Logic of Change: Semantics of Object Systems with Active Relations

I.Bider (IbisSoft, Stockholm, Sweden, email: ilia@ibissoft.se)
M.Khomyakov (Magnificent Seven, Moscow, Russia, email: ari@online.ru)
E.Pushchinsky† (IBI, New-York, NY)

Abstract. In traditional approaches to object-oriented programming, objects are ”active”, while relations between them are ”passive”. The activeness of an object reveals itself when the object invokes a method (function) as a reaction on a message from another object (or itself). While this model is suitable for some tasks, like arranging interactions between windows, widgets and the end-user in a typical GUI environment, it’s not appropriate for others. Business applications development is one of the examples. In this domain, relations between conceptual objects are at least as important as objects themselves, and the more appropriate model for this field would be the one where relations are ”active” while objects are ”passive”. A version of such a model is presented in the paper. The model considers a system as consisting of a set of objects**,** a code of laws**,** and a set of connectors, each connector hanging on a group of objects that must obey a certain law. Formal logical semantics of this model is presented as a way of analyzing the set of all possible trajectories of all possible systems. The analysis allows to differentiate valid trajectories from invalid ones. The procedural semantics is presented as a state machine that given an initial state, generates all possible trajectories that can be derived from this state. This generator can be considered as a model of a connectors scheduler that allows various degrees of parallelism, from sequential execution to the maxim possible parallelism. In conclusion, a programming language that could be appropriate for the proposed computer environment is discussed.

1. Motivation

Any object-oriented system can be viewed as consisting of objects and relations between them. Relations are usually ”passive”, while objects are ”active”. The latter interact by exchanging messages along the established links (relations), each message triggering the invocation of a method in the receiver. The object model of this kind was developed in the field of object-oriented programming, and it fits the systems where objects cooperate in order to solve some computational tasks.

However, the model that stresses active objects doesn’t properly reflect the nature of other kinds of object systems, e.g., the ones that would be suitable for business processes modeling. All leading manuals on object-oriented analysis (see, for example, [1, p.153], [2, p. 61], [3, p. 301]) suggest searching for nouns when analyzing the business language of a particular domain. Nouns become objects, while verbs become methods inside those objects. The typical result of using the object-centered approach to business application design is an employee object that has a method of calculating it’s own wages, which doesn’t make any sense from the point of view of application domain.

The problems of applying the traditional object-centered approach to business processes were the subject of discussions at a number of workshops on process modeling (e.g., Workshop on Modeling Software Processes and Artifacts, ECOOP’97, and Workshop on Business Modeling for OO Systems, OOPSLA’97). The analysis of the business processes domain shows that in this field, relations are at least as important as objects themselves. For example, the calculation of wages should be considered as an "active" relation between the amount of money on the employee's account and the time he/she worked for the company, rather than a method inside the employee class. (Obviously, this is not the only relation that affects the amount of money on the employee's account.)

In the alternative object-oriented model presented in this paper, relations are active, while objects are (relatively) passive. The relationship between the proposed model and the traditional one may be compared to the relationship between the corpuscular and wave theories in the quantum mechanics. They are equivalent in the sense that they can be formally translated into one another. However, for certain computational tasks one theory fits better than the other. We consider our model to be a better fit for business processes modeling while it may be less suitable for other tasks, like defining interactions between windows, widgets and the end-user in a typical GUI environment.

The paper is devoted to discussing the formal semantics of a model with active relations, the experience of using this model (more precisely, its modification) for business software development is discussed elsewhere [4,5,6]. The approach used in our formalism is similar to the one adopted in the field of logic programming, see, for example, [7]. There is a number of works that extend logic programming to the object-oriented paradigm, see, for example [8], and a survey in [9]. They are devoted to developing formalisms, which facilitate deductive reasoning from the current state of the object-oriented database or the state sequence. Our aim is different, we concentrate on finding a formalism which helps to describe the behavior of a certain class of dynamic systems.

In our formalism, we use the notion of possible worlds that was developed in the field of modal logic starting with the works of Hintikka [10], and Kripke [11]. However, we don't use this notion for speculation about different truth values of a given formula in different worlds. We use it to formally differentiate valid worlds from invalid ones. Another difference is that we view a possible world not as a set of predicates that describes it, but as an infinite sequence of its states - the world's history. Philosophically, our dynamic worlds are close to the ones that are used in the versions of linguistic semantics based on the catastrophe theory [12], and the Montague logic [13].

The paper has the following structure. In section 2, we give a short informal description of the suggested model. In section 3, we present a formal logical semantics of it. In section 4, we discuss the model's procedural semantics without presenting the complete formalism of it. Section 5 is devoted to discussing main characteristics of a programming language suitable for our model. Section 6 contains a brief history of the project.

