned}[p : pay(\$3)]Payed(p, \$3) &\leftrightarrow [p : pay(\$1 + \$2)]Payed(p, \$3)\\ &\leftrightarrow [p : pay(\$1 + \$2)]Payed(p, \$1 + \$2)\\ &\leftrightarrow [p : pay(\$3)]Payed(p, \$1 + \$2).\end{aligned}$$

Reasoning about the equality of process terms is done within $Spec_{Process}$ and reasoning about the equality of predicate applications is done in $Spec_{Stat}$. The inference rules of $DYN^{=}$ make these inferences $Spec_{Dyn}$ inferences as well. The point of the example is that the equalities between terms $Spec_{ADT}$-terms derived in $Spec_{Process}$ will be the same as those derived in $Spec_{Stat}$, because both are conservative extensions of $Spec_{ADT}$.

5. Specification in deontic logic

If Sig_{Dyn} is a dynamic logic signature, $L(Sig_{Dyn})$ can be used as a language for deontic constraints by introducing *violation states* $V : \iota : \alpha$ and $V : \iota : -\alpha$, with the intuitive meaning of "ι illegally performed α" and "ι illegally failed to perform α", respectively. With any dynamic logic signature Sig_{Dyn}, a *deontic logic signature* Sig_{Deon} corresponds, in which these predicates are all declared for all ι and α. We introduce the deontic modalities by definition for all process terms as follows:

- $\mathbf{P}(\alpha) \stackrel{\text{def}}{\Leftrightarrow} \neg[\alpha]V{:}\,\alpha$("$\alpha$ is permitted"),
- $\mathbf{O}(\alpha) \stackrel{\text{def}}{\Leftrightarrow} [-\alpha]V{:}\,\alpha$("$\alpha$ is obligatory"),
- $\mathbf{F}(\alpha) \stackrel{\text{def}}{\Leftrightarrow} \neg\mathbf{P}(\alpha)$ ("α is forbidden").

Every action now has a *deontic effect* and a *non-deontic effect*. The deontic effect is that it raises a *violation flag* (a V-predicate becomes true) or not; the non-deontic effect is the effect it has on the other predicates and on the attributes.

This simple introduction of deontic modalities into dynamic logic is surprisingly powerful and resolves many paradoxes of deontic logic. We will not go into this, but refer the reader to work done earlier on this [10,34,35,51].

An example deontic specification is

deontic constraint spec *DeonticLibraryConstraints*
import
DynamicLibraryConstraints
variables
p : *PERSON*
b, b' : *BOOK*
l : *LIBRARY*
deontic constraints

[N0] $\mathbf{P}(p : borrow(b)) \leftrightarrow$
$Member(p) \wedge \neg V : p : -return(b')$

[N1] $[p : borrow(b)]\ [\mathbf{clock}(21)](Borrowed(b, p) \leftrightarrow$
$V : p : -return(b))$

[N2] $V : p : -return(b) \leftrightarrow$
$[p : return(b)](V : p : return(b) \wedge \neg V : p : -return(b))$

[N3] $[p : pay(\$2, b)]\ \neg V : p : return(b)$

[N4] $[p : borrow(b)]\mathbf{O}(p : return(b), < 21)$

[N5] $[p : borrow(b)]\ [\mathbf{clock}(21)]\ (Borrowed(b, p) \leftrightarrow \mathbf{O}(l : remind(p, b))$

[N6] $Present(b) \wedge Reserved(p) \leftrightarrow$
$(p = front(reservations(b))\mathbf{O}(l : notify(p, b)))$

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. $V : p : -return(b')$ is a predicate with two arguments, p and b'. It is the violation flag raised when p does not perform $return(b')$ on time. [N1] gives a condition under which this flag is raised. It says that the flag $V : p : -return(b')$ is raised if the book is still borrowed after 21 days. **clock**(21) is the process in which the clock makes 21 ticks (defined in [48]). [N2] says that returning the book too late cancels the violation of not returning it, but raises another one, called $V : p : return(b)$, which [N3] says can be cancelled by paying \$2. [N4] says that borrowing a book creates the obligation to return it within 21 days. $\mathbf{O}(p : return(b), < 21)$ is an obligation to perform a choice of steps, viz. return the book, or let the clock tick once and return the book, etc. A precise semantics of this binary predicate is given in [48]. 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.

6. Two models for free choice

The axioms presented so far have many models, and the completeness result of theorem 1 holds only if we take all models into account. We now return to our original concern with the paradox of free choice permission and define two particular models of a deontic constraint specification that resolve that paradox. Since a deontic constraint specification is just a dynamic constraint specification with some extra predicates, we continue the discussion in terms of dynamic constraint specifications.

A model of a dynamic constraint specification will have a structure as shown in fig. 1, which is repeated in fig. 6. By the definition of dynamic constraint specifications, each $w \in PW$ and the process algebra $\mathcal{A}_{Process}$ must contain the underlying

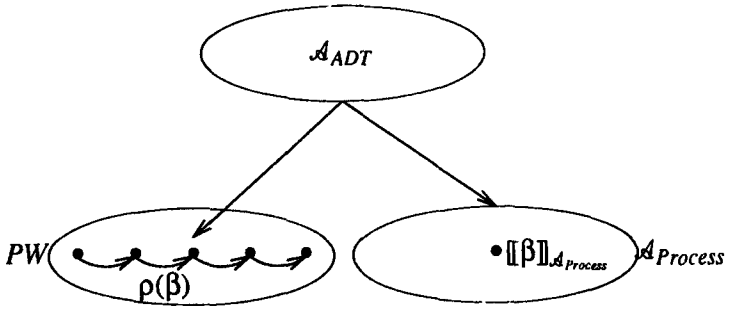

Fig. 6. The structure of any intended model of a dynamic constraint specification.

ADT. To define an intended model $\mathcal{M}_f$ for free choice, we will take for PW the set of all possible worlds that contain $\mathcal{A}_{ADT}$ as a $Spec_{ADT}$-reduct. In our example specification, we took the initial algebra of *Processes* as our $\mathcal{A}_{Process}$, but any other process algebra containing $\mathcal{A}_{ADT}$ as $Spec_{ADT}$ is also admissible. We assume such a choice made, and then define two intended models, $\mathcal{M}_f$ and $\mathcal{M}_a$, by defining two functions ρ_f and ρ_a that define the effect of each process on the world. Roughly, both functions will use a step-trace semantics, in which each action is interpreted as a set of possible state transitions, and each process containing sequence operators as a set of traces of such transitions. The two functions differ in the interpretation they assign to active choice.

6.1. MODEL 1: A MODEL FOR FREE CHOICE

First, we assume a function

$$effect : T_{\text{ATOMIC\_ACTION}}(X) \rightarrow (\Sigma \rightarrow (PW \rightarrow PW)).$$

effect(a)(σ) defines the effect of an atomic action on the state of the world. In general, a dynamic logic specification does not determine the effect of actions 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 action is applied. We leave open how *effect* is chosen, and require only that the function satisfies

$$[\![\mathbf{a}]\!]_{\mathcal{A}_{Process},\sigma} = [\![\mathbf{b}]\!]_{\mathcal{A}_{Process},\sigma} \Rightarrow effect(\mathbf{a})(\sigma) = effect(\mathbf{b})(\sigma). \tag{15}$$

We first define ρ_f on $T_{ACTION}(X)$ and later extend this to a function on $T_{PROCESS}(X)$. To define $\rho_f : T_{\text{ACTION}}(X) \to (\Sigma \to (PW \to \mathcal{P}(PW)))$, we define a set of functions $PW \to PW$ to each $\alpha \in T_{\text{ACTION}}(X)$. We do this inductively. First, we need the domain of *steps*.

DEFINITION 18 (STEPS)

The set of all possible *steps* is

$$STEP = \mathcal{F}^+(T_{\text{ATOMIC\_ACTION}}(X)),$$

with typical element S. ($\mathcal{F}^+$ is the finite non-empty subset operator.) The elements of *STEP* are written

$$\begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_n \end{bmatrix}.$$

If $\iota : a \in S$, we write $\iota \in S$.

DEFINITION 19 (COMPATIBLE STEPS)

A step

$$\begin{bmatrix} \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_n \end{bmatrix}$$

is called *compatible* iff for all $w \in PW$ and all $\sigma \in \Sigma$

$$effect(\mathbf{a}_1)(\sigma) \circ \ldots \circ effect(\mathbf{a}_n)(\sigma)(w) = effect(\mathbf{a}_{i_1}) \circ \ldots \circ effect(\mathbf{a}_{i_n})(w)$$

for all permutations $\langle i_1, \ldots, i_n \rangle$ of $\langle 1, \ldots, n \rangle$. The set of compatible steps is called $STEP^{Comp}$.

A step is a synchronous occurrence of a non-empty finite set of atomic actions. To cater for nondeterminism, we introduce step choices.

DEFINITION 20 (STEP CHOICES)

The set of all possible *step choices* is defined by

$$CHOICE = \mathcal{P}(STEP^{Comp}),$$

with typical element C.

A step choice C is thus any set of the form

$$\left\{ \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_n \end{bmatrix}, \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_m \end{bmatrix}, \ldots \right\}.$$

This is a passive choice between steps. To define our intended free choice model, we must define an accessibility relation for every process term. We will call this relation ρ_f. As an intermediate step in the definition of ρ_f, we first define a step choice for every action. The case of processes containing the sequence operator is a simple extension of this, given later.

DEFINITION 21 (STEP CHOICE INTERPRETATION)

The function

$$choice : T_{\text{ACTION}}(X) \to CHOICE$$

is defined as follows:

(1) $choice(\mathbf{a}) = \{S \in STEP^{Comp} \mid \mathbf{a} \in S\}$.

(2) $choice(\iota : fail) = \{S \in STEP^{Comp} \mid \iota \notin S\}$, where $\iota \in S$ is shorthand for "there is an a with $\iota : a \in S$".

(3) $choice(\iota : any) = \{S \in STEP^{Comp} \mid \iota \in S\}$.

(4) $choice(\alpha_1 + \alpha_2) = choice(\alpha_1) \cup choice(\alpha_2)$.

(5) $choice(\alpha_1 \,\&\, \alpha_2) = choice(\alpha_1) \cap choice(\alpha_2)$.

(6) $choice(-\alpha) = STEP^{Comp} \backslash choice(\alpha)$.

(7) $choice(\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2)) = choice(\iota : \alpha_1) \cap choice(\iota : \alpha_2)$.

(8) $choice(\iota_0 : (\iota_1 : \alpha_1 \oplus \iota_2 : \alpha_2)) = choice(\iota_1 : \alpha_1) \cup choice(\iota_2 : \alpha_2)$, where at least two of ι_0, ι_1, ι_2 differ.

choice is well defined, because it preserves compatibility of steps.

Remarks

(1) $choice(\mathbf{a})$ is a passive choice over the set of all compatible steps in which **a** participates. This is nondeterminism of the underspecification kind. Performance

of **a** can thus lead to any world out of a set of possible worlds. This makes it easy to interpret action negation below.

(2) *choice*(ι : *fail*) is the set of all compatible synchronization sets in which ι does not participate. Together with clause (4) of the definition, this has the consequence that equality (8),

$$\iota : \alpha + \iota : \mathit{fail} = \iota : \alpha,$$

does not hold. *choice*($\iota : \alpha + \iota :$ *fail*) is a choice in which ι participates or in which it does not participate, and this is not equal to *choice*($\iota : \alpha$), which is a choice in which ι participates.

With clause (5) of the definition, it has the consequence that equation (9),

$$\iota : \alpha \,\&\, \iota : \mathit{fail} = \iota : \mathit{fail},$$

does not hold. The choice associated with the left-hand side is empty, because the intersection between *choice*($\iota : \alpha$) and $\iota :$ *fail* is empty, but the choice associated with the right-hand side includes all steps in which ι does not participate. Only in a single-actor world are these choices equal.

(3) *choice*(ι : *any*) is the set of all compatible synchronization sets in which ι participates.

(4) Passive choice is just the union of the passive choices that are the branches, This gives us commutativity, associativity, and idempotence of passive choice (properties `[PC1-3]` in the specification *Actions*). Because ($\iota : \alpha$) $\cup$ *choice*(ι : *any*) = *choice*(ι : *any*), axiom `[ANY1]` in *Actions* is satisfied.

(5) α_1 & α_2 is a passive choice over steps that are in both *choice*(α_1) and *choice*(α_2). There may be no common steps, so synchronous execution may fail. This definition gives us distributivity of synchronization over passive choice (axiom `[D]` in *Actions*). This interpretation of synchronization satisfies $\iota : \alpha \,\&\, \iota :$ *any* $= \iota : \alpha$ (axiom `[ANY2]` in *Actions*).

(6) Actions are negated with respect to all possible steps. Thus, $-\iota : a$ is the set of all possible steps in which ι does not participate with a. These are the steps in which ι participates with another event, or in which ι does not participate at all. Note that $-\iota :$ *fail* is a passive choice over $\iota :$ *any* and any action in which ι does not participate. Thus, in this semantics, $-\iota :$ *fail* $\neq \iota :$ *any*. This definition of action negation makes *CHOICE* a Boolean algebra (properties `[N1-3]` in *Actions*).

(7) *choice* defines the effect of free choice as the intersection of the effect of branches and the effect of imposed choice as the union of the effect of the branches. This is a particular view of active choice which formalizes the *deontic effect* of such a choice as follows. In $\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2)$, ι chooses between α_1 *and* α_2, and to resolve the paradox of free choice permission, we must formalize the fact that ι must have permission to perform both branches of the choice, not just one

as in classical deontic logic. This is formalized by (7) by simply taking the effect of the choice to be the conjunction of the effect of the branches. If one of the branches raises a violation flag, the choice will do so as well. In the case of imposed choice, active choice has the same effect as passive choice, so that the deontic effect of making the imposed choice is equal to the union of the deontic effects of performing the branches.

This gives us what we want for the deontic effect of an action, but for non-deontic effects, this is counterintuitive. For example, it gives us that the deontic effect of choosing between shooting the president and chewing the gum is to raise a violation flag, which gives us that it is forbidden to make the choice because one of the branches is forbidden. However, the non-deontic part of the effect, which does not concern the effect on the violation flag but on the rest of the world, is defined to be the intersection of the effects of the branches as well. But in the example this intersection is empty, which would give us that there is no non-deontic effect of making the choice. This is counterintuitive. The internal choice model presented below does not have this defect.

Having associated a passive choice with every action, we can now define the effect of every action on the world. The accessibility relation ρ_f is then defined in terms of the effect function.

DEFINITION 22 (EFFECT OF A STEP)

The function *effect* is overloaded with the declaration

$$effect : STEP^{Comp} \rightarrow (\Sigma \rightarrow (PW \rightarrow \mathcal{P}(PW)))$$

by defining for $S \in STEP^{Comp}$

$$effect(S)(\sigma) = effect(\mathbf{a}_1)(\sigma) \circ \ldots \circ effect(\mathbf{a}_n)(\sigma),$$

where the *effect* on the right-hand side is the one already defined.

Thus, the effect of performing a finite non-empty set of atomic actions is the composition of the effects of the atomic actions, if they are compatible, and is not defined otherwise.

DEFINITION 23 (FREE CHOICE SEMANTICS)

The state transition semantics for free choice,

$$\rho_f : T_{\text{ACTION}}(X) \rightarrow (\Sigma \rightarrow (PW \rightarrow \mathcal{P}(PW))),$$

is defined by

$$\rho_f(\alpha)(\sigma)(w) = \{effect(S)(\sigma)(w) \mid S \in choice(\alpha)\}.$$

Starting from a world w, ρ_f simply leads to the set of worlds reachable by the steps in $choice(\alpha)$, as determined by the effect of the atomic actions in each step.

Any ρ may assign the same effect to different processes, so that processes that are different in $\mathcal{A}_{Process}$ may be indistinguishable in their effects on the world. For example, passive choice is distributive over synchronization as far as ρ_f is concerned, but this equality does not hold in the process algebra.

THEOREM 4

For any dynamic constraint specification $Spec_{Dyn}$ that includes *Actions* as a process specification, $\mathcal{M}_f$ is a model.

Proof

This has been proven in the remarks explaining our definition of ρ_f. □

We can now relate the internal structure of process terms to the internal structure of postconditions.

THEOREM 5

If ι_0, ι_1 and ι_2 are not all equal, then the following formulas are true in $\mathbf{M}_f$.

(1) $[\alpha_1 + \alpha_2]\Phi \leftrightarrow [\alpha_1]\Phi \wedge [\alpha_2]\Phi$.