2. Informal Discussion

Our model is aimed at describing the behavior of complex dynamic systems which we present as consisting of:

- a set of ***objects,***
- a code of ***laws,***

- a set of ***connectors***, each connector hanging on a group of objects that must obey a certain law.

Connectors are not hanging in the air, they are included in the objects, more precisely in their bodies. This makes the objects complexly structured, the structure even permits a connector hanging on a particular object **to be included** in the body of this object.

Objects in the system constantly change. The connectors monitoring their operands, i.e. objects they hang on, react to these changes, restore order and ”fall asleep”. Connectors work concurrently and independently, maintaining the homeostasis of the unified system.

The *laws* are analogues to the predicates of logical programming. However, they differ from the latter in two important aspects:

- The truth values are defined on objects’ partial histories, not on the static states.
- If for some histories the law is not satisfied, it includes the action(s) of how to restore the order.

Connectors maintain local order in the system. As to global order, it is determined by the following principles:

- ***Causality***, which implies that every perturbation must have a reason.
- ***Preservation***, restricting types of possible changes.
- ***Inertness,*** requiring that before the order is restored according to the law, the connector’s operands shouldn’t change for a while.
- ***Limited lawlessness***, guaranteeing law enforcement in a world where laws are sometimes broken. This principle allows to settle conflict-of-interest cases between different connectors hanging on common objects.

Thus defined, our model has a reactive nature, i.e. changes in objects are made as a reaction to changes in other objects. However, this doesn’t mean that the system is purely reactive.

- First, our laws are not deterministic. Two kinds of nondeterminism are permitted:

  (a) If the law is broken, there can be more than one way of restoring it.
  (b) Even when the law is not broken, the action(s) can be defined that transfer the obeying object(s) into another (stable) state.

  Both (a) and (b) represent a ”window” into the external (for the system) world; (b) provides the possibility of change even if the whole system is in a stable state.

- Second, connectors that hang on an object and at the same time are included in its body make the object (pro)active. In our model, these connectors perform the task done by methods in OO programming languages. Changes in an operand of such connector serve as a message that invokes the connector and results in modification of the object’s state. This approach to making objects active is very powerful. Changes in the object’s state may result in reconfiguring its active part (this includes the possibility of removing the connector that introduces the changes). Next reaction to the same signal after reconfiguring may cause a completely different chain of actions inside (and outside) the object, than it has caused the first time.

A connector entrusted to guard a law over its operands is the main programming unit of our model. As such, it may be compared with a function in a traditional programming language, like C, or a method in an objet-oriented programming language, like Smalltalk, Java, or C++. The power of a programming language depends on the possibility of decomposing complex computations into the simpler ones, including the recursive decomposition. In traditional languages, this power is obtained through (recursive) procedure calls, or function invocations. In object-oriented languages, the same is gained by one object sending a message to another one, or itself (which may include direct or indirect recursion).

In our model, the connector guarding the law can't directly invoke (or send a message to) another connector. The philosophy of decomposition in the model is completely different, and it corresponds to the nature of the system. The task of the connector as a programming unit is not to do calculations, but watch its operands and restore order in them. Decomposition for this goal may mean that the "main" law requires that there should always be connectors that guard some "local" laws between (part of) the main connector's operands. If those connectors are absent, the main connector will add them. Then, it's up to the local connectors to ensure that the local laws are satisfied. In case a local law is equal to the main one, we'll get the recursive decomposition (see example in section 5).

A similar way of decomposition may be found in programming language REFAL [14,15]. REFAL has a notion of working area where pure data can be mixed with calls to executable functions. If one function "wants" to invoke another one, it can place a call to the second function in the working area (instead of direct invoking the latter). This call is activated later when the processor of working area reaches it.

It's worthwhile to mention that the above decomposition does not guarantee that the main law will be satisfied in the end. Changes introduced by other connectors can create a situation were all local laws of the given decomposition are never satisfied simultaneously. Decomposition guarantees only that if nothing else stays in the way, the main law will be satisfied in the end.

3. Logical Semantics - Definition of Valid Worlds

The logic way of formalizing the behavior of dynamic systems consists of analyzing the set of all possible trajectories of all possible systems. Below, we present an extract of our definitions that concern logical semantics. The definitions are structured in the normal for logic programming way (as outlined in [7]) with the reservation mentioned above in section 1. In defining the system state we follow an approach similar to the outlined in the club systems [16].

3.1 Alphabet

Alphabet *A* consists of:

(a) constants: *a, b, c, d, ...* (finite or infinite set)
(b) objects: *o1, o2, ...* (infinite set)
(c) predicate symbols: *p, q, ...* (finite set)
(d) punctuation symbols: "*(*", "*)*", ","
(e) embryo symbol: ⊥

Every constant is assigned one of the predefined types: τ,δ, ... (finite set). Every predicate is assigned arity *n*, and type $\tau 1 \times \tau 2 \times ... \times \tau k$, $k < n$, where τi is a constant type.