(2) $\langle\alpha_1 + \alpha_2\rangle\Phi \leftrightarrow \langle\alpha_1\rangle\Phi \vee \langle\alpha_2\rangle\Phi$.

(3) $[\alpha_1 \,\&\, \alpha_2]\Phi \leftarrow [\alpha_1]\Phi \vee [\alpha_2]\Phi$.

(4) $\langle\alpha_1 \,\&\, \alpha_2\rangle\Phi \rightarrow \langle\alpha_1\rangle\Phi \wedge \langle\alpha_2\rangle\Phi$.

(5) $[\iota_0 : (\iota_1 : \alpha_1 \oplus \iota_2 : \alpha_2)]\Phi \leftrightarrow [\iota_1 : \alpha_1]\Phi \wedge [\iota_2 : \alpha_2]\Phi$.

(6) $\langle\iota_0 : (\iota_1 : \alpha_1 \oplus \iota_2 : \alpha_2)\rangle\Phi \leftrightarrow \langle\iota_1 : \alpha_1\rangle\Phi \vee \langle\iota_2 : \alpha_2\rangle\Phi$.

(7) $[\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2)]\Phi \leftarrow [\iota : \alpha_1]\Phi \vee [\iota : \alpha_2]\Phi$.

(8) $\langle\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2)\rangle\Phi \rightarrow \langle\iota : \alpha_1\rangle\Phi \wedge \langle\iota : \alpha_2\rangle\Phi$.

Proofs of this can be found in [34].

Remarks

(1) Formulas (1)–(4) are standard. To be certain that the effect Φ is produced by a passive choice, we must be certain that it is brought about by both branches (1), but to know that it can be brought about by a passive choice is equivalent to knowing that it can be brought about by at least one of the branches (2).

(2) Formulas (3) and (4) reflect the fact that the synchronized actions may not be compatible. α_1 & α_2 may not have a successor world at all, whereas separately α_1 and α_2 have. $[\alpha_1 \,\&\, \alpha_2]\Phi$ is then trivially true but $[\alpha_1]\Phi \vee [\alpha_2]\Phi$ may be false, so the implication goes only one way. 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 [35] did this by adding *DONE* : α predicates to the language, but Van der Hoek and Meyer [23] show how to do this in general.

(3) Free choice has the properties of synchronous execution and imposed choice has the properties of passise choice. The properties of free choice can be given an intuitive interpretation as follows. We must realize that in active choice, choice does not bring us to a next possible world, but it does occur at a point in time preceding the execution of α_1 and α_2. The left-hand side of (8) then says

"after ι's choice, the system is in a state
where Φ can be brought about",

and this indeed implies the right-hand side, which says that both branches can bring about Φ. Applying the duality, the left-hand side becomes

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

which is equivalent to

"after ι's choice, the system can be in a state
where Φ will be brought about"

and this is the correct reading of the left-hand side of (7). It is indeed implied by the right-hand side of (7), which says that one of the branches will bring about Φ.

Using the definitions of deontic operators, the following theorem follows immediately.

THEOREM 6

If ι_0, ι_1 and ι_2 are not all equal, then the following formulas are true in $\mathbf{M}_f$.

(1) $\mathbf{F}(\alpha_1 + \alpha_2) \leftrightarrow \mathbf{F}(\alpha_1) \wedge \mathbf{F}(\alpha_2)$.

(2) $\mathbf{P}(\alpha_1 + \alpha_2) \leftrightarrow \mathbf{P}(\alpha_1) \vee \mathbf{P}(\alpha_2)$.

(3) $\mathbf{F}(\alpha_1 \,\&\, \alpha_2) \leftarrow \mathbf{F}(\alpha_1) \vee \mathbf{F}(\alpha_2)$.

(4) $\mathbf{P}(\alpha_1 \,\&\, \alpha_2) \rightarrow \mathbf{P}(\alpha_1) \wedge \mathbf{P}(\alpha_2)$.

(5) $\mathbf{F}(\iota_0 : (\iota_1 : \alpha_1 \oplus \iota_2 : \alpha_2)) \leftrightarrow \mathbf{F}(\iota_1 : \alpha_1) \wedge \mathbf{F}(\iota_2 : \alpha_2)$.

(6) $\mathbf{P}(\iota_0 : (\iota_1 : \alpha_1 \oplus \iota_2 : \alpha_2)) \leftrightarrow \mathbf{P}(\iota_1 : \alpha_1) \vee \mathbf{P}(\iota_2 : \alpha_2)$.

(7) $\mathbf{F}(\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2)) \leftarrow \mathbf{F}(\iota : \alpha_1) \vee \mathbf{F}(\iota : \alpha_2)$.

(8) $\mathbf{P}(\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2)) \rightarrow \mathbf{P}(\iota : \alpha_1) \wedge \mathbf{P}(\iota : \alpha_2)$.

Formulas (7) and (8) resolve the paradox of free choice permission. An actor is permitted to do α_1 as well as α_2 iff he is permitted to choose between them and he is forbidden to choose between them if he is forbidden to do one of the branches. For imposed choice and passive choice, on the other hand, the usual validities hold. Thus, $\alpha_1 + \alpha_2$ is permitted to occur if at least one of the branches is permitted to occur. This is natural, for "permitted to occur" means "can lead to a permitted state of the world". On the other hand, $\alpha_1 + \alpha_2$ is forbidden to occur iff both branches are forbidden, for then it is certain that the passive choice will lead to a forbidden state of the world.

Note that $\mathbf{P}(\alpha_1 + \alpha_2)$ has a different reading than $\mathbf{P}(\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2))$. In $\mathbf{P}(\alpha_1 + \alpha_2)$, there is no actor and we just state something about desirability of the possible states of the world that can be reached by the passive choice. In $\mathbf{P}(\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2))$, on the other hand, there is an actor and we state something about the desirability of his making a choice.

To give a state-transition semantics to processes, we must extend the definition of ρ_f to $T_{\text{PROCESS}}(X)$. The basic idea is simply that the effect of the sequence operator on the world is simply a composition of the effect functions of its arguments. The only complication is that deadlock should remain local, so that, for example (see `[FAIL1-2]` in *Processes*)

$$\rho_f(i : \mathit{fail}; \beta_1; \iota : \alpha; \beta_2) = \rho(\iota : \mathit{fail}; \beta_1; \beta_2).$$

Other actors can continue even when ι gets stuck. The easiest way is to define ρ_1 for terms in which all occurrences of ι after ι has failed are removed, and then extend the definition to other process terms that are in its congruence class.

DEFINITION 24 (REDUNDANCY)

Let $\beta \in T_{\text{PROCESS}}(X)$.

(1) Every term of which the main (outermost) operator is ; is called a *sequence.*

(2) A sequence is called *redundant* if it contains a pattern $\iota : \mathit{fail}; \Box; \iota : \theta; \Box$, where θ is either α or *fail*, and $\Box$ is a (possibly empty) context. A sequence is called *non-redundant* if it is not redundant.

(3) β is called non-redundant if it does not contain redundant sequences.

The following is easy to prove.

LEMMA 1

In the specification *Processes*, for any $\beta \in T_{\text{PROCESS}}(X)$ there is a unique non-redundant PROCESS term equal to it.

DEFINITION 25 (EXTENSION OF ρ_f TO PROCESS TERMS)

Extend the definition of ρ_f as follows.

$$\rho_f : T_{\text{PROCESS}}(X) \to (\Sigma \to (PW \to \mathcal{P}(PW)))$$

is defined inductively by

(1) If $\beta \in T_{\text{ACTION}}(X)$, then use the definition for STEP terms.

(2) If β is of the form $\iota : (\beta_1 \oplus \beta_2)$, then use the corresponding definition for STEP terms.

(3) Otherwise, let $\hat{\beta}$ be the unique non-redundant process term equal to β, and let $\hat{\beta} = \beta_1; \beta_2$ (any arbitrary decomposition will do). Then by the inductive build-up, $\rho(\beta_i)(\sigma)$ is a function $PW \to \mathcal{P}(PW)$, for $i = 1, 2$. Then define $\rho_f(\beta)(\sigma)(w) = \{f_2 \circ f_1 \mid f_i \in \rho_f(\beta_i)(\sigma), i = 1, 2\}$, where $\circ$ is the usual function composition lifted to sets.

THEOREM 7

For any dynamic constraint specification $Spec_{Dyn}$ that includes *Processes* as a process specification, $\mathcal{M}_f$ is a model.

THEOREM 8

The following formulas are true in $\mathcal{M}_f$.

(1) $[\beta_1; \beta_2]\Phi \leftrightarrow [\beta_1]([\beta_2]\Phi)$.

(2) $\langle\beta_1; \beta_2\rangle\Phi \leftrightarrow \langle\beta_1\rangle(\langle\beta_2\rangle\Phi)$.

(3) $\mathbf{F}(\beta_1; \beta_2) \leftrightarrow [\beta_1]\mathbf{F}(\beta_2)$.

(4) $\mathbf{P}(\beta_1; \beta_2) \leftrightarrow \langle\beta_1\rangle\mathbf{P}(\beta_2)$.

Formulas (1) and (2) are standard theorems of dynamic logic [34]. They follow immediately from the definition of sequence as function composition.

Formula (3) says that a sequence is forbidden iff the prohibition will remain in force when we start performing the sequence, and a violation is raised when the whole sequence is performed. Secondly, a sequence is permitted iff after performing an initial part we may reach a state in which the last part is permitted. Clause (4) may be viewed as a paradox, because it says in effect that the goal (β_2) justifies

any means (β_1) [32]. This paradox can be easily removed by introducing *iterated permission* as follows.

DEFINITION 26 (ITERATED PERMISSION)

Iterated permission of a sequence is inductively defined as follows.

- $\mathbf{P}^*(\alpha) \stackrel{\text{def}}{\Leftrightarrow} \mathbf{P}(\alpha)$.
- $\mathbf{P}^*(\beta_1 + \beta_2) \stackrel{\text{def}}{\Leftrightarrow} \mathbf{P}^*(\beta_1) \vee \mathbf{P}^*(\beta_2)$.
- $\mathbf{P}^*(\beta_1 \,\&\, \beta_2) \stackrel{\text{def}}{\Leftrightarrow} \mathbf{P}^*(\beta_1) \wedge \mathbf{P}^*(\beta_2)$.
- $\mathbf{P}^*(\beta_1; \beta_2) \stackrel{\text{def}}{\Leftrightarrow} \mathbf{P}^*(\beta_1) \wedge \langle \beta_1 \rangle \mathbf{P}^*(\beta_2)$.

This preserves the properties of **P** but the last clause eliminates the problem that a permitted goal justifies any means. We will use iterated permission to solve the paradox of free choice permission in the model of internal choice given below.

6.2. MODEL 2: A MODEL FOR INTERNAL CHOICE

The state transition semantics that ρ_f gives to active choice concentrates on deontic effects of an action, i.e. on the effect of that an action has on the violation predicate. It takes the effect of a free choice $\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2)$ to be the intersection of the effects of $\iota : \alpha_1$ and $\iota : \alpha_2$. This is true for the deontic effect of an action, but it is false in general for the non-deontic effects of an action. What we would like is the following:

- The non-deontic effect of an active choice is defined compositionally in terms of the non-deontic effect of the branches. In particular, the non-deontic effect of an active choice should just be the union of the non-deontic effects of its branches, just as in the case of passive choice. Thus, if you choose, you choose and you do not perform both branches simultaneously.[1)]
- The deontic effect of an active choice is defined compositionally in terms of the deontic effect of its branches. In particular, we would like to define the deontic effect of performing the choice as the *conjunction* of the deontic effect of performing the branches. This corresponds to the intuition that if we have permission to choose between β_1 and β_2, then we have permission to choose β_1 and permission to choose β_2.

We can do this simply if we define active choice in terms of passive choice via equation (12), reproduced here:

[1)] This may be called *Tomas' Predicament*: In Milan Kundera's *The Unbearable Lightness of Being*, Tomas has problems choosing between Tereza and Sabina and then complains "If only I could live two lives simultaneously and then choose the best afterwards".

$$\iota_0{:}(\alpha_1 \oplus \alpha_2) = \underline{\iota_0{:}(\alpha_1 \leftarrow\!\oplus\ \alpha_2)}; \alpha_1 + \underline{\iota_0{:}(\alpha_1 \oplus\!\rightarrow\ \alpha_2)}; \alpha_2. \tag{12}$$

Remember that $\underline{\iota_0{:}(\alpha_1 \leftarrow\!\oplus\ \alpha_2)}$ and $\iota_0{:}(\alpha_1 \oplus\!\rightarrow\ \alpha_2)$ are the atomic actions of making a binary choice.

We extend *Processes* to *Pocesses*$^+$ by adding axiom (12).

DEFINITION 27 (INTERNAL CHOICE SEMANTICS)

The intended model $\mathcal{M}_i$ of a specification $Spec_{Dyn}$ that imports *Processes*$^+$ is defined in the same way as $\mathcal{M}_f$, except that

- we take the function ρ_i instead of ρ_f, where
- ρ_i is defined just as ρ_f (definition 23),
- except that we use the function choice$_i$ instead of choice, where
- choice$_i$, in turn, is defined just as choice (definition 21), but clauses (7) and (8), giving a semantics to active choice, are dropped.

Clauses (7) and (8) are not needed in this model because active choice is reduced to passive choice. The following is then trivial.

THEOREM 9

$\mathcal{M}_i$ is a model of any dynamic constraint specification $Spec_{Dyn}$ that includes *Processes*$^+$ as a process specification.

To solve the paradox of free choice permission using (12), we need two axioms that we call the *axioms of free choice permission*:

$$\mathbf{P}(\underline{\iota : (\iota : \alpha_1 \oplus\!\rightarrow\ \iota : \alpha_2)}) \leftrightarrow \mathbf{P}(\iota : \alpha_1) \wedge \mathbf{P}(\iota : \alpha_2), \tag{16}$$

$$\mathbf{P}(\underline{\iota : (\iota : \alpha_1 \leftarrow\!\oplus\ \iota : \alpha_2)}) \leftrightarrow \mathbf{P}(\iota : \alpha_1) \wedge \mathbf{P}(\iota : \alpha_2). \tag{17}$$

An actor is permitted to choose between two actions iff he is permitted to perform both branches. This gives us that an active free choice is permitted iff he is permitted to perform either branch and after his choice, the chosen branch is still permitted, as stated in the following theorem.

THEOREM 10

If $Spec_{Dyn}$ imports *Processes*$^+$ and contains axioms (16) and (17), then

$$\mathbf{P}^*(\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2)) \leftrightarrow \mathbf{P}(\iota : \alpha_1) \wedge \mathbf{P}(\iota : \alpha_2) \wedge (\langle \underline{\iota : (\iota : \alpha_1 \leftarrow\!\oplus\ \iota : \alpha_2)} \rangle \mathbf{P}(\alpha_1)$$
$$\vee \langle \underline{\iota : (\iota : \alpha_1 \oplus\!\rightarrow\ \iota : \alpha_2)} \rangle \mathbf{P}(\alpha_2)).$$

Proof

$$\mathbf{P}^*(\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2))$$

$$\leftrightarrow (\mathbf{P}^*(\underline{\iota : (\alpha_1 \leftarrow\!\!\oplus\ \alpha_2)}) \wedge \langle \underline{\iota : (\alpha_1 \leftarrow\!\!\oplus\ \alpha_2)}\rangle \mathbf{P}^*(\alpha_1)) \vee \qquad (12) \text{ and}$$

$$(\mathbf{P}^*(\underline{\iota : (\alpha_1 \oplus\!\!\rightarrow\ \alpha_2)}) \wedge \langle \underline{\iota : (\alpha_1 \oplus\!\!\rightarrow\ \alpha_2)}\rangle \mathbf{P}^*(\alpha_2)) \qquad \text{def. of } \mathbf{P}^*$$

$$\leftrightarrow (\mathbf{P}(\alpha_1) \wedge \mathbf{P}(\alpha_2) \wedge \langle \underline{\iota : (\alpha_1 \leftarrow\!\!\oplus\ \alpha_2)}\rangle \mathbf{P}(\alpha_1)) \vee \qquad (16), (17)$$

$$(\mathbf{P}(\alpha_1) \wedge \mathbf{P}(\alpha_2) \wedge \langle \underline{\iota : (\alpha_1 \oplus\!\!\rightarrow\ \alpha_2)}\rangle \mathbf{P}(\alpha_2))$$

$$\leftrightarrow \mathbf{P}(\iota : \alpha_1) \wedge \mathbf{P}(\iota : \alpha_2) \wedge \qquad \text{prop. logic}$$

$$(\langle \underline{\iota : (\iota : \alpha_1 \leftarrow\!\!\oplus\ \iota : \alpha_2)}\rangle \mathbf{P}(\alpha_1) \vee \langle \underline{\iota : (\iota : (\alpha_1 \oplus\!\!\rightarrow\ \iota : \alpha_2)}\rangle \mathbf{P}(\alpha_2)). \qquad \square$$

This result is intuitively plausible, for it says that

> it is possible that an active choice leads to a permitted state of the world

iff

> each of the branches can lead to a permitted state of the world and after at least one internal choice, the chosen branch can still lead to a permitted world.