3.2 Connectors

Let:

(a) *p* be an *n*-ary predicate symbol of type $\tau 1 \times \tau 2 \times ... \times \tau k$,
(b) $x_1, ..., x_k$ be a *k*-tuple of constants, such that x_i is a constant of type τi, i = 1 ,..., *k*,
(c) $x_{k+1}, ..., x_n$ be an (*n-k*)-tuple of objects

then expression $c = p(x_1, ..., x_n)$ is called a (well-formed) **connector**.

Predicate symbol *p* is called a scheme of connector *c*, and is denoted as *scheme(c)*. $x_1, ..., x_n$ are called terms of connector *c,* and the set of terms is denoted as *terms(c).* The *i*-th term is denoted as *term(c,i)*, while position *i* occupied by term $x \in terms(c)$ is denoted as *pos(c,x).* Objects $x_{k+1}, ..., x_n$ are called operands of connector *c,* and the set of operands is denoted by *op(c).*

3.3 States

Let *O* be the set of all objects from alphabet *A*, *UC* be the set of all possible connectors, and *CAR =P(UC)* be the powerset of *UC.* Let *st* be a function

$st: O \rightarrow CAR \cup \{\perp\}$,

and *alive(st)* be the set of objects $o \in O$ for which $st(o) \neq \perp$. Then *st* is called a (well-formed) **state** if:

(a) *alive(st)* is a finite set,
(b) for any $o \in alive(st)$ and $c \in st(o)$, $op(c) \subseteq alive(st)$. ("Embryo can't have any relations with others")

Denote the set of all possible states as *UST.*

Definitions:

1. Let $c = p(x_1, ..., x_n)$ be a connector, and in state *st,* there is an object *o* such that $c \in st(o)$, then we'll say that connector *c exists* or *takes place* in state *st*. We also say that connector *c* hangs on its operands - objects from *op(c).*

2. Let $st \in UST$, and *s* be a finite subset of objects*,* and *d* be a nonnegative integer. Then the set of objects

$$guts(s, st, d) = \begin{cases} s \text{ , when } d = 0 \\ \{o \mid o \in guts(s, st, d\text{-}1), \text{ or there exists an object } o1 \in guts(s, st, d\text{-}1) \\ \text{and a connector } c \in st(o1) \text{ such that } o \in op(c)\}, \text{ when } d > 0 \end{cases}$$

is called *guts* of depth *d* of set *s* in state *st.*

3. Let $st \in UST$, and *d* is a positive integer. Then the restriction of function *st* on the subset *guts(s, st, d-1)* is called a *d*-depth body of set *s* in state *st*, and is denoted as *body(s, st).*

The notion of *d*-depth body is introduced in order to restrict the visibility of the connector's operands to a certain depth.

3.4 Worlds

Let *T* be a discrete time axes - the set of all nonnegative integers called time ticks. Consider a function

$w: T \text{ -> } UST,$

such that for any $t, t' \in T$, where $t' > t$: $alive(w(t)) \subseteq alive(w(t'))$. ("When born, never disappears.")

Let $t \in T$, $t > 0$, be a moment of time such that $w(t) \neq w(t\text{-}1)$, we'll call such moment an event. Denote the set of objects *o* that belong to *alive(w(t))*, but do not belong to *alive(w(t-1))* as *born(w,t)*:

$born(w,t) = \{o \mid o \in O, o \in alive(w(t) \text{ and } o \notin alive(w(t\text{-}1))\}$

Denote the set of objects *o* that belong to *alive(w(t-1))* as *changed(w,t)* if:

$changed(w,t) = \{o \mid o \in alive(w(t\text{-}1) \text{ and } w(t)(o) \neq w(t\text{-}1)(o)\}$

Let *d, h* be positive integers. A function *w: T -> UST* is called a (well-formed) **world** with depth *d* and inertness *h* if for any event *t* there exists a connector *c*, such that:

(a) c exists in state *w(t')* for all $t' \in [t\text{-}1, t\text{-}h]$ and $body(op(c), w(t'), d) = body(op(c), w(t\text{-}1), d)$ for all $t' \in [t\text{-}1, t\text{-}h]$

(b) changed$(w,t) \subseteq guts(op(c), w(t\text{-}1), d\text{-}1) \cap guts(op(c), w(t), d\text{-}1)$

(c) $guts(op(c), w(t), d) \subseteq guts(op(c), w(t\text{-}1), d) \cup born(w, t)$

(d) if $o \in born(w,t)$ and $o \notin guts(op(c), w(t), d\text{-}1)$ then $w(t)(o) = \varnothing$

This connector is called a *potential actor* of event t.

(a)-(d) formalize the principles of ***Causality***, ***Preservation***, and ***Inertness***:

- An event happens in the world only when one of the possible actors introduces changes in it's operands.
- Changes are localized