The theorem can be simplified if we assume that a permission cannot be created by an internal choice,

$$([\underline{\iota_0 : (\alpha_1 \leftarrow\!\!\oplus\ \alpha_2)}]\mathbf{P}(\alpha)) \leftrightarrow \mathbf{P}(\alpha), \qquad (18)$$

$$([\underline{\iota_0 : (\alpha_1 \oplus\!\!\rightarrow\ \alpha_2)}]\mathbf{P}(\alpha)) \leftrightarrow \mathbf{P}(\alpha), \qquad (19)$$

for arbitrary α, α_1 and α_2. This is a plausible assumption that can be motivated by an appeal to intuition. These formulas imply their duals (using $\langle \cdot \rangle$ instead of $[\cdot]$), and these allow us to simplify theorem 10 as follows.

THEOREM 11

If $Spec_{Dyn}$ imports $Processes^+$ and contains axioms (16), (17), (18), and (19), then

$$\mathbf{P}(\iota : (\iota : \alpha_1 \oplus \iota : \alpha_2)) \leftrightarrow \mathbf{P}(\iota : \alpha_1) \wedge \mathbf{P}(\iota : \alpha_2). \qquad (20)$$

The proof of this is immediate. The assumptions on which this result is based can be summarized as follows.

(1) There is a difference between active and passive choice.

(2) Active choice does not create permission that was not already there.

(3) Active free choice is permitted iff the actor making the choice is permitted to choose both branches.

There is no magic here; as in any formal system, what we get out is what we put in. However, what we put in can be judged reasonable on independent grounds, and we have developed a formalism in which we can state precisely what we want to put in, without getting counterintuitive results like the paradox of free choice permission.

We should perhaps add that parts (1)–(4) of theorem 6 are still valid in $\mathcal{M}_i$. For example, we have

$$\mathbf{P}(\alpha_1 + \alpha_2) \leftrightarrow \mathbf{P}(\alpha_1) \vee \mathbf{P}(\alpha_2).$$

Theorem 20 defines the deontic effect of an active choice compositionally as the conjunction of the deontic effect of the branches. The non-deontic effect of the active choice is still defined in terms of the non-deontic effect of performing its branches, as stated in the following theorem.

THEOREM 12

If $Spec_{Dyn}$ imports $Processes^+$, then

$$[\iota_0 : (\alpha_1 \oplus \alpha_2)]\Phi \leftrightarrow [\underline{\iota_0 : (\alpha_1 \leftarrow\!\oplus\ \alpha_2)}][\alpha_1]\Phi \vee [\underline{\iota_0 : (\alpha_1 \oplus\!\rightarrow\ \alpha_2)}]\ [\alpha_2]\Phi.$$

This holds even without the axioms of free choice permission.

7. Discussion

7.1. NONDETERMINISM AND INITIATIVE IN CCS AND CSP

De Nicola and Hennessy [39] give a translation function **tr** from CCS to TCCS which deletes all occurrences of τ and replaces choice by internal or external choice, depending upon the first event of the chosen branches. This translation makes sense if we restrict ourselves to a two-actor system, where one actor (the observer o) initiates all atomic events and either actor (o or the machine m) can initiate a choice. Example translations (in our actor formalism) are

(1) $\mathbf{tr}(a + b) = o : (o : a \oplus o : b)$,

(2) $\mathbf{tr}(\tau; a + b) = m : (o : (o : a \oplus o : b) \oplus o : a)$,

(3) $\mathbf{tr}(\tau; a + \rho; b) = m : (o : (o : a \oplus o : b) \oplus o : a \oplus o : b)$.

The last translation assumes choice is associative, which we saw earlier is not obvious in an actor-oriented specification. Making actors explicit shows that there are more ways to interpret τ. For example, the translation in (2) says that m chooses between letting o do a or giving o the choice to do a or b. A translation that stays closer to the original would be

(2′) $\mathbf{tr}(\tau; a + b) = o : ((m : c\,;\, o : a) \oplus o : b)$

for an event c, and assuming CCS choice is external. One can argue that o is not able to make the choice on the right-hand side of (2′) because o does not know what m will do, but then neither is m on the right-hand side of (2) able to make a choice. Perhaps the problem in (2) is that the initiative of the choice lies partly with m and partly with o (if o is fast enough, he can press the button before m does τ).

This problem is also present in (3), where the intention is that the choice is made by m. A simpler translation would therefore be

(3′) $\mathbf{tr}(\tau; a + \tau; b) = m : (o : a \oplus o : b).$

Using internal choices, this becomes

(3″) $\mathbf{tr}(\tau; a + \tau; b) = o : (m : \underline{\iota_0 : (\alpha_1 \leftarrow\!\oplus\ \alpha_2)}; o : a \oplus m : \underline{\iota_0 : (\alpha_1 \oplus\!\rightarrow\ \alpha_2)}; o : b).$

Using actors, one is forced to make explicit which choice one makes.

The problem that an actor cannot choose if he has not enough information is illustrated neatly by a number of CSP laws [22, pp. 103–107]. Translated into process terms with actors, these are:

(1) $m : (o : a\,;\, x \oplus o : a\,;\, y) = o : a\,;\, m : (x \oplus y),$

(2) $o : (o : a\,;\, x \oplus o : a\,;\, y) = o : a\,;\, m : (x \oplus y),$

(3) $m : (x \oplus o : (y \oplus z)) = o : (m : (x \oplus y) \oplus m : (x \oplus z)),$

and another one like (3), with the roles of o and m reversed. In (1), m cannot make the choice on the left-hand side and the initiative to do something lies with o. However, m retains initiative as far as the choice is concerned. In (2), o does not choose at all, but simply does a and passes control over the choice to m. It is not obvious why control should be passed to m in (2). Together, (1) and (2) imply that the initiative for a "truly" nondeterministic choice always lies with m. Thus, two forms of nondeterminism are identified: "true" nondeterminism of the $ax + ay$ kind, in which an event leads to an element of a set of possible next states, and lack of control over a choice, as in $\iota_0 : (\iota : \alpha_1 \oplus \alpha_2)$, where ι has no control over the choice but ι_0 has. We see nothing wrong in identifying these two forms of nondeterminism, but see no particular reason for making this identification either.

Formula (3) says that

> Mary chooses between x and giving Otto the choice between y and z

iff

> Otto chooses between offering Mary the choice between x and y and offering Mary the choice between x and z.

The problem with this is that the order of choices as well as what the actors choose from is different. Hoare [22, pp. 107–108] argues for this equation on the grounds that the effect of both sides is the same. We think these conflicting intuitions can be harmonized as follows. To do justice to the intuition that the processes are not equal, we simply delete axiom (3) from the process theory. This prevents them from being equal in $\mathcal{A}_{Process}$. However, to do justice to Hoare's intuition that the two processes have the same effect in *PW*, we could add

(3′) $[m : (x \oplus o : (y \oplus z))]\Phi \leftrightarrow [o : (m : (x \oplus y) + m : (x \oplus z))]\Phi$,

as an axiom to $Spec_{Dyn}$. ρ then assigns the same effect to these otherwise different processes.

7.2. ACTIVE OBJECTS

In our approach, the concept of a globally unique identifier for each actor is crucial. Identifiers are also crucial for objects [3,27], so we could try to unify the concepts of actor and object in that of an active object. Elsewhere [45,47], it is argued that in an algebraic specification framework, the only essential addition to be made to get an object-oriented specification language is the idea of *localization* of properties (often called local state, local instance variables, local attributes) and of events (often called method). Localization is often called encapsulation in object-orientation. We can bring this in quite easily by a number of syntactic restrictions. We briefly show how this can be done.

Let us call any sort for which $Spec_{Stat}$ or $Spec_{Dyn}$ defines functions ot predicates as *object idenitifer sort.* Properties are localized by allowing only unary functions and predicates to be declared for identifier sorts. We then call a function an *attribute* and can say of every attribute and (non-deontic) predicate that is has a single *subject.* For deontic predicates (obligation, permission, prohibition, violation), we call the actor of the event to which the predicate is applied the subject. Thus, the subject of *reservations*(b) is b, and of $\mathbf{O}(p : return(b))$ it is p. The predicate *Borrowed*(b, p) is inadmissible in this approach, because it has more than one argument. We can now think of each DB state as a set of tuples of the form $\langle oid, \langle a_1 : v_1, \ldots, a_n : v_n, P_1 : b_1, \ldots, P_m : b_m \rangle\rangle$, where a_i and P_j are all attributes and predicates applicable to the object identifier *oid*, v_i are values (possibly used as oid's elsewhere) and b_j are Boolean values.

Events can be localized by treating their first (non-actor) argument as the identifier of the object whose state is changed by the event. We call this the *subject* of the event. For example, in p : *return*(b), b is called the event subject. This means that we regard p : *return*(b) as an event in the life of b, even though it is initiated by p. Furthermore, we regard it as a local event in the life of b that may change the local state of b. To represent the fact that p initiates the event, we must define p : *return*(b) as part of a *communication event* in which p and b participate, each with a local event. Pursuing this idea would exceed the bounds of this paper, but this has been done in detail elsewhere [46,50,47] in the context of the specification language CMSL. See [42,43] for a related approach in the context of the specification language Oblog.

We call a constraint local if there is a single subject appearing in all subject argument positions of predicates, attributes and events. Static ICs may be local or global, but we require dynamic ICs (containing occurrences of [·]) to be local only, in order to enforce a local state to be changed by a local event only. We look at each of the constraints given in this paper to see whether they are local or global. The different parts of the specification are collected together in the appendix.

In *StaticLibraryConstraints*, constraint `[C1]` is local and in *DynamicLibrary-Constraints*, `[D1]` and `[D2]` are all local because all predicates, attributes and events occurring in them have one argument, and that argument is the same variable throughout each constraint. They are constraints on individual books. `[D2]` blocks the event *borrow* initiated by a person but suffered by a book, and by our decisions this is seen as an event in the life of a book (it may change the state of the book it is applied to) but not of a person (it cannot change the state of the person initiating the event). This makes clear that the decision to localize *all* events is too severe, for it disallows communication between objects. Some events are global in the sense that they are shared by objects. Localization can still be maintained by requiring a shared event to consist of the synchronous execution (composition by &) of a number of local events, and to prohibit defining the effect of a shared event other than by stating of which local events it is composed.

`[D0]` is not local, because it uses a predicate that is not local. We fix this below. `[N0]` and `[N2-4]` are all local rules applicable to persons. `[N1]` and `[N5]` are not local, because of their use of the binary predicate *Borrowed*. `[N6]` is global static IC with three subjects, a book, a person, and a library. An example of how to make some of these constraints local is

deontic constraint spec *ObjectOrientedLibrarySpec*
attributes
reservations : *BOOK* → *QUEUE*
last_borrower : *BOOK* → *PERSON*
predicates
Present : *BOOK*
Borrowed : *BOOK*

constraints

[SO′] $Present(b) \leftrightarrow \neg Borrowed(b)$

[DO′] $[p : borrow(b)]Borrowed(b) \wedge last\_borrower(b) = p$

[D3′] $[p : return(b)]Present(b) \wedge \neg Borrowed(b)$

[N1′] $[p : borrow(b)]$ [**clock**(21)]

$(Borrowed(b) \wedge last\_borrower(b) = p \rightarrow V : p : -return(b))$

deontic constraint spec *ObjectOrientedLibrarySpec*

[N1′] is about persons, and in addition contains a reference to the clock. Probably, we should allow clock events to appear in any rule. [N1′] ignores the possibility that the same person returned and borrowed the book for the second time within three weeks. This can be fixed if we monitor the process of borrowing and returning more closely. We omit this for reasons of space.

It is important to realize that the important concept of *encapsulation* in object-oriented modeling has several different formalizations that are not equivalent. *Textual* encapsulation is the hiding of names declared in one part of a specification from visibility in other parts of a specification. This is handled adequately by the import mechanism common in algebraic specifications, possibly supplemented with a name-hiding mechanism. *Semantic* encapsulation is the localization of state and behavior in individual objects, so that each object has a local state that is only changed by local events. Localization of state is enforced by the requirement that attributes and predicates are unary, and localization of state changes is enforced by the requirement that dynamic constraints, which define the effect of events on objects, are local. The concept of encapsulation in object-oriented specification is discussed at length in [47].

In this approach, the concept of *active object* is formalized as an actor that is also an object. An active object need not be active all the time. Rather, activity is a relation between an actor and an event occurrence, and actors may perform or suffer events. However, only actors can perform events, and non-actors can only suffer them.

8. Summary and conclusions

We gave a semantic structure for dynamic logic with equality that separates the algebraic semantics $\mathcal{A}_{Process}$ of uninterpreted process terms from the state transition semantics ρ of the effect that processes have on the world. This allowed us to experiment with different state transition semantics, keeping the algebraic process semantics invariant. We gave a sound and complete inference system for this version of dynamic logic.

In addition to the semantic structure mentioned above, special features of our version of dynamic logic are the absence of iteration as a process operator and the introduction of the synchronous execution and action negation operators, as well as the explicit specification of these in algebraic specifications.

The extension of dynamic logic to deontic logic builds on earlier work done by Meyer [34], and the use of active and passive choice to solve the paradox of free choice permission builds upon the earlier idea to use the CSP distinction between internal and external choice to solve this paradox [33]. The extension of the specification language given in this paper to object-oriented specification is motivated in more detail in Wieringa [45,47].

As a preliminary to implementing parts of the language, current work includes giving an operational semantics to sublanguages of the specification language used in this paper. A topic high on our priority list is the exploration of different semantics of action negation and in general of more interesting process semantics than the initial semantics. Extension of dynamic logic to recursively defined processes and to processes with communication will also be investigated.

Appendix: The example specification

value spec $PersonIdentifiers$
  import
    $Booleans$
  sorts
    $PERSON$
  functions
    $p_0 : PERSON$
    $next : PERSON \longrightarrow PERSON$
    $eq : BOOK \times BOOK \longrightarrow BOOL$
  variables
    $x, x_1, x_2 : PERSON$
  equations
[1] $x \ \ eq \ \ x = true$
[2] $p_0 \ \ eq \ \ next(x) = false$
[3] $next(x) \ \ eq \ \ p_0 = false$
[4] $next(x_1) \ \ eq \ \ next(x_2) = x_1 \ \ eq \ \ x_2$
end spec $PersonIdentifiers$

value spec $ActorIdentifiers$
  import
    $PersonIdentifiers$, $LibraryIdentifiers$
  sorts
    $PERSON \leq ACTOR$,
    $LIBRARY \leq ACTOR$
end spec $ActorIdentifiers$

process spec *LibraryEvents*
import
$PersonIdentifiers$, $BookIdentifiers$, $LibraryIdentifiers$, $Money$
sorts
$PERSON\_EVENT$
$LIBRARY\_EVENT$
functions
$borrow : BOOK \longrightarrow PERSON\_EVENT$
$return : BOOK \longrightarrow PERSON\_EVENT$
$reserve : BOOK \longrightarrow PERSON\_EVENT$
$pay : MONEY \longrightarrow PERSON\_EVENT$
$notify : PERSON \times BOOK \longrightarrow LIBRARY\_EVENT$
end spec *LibraryEvents*

process spec *Actions*
import

$LibraryActions$, $LibraryActors$
sorts
$PERSON\_EVENT \leq EVENT$
$LIBRARY\_EVENT \leq EVENT$
$ATOMIC\_ACTION \leq ACTION$
functions
$any : EVENT$
$fail : EVENT$
$\_\oplus\_ : ACTION \times ACTION \rightarrow EVENT$
$\_:\_ : PERSON \times PERSON\_EVENT \longrightarrow ATOMIC\_ACTION$
$\_:\_ : LIBRARY \times LIBRARY\_EVENT \longrightarrow ATOMIC\_ACTION$
$\_:\_ : ACTOR \times EVENT \longrightarrow ACTION$
$\_+\_ : ACTION \times ACTION \longrightarrow ACTION$
$\_\&\_ : ACTION \times ACTION \longrightarrow ACTION$
$-\_ : ACTION \longrightarrow ACTION$
variables
$\iota, \iota_0, \iota_1, \iota_2 : ACTOR$
$\alpha : EVENT$
$\boldsymbol{\alpha}, \boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \boldsymbol{\alpha}_3 : ACTION$
equations
[PC1] $\boldsymbol{\alpha}_1 + \boldsymbol{\alpha}_2 = \boldsymbol{\alpha}_2 + \boldsymbol{\alpha}_1$
[PC2] $(\boldsymbol{\alpha}_1 + \boldsymbol{\alpha}_2) + \boldsymbol{\alpha}_3 = \boldsymbol{\alpha}_1 + (\boldsymbol{\alpha}_2 + \boldsymbol{\alpha}_3)$
[PC3] $\boldsymbol{\alpha} + \boldsymbol{\alpha} = \boldsymbol{\alpha}$

[N1] $- \ -\alpha = \alpha$
[N2] $-(\alpha_1 + \alpha_2) = -\alpha_1 \& -\alpha_2$
[N3] $-(\alpha_1 \ \& \ \alpha_2) = -\alpha_1 + -\alpha_2$

[D] $\alpha \ \& \ (\alpha_1 + \alpha_2) = (\alpha \ \& \ \alpha_1) + (\alpha \ \& \ \alpha_2)$

[ANY1] $\iota{:}\alpha + \iota{:}any = \iota{:}any$
[ANY2] $\iota{:}\alpha \ \& \ \iota{:}any = \iota{:}\alpha$

[AC1] $\iota{:}(\alpha_1 \oplus \alpha_2) = \iota{:}(\alpha_2 \oplus \alpha_1)$
end spec *Actions*

process spec *Processes*
import
Actions
sorts
$EVENT \leq CHOICES$
$ACTION \leq PROCESS$
functions
$\_;\_ : PROCESS \times PROCESS \longrightarrow PROCESS$
$\_+\_ : PROCESS \times PROCESS \longrightarrow PROCESS$
$\_\oplus\_ : PROCESS \times PROCESS \longrightarrow CHOICES$
$\_:\_ : ACTOR \times CHOICES \longrightarrow PROCESS$
variables
$\iota : ACTOR$
$\alpha : ACTION$
$\beta, \beta_1, \beta_2, \beta_3 : PROCESS$
equations
[S1] $(\beta_1;\beta_2);\beta_3 = \beta_1;(\beta_2;\beta_3)$

[PC4] $\beta_1 + \beta_2 = \beta_2 + \beta_1$
[PC5] $(\beta_1 + \beta_2) + \beta_3 = \beta_1 + (\beta_2 + \beta_3)$
[PC6] $\beta + \beta = \beta$

[SYN1] $(\alpha_1;\beta_1)\&(\alpha_2;\beta_2) = (\alpha_1\&\alpha_2);(\beta_1\&\beta_2)$
[SYN2] $\alpha_1\&(\alpha_2;\beta_2) = (\alpha_1\&\alpha_2);\beta_2$
[SYN3] $(\alpha_1;\beta_1)\&\alpha_2 = (\alpha_1\&\alpha_2);\beta_1$

[D1] $\beta_1;\beta_3 + \beta_2;\beta_3 = (\beta_1 + \beta_2);\beta_3$
[D2] $\beta_3;\beta_1 + \beta_3;\beta_1 = \beta_3;(\beta_1 + \beta_2)$
[D3] $\iota{:}(\beta_1;\beta_3 \oplus \beta_2;\beta_3) = \iota{:}(\beta_1 \oplus \beta_2);\beta_3$

[S2] $\iota{:}(\beta_1 \oplus \beta_2) = \iota{:}(\beta_2 \oplus \beta_1)$

[FAIL1] $\iota{:}fail; \iota{:}\beta = \iota{:}fail$
[FAIL2] $\iota{:}fail; \beta; \iota{:}\beta = \iota{:}fail; \beta$

[AC2] $\iota{:}(\beta_1 \oplus \beta_2) = \iota{:}(\beta_2 \oplus \beta_1)$
end spec *Processes*

static constraint spec *StaticLibraryConstraints*
import
Persons, *BookIdentifiers*, *Queues*
functions
$reservations : BOOK \longrightarrow QUEUE$
predicates
$Reserved : BOOK$
$Available : BOOK$
$Present : BOOK$
variables
$b : BOOK$
static constraints
[C1] $Available(b) \leftrightarrow Present(b) \wedge \neg Reserved(b)$
end spec *StaticLibraryConstraints*

dynamic constraint spec *DynamicLibraryConstraints*

import
StaticLibraryConstraints, *Processes*
variables
$p : PERSON$
$b : BOOK$
$q : QUEUE$
dynamic constraints
[D0] $[p{:}borrow(b)]Borrowed(b, p) \wedge \neg Present(b)$
[D1] $Reserved(b) \rightarrow$
$(p = head(q) \wedge q = reservations(b) \rightarrow$
$[p{:}borrow(b)]reservations(b) = tail(q))$

[D2] $Reserved(b) \rightarrow$
$(\neg(p = head(q) \wedge q = reservations(b)) \rightarrow [p{:}borrow(b)]false)$
end spec *DynamicLibraryConstraints*

deontic constraint spec *DeonticLibraryConstraints*
import
DynamicLibraryConstraints
variables
$p : PERSON$
$b, b' : BOOK$
$l : LIBRARY$
deontic constraints
[N0] $\mathbf{P}(p{:}borrow(b)) \leftrightarrow$
$Member(p) \wedge \neg V{:}p{:} - return(b')$
[N1] $[p{:}borrow(b)][\mathbf{clock}(21)](Borrowed(b,p) \rightarrow$
$V{:}p{:} - return(b))$
[N2] $V{:}p{:} - return(b) \leftrightarrow$
$[p{:}return(b)](V{:}p{:}return(b) \wedge \neg V{:}p{:} - return(b))$
[N3] $[p{:}pay(\$2, b)]\neg V{:}p{:}return(b)$
[N4] $[p{:}borrow(b)]\mathbf{O}(p{:}return(b), < 21)$
[N5] $[p{:}borrow(b)][\mathbf{clock}(21)](Borrowed(b,p) \leftrightarrow \mathbf{O}(l{:}remind(p,b))$
[N6] $Present(b) \wedge Reserved(p) \leftrightarrow$
$(p = front(reservations(b))\mathbf{O}(l{:}notify(p,b)))$
end spec *DeonticLibraryConstraints*

Acknowledgements

Earlier versions of this work were presented at the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91) in Brighton and at the 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems (MFDBS'91) in Rostock. We thank the referees of those papers for their constructive comments. This work was also presented at a meeting of the Esprit working group COMPASS in Braunschweig. We thank the participants of that meeting, especially Professors Hans-Dieter Ehrich and Hartmut Ehrig, who pointed out our nonstandard use of the terminology of conservative extensions. The concept of internal choice was discussed with Martin Sadler of Hewlett–Packard Laboratories, who suggested that the idea might not be as bad as it first seemed. Paul Spruit eliminated a number of errors in an earlier version of the paper. Thanks are due to the anonymous referee who gave detailed and constructive comments on the paper, which led to some significant improvements.

References

[1] A. al Hibri, *Deontic Logic* (University Press of America, 1978).
[2] L. Åqvist, Deontic logic, in: *Handbook of Philosophical Logic II*, ed. D.M. Gabbay and F. Guenthner (Reidel, 1984) pp. 605–714.

[3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik, The object-oriented database system manifesto, in: *1st Int. Conf. on Deductive and Object-Oriented Databases*, ed. W. Kim, J.-M. Nicolas and S. Nishio (1989) pp. 40–57.

[4] J.C.M. Baeten and W.P. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science 18 (Cambridge University Press, 1990).

[5] J.A. Bergstra and J.W. Klop, Process algebra for synchronous communication, Info. Control. 60(1984)109–137.

[6] J.A. Bergstra and J.W. Klop, Algebra of communicating processes with abstraction, Theor. Comput. Sci. 37(1985)77–121.

[7] J.A. Bergstra and J.W. Klop, Algebra of communicating processes, in: *Mathematics and Computer Science*, ed. J.W. de Bakker, M Hazewinkel and J.K. Lenstra, CWI Monographs 1 (North-Holland, 1986) pp. 89–138.

[8] H.-N. Casteñeda, The paradoxes of deontic logic, in: *New Studies in Deontic Logic* (Reidel, 1981).

[9] D.T. Sannella and D.B. MacQueen, Completeness of proof systems for equational specifications, IEEE Trans. Software Eng. SE-11(1985)454–461.

[10] F.P.M. Dignum and J.-J.Ch. Meyer, Negations of transactions and their use in the specification of dynamic and deontic integrity constraints, in: *Semantics for Concurrency*, ed. M.Z. Kwiatkowska, M.W. Shields and R.M. Thomas (Springer, 1990) pp. 61–80.

[11] H.-D. Ehrich, M. Gogolla and U.W. Lipeck, *Algebraische Spezifikation abstrakter Datentypen* (B.G. Teubner, 1989).

[12] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*, EATCS Monographs on Theoretical Computer Science, Vol. 6 (Springer, 1985).

[13] J. Fiadeiro and T. Maibaum, Temporal reasoning over deontic specifications, J. Logic Comput. 1(1991).

[14] D. Føllesdal and R. Hilpinen, Deontic logic: An introduction, in: *Deontic Logic: Introductory and Systematic Readings*, ed. R. Hilpinen (Reidel, 1971) pp. 1–35.

[15] L.T.F. Gamut, *Logic, Language and Meaning 2: Intensional Logic and Logical Grammar* (University of Chicago Press, 1991). L.T.F. Gamut is a pseudonym for J.F.A.K. van Benthem, J. Groenendijk, D. de Jongh, M. Stokhof and H. Verkuyl.

[16] R.J. van Glabbeek, Comparative concurrency semantics and refinement of actions, Ph.D. Thesis, Vrije Universiteit/Centrum voor Wiskunde en Informatica, Amsterdam (1990).

[17] R.J. van Glabbeek and F.W. Vaandrager, Modular specifications in process algebra with curious queues, in: *Algebraic Methods: Theory, Tools, and Applications*, ed. M. Wirsing and J.A. Bergstra, Lecture Notes in Computer Science 394 (Springer, 1989) pp. 465–506.

[18] J.A. Goguen and J. Meseguer, Completeness of many-sorted equational logic, SIGPLAN Notices 17(1982)9–17.

[19] J.A. Goguen and J. Meseguer, Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations, Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab (July 1989).

[20] D. Harel, Dynamic logic, in: *Handbook of Philosophical Logic II*, ed. D.M. Gabbay and F. Guenthner (Reidel, 1984) pp. 497–604.

[21] R. Hilpinen, Conditionals in possible worlds, in: *Contemporary Philosophy, A New Survey*, Vol. 1, ed. G. Fløstad (Reidel) pp. 299–335.

[22] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice–Hall, 1985).

[23] W. van der Hoek and J.-J.Ch. Meyer, Explicating some issues in implicit knowledge, Technical Report, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (September 1990).

[24] G.E. Hughes and M.J. Cresswell, *A Companion to Modal Logic* (Methuen, 1984).

[25] G. Kalinowski, *Einführung in die Normenlogik* (Athenäum Press, 1972).

[26] H. Kamp, Free choice permission, Aristotelian Soc. Proc. N.S. 74(1973–1974)57–74.

[27] S.N. Khoshafian and G.P. Copeland, Object identity, in: *Object-Oriented Programming Systems, Languages and Applications*, SIGPLAN Notices 22 (12) (1986) pp. 406–416.

[28] S. Khosla, System specification: A deontic approach, Ph.D. Thesis, Department of Computing, Imperial College, London (1988).

[29] S. Khosla and T.S.E. Maibaum, The prescription and description of state based systems, in: *Temporal Logic in Specification*, ed. B. Banieqbal, H. Barringer and A. Pnueli (Springer, 1987) pp. 243–294.

[30] D. Kozen and J. Tiuryn, Logics of programs, in: *Handbook of Theoretical Computer Science*, ed. J. van Leeuwen (Elsevier Science, 1990) pp. 789–840.

[31] R.M. Lee, Bureaucracies as deontic systems, ACM Trans. Office Info. Syst. 6(1988)87–108.

[32] R. van der Meyden, The dynamic logic of permission, in: *Proc. 5th IEEE Conf. on Logic in Computer Science*, Philadelphia (1990) pp. 72–78.

[33] J.J.Ch. Meyer, Free choice permissions and Ross' paradox: Internal vs external nondeterminism, Technical Report IR-130, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (August 1987).

[34] J.-J.Ch. Meyer, A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic, Notre Dame J. Formal Logic 29(1988)109–136.

[35] J.-J.Ch. Meyer, Using programming concepts in deontic reasoning, in: *Semantics and Contextual Expression*, ed. R. Bartsch, J.F.A.K. van Benthem and P. van Emde Boas (FORIS Publications, Dordrecht/Riverton, 1989) pp. 117–145.

[36] J.-J.Ch. Meyer and E. de Vink, Step semantics for "true" concurrency with recursion, Distr. Comput. 3(1989)130–145.

[37] J.-J.Ch. Meyer and R.J. Wieringa, Actor-oriented system specification with dynamic logic, in: *Proc. Int. Joint. Conf. on Theory and Practice of Software Development (TAPSOFT'91)*, Vol. 2, ed. S. Abramsky and T.S.E. Maibaum, Lecture Notes in Computer Science 494 (Springer, 1991) pp. 337–357.

[38] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science (Springer, 1980).

[39] R. de Nicolas and M. Hennessy, CCS without τ's, in: *Proc. Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, Vol. 1, ed. H. Ehrig, R. Kowalski, G. Levi and U. Montanari, Lecture Notes in Computer Science 249 (Springer, 1987) pp. 138–152.

[40] R. Reiter, Equality and domain closure in first-order databases, J. ACM 27(1980)235–249.

[41] R. Reiter, Towards a logical reconstruction of relatitional database theory, in: *On Conceptual Modelling*, ed. M.L. Brodie, J. Mylopoulos and J.W. Schmidt (Springer, 1984) pp. 191–233.

[42] A. Sernadas and H.-D. Erhich, What is an object, after all?, in: *Object-Oriented Databases (DS-4)*, Windermere, UK (July 1990), IFIP Working Group 2.6.

[43] A. Sernadas, J. Fiadeiro, C. Sernadas and H.-D. Ehrich, The basic building blocks of information systems, in: *Information System Concepts: An In-Depth Analysis*, ed. E.D. Falkenberg and P. Lindgreen (North-Holland, 1989) pp. 225–246.

[44] G. Smolka, W. Nutt, J. Goguen and J. Meseguer, Order-sorted equational computation, in: *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, ed. M. Nivat and H. Aït-Kaci (Academic Press, 1989) pp. 297–367.

[45] R.J. Wieringa, Algebraic foundations for dynamic conceptual models, Ph.D. Thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (May 1990).

[46] R.J. Wieringa, Equational specification of dynamic objects, in: *Object-Oriented Databases: Analysis, Design, and Construction (DS-4)*, ed. R.A. Meersman, W. Kent and S. Khosla (North-Holland, 1991) pp. 415–438.

[47] R.J. Wieringa, A formalization of objects using equational dynamic logic, in: *2nd Int. Conf. on Deductive and Object-Oriented Databases*, ed. C. Delobel, M. Kifer and Y. Masunaga, Lecture Notes in Computer Science 566 (Springer, 1991) pp. 431–452.

[48] R.J. Wieringa, J.-J. Ch. Meyer and H. Weigand, Specifying dynamic and deontic integrity constraints, Data Knowledge Eng. 4(1989)157–189.

[49] R.J. Wieringa and J.-J.Ch. Meyer, Actor-oriented specification of dynamic and deontic integrity constraints, in: *3rd Symp. on Mathematical Fundamentals of Database and Knowledge Base Systems (MFDBS 91)*, ed. B. Thalheim, J. Demetrovics and H.-D. Gerhardt, Lecture Notes in Computer Science 495 (Springer, 1991) pp. 89–103.

[50] R.J. Wieringa and R.P. Van De Riet, Algebraic specification of object dynamics in knowledge base domains, in: *Artificial Intelligence in Databases and Information Systems (DS-3)*, ed. R.A. Meersman, Zhongshi Shi and Chen-Ho Kung (North-Holland, 1990) pp. 411–436.

[51] R.J. Wieringa, H. Weigand, J.-J.Ch. Meyer and F. Dignum, The inheritance of dynamic and deontic integrity constraints, Ann. Math. Art. Int. 3(1991)393–428.

[52] G.H. von Wright, An essay in deontic logic and the general theory of action, Acta Philosophica Fennica, Fasc. 21 (North-Holland, 1968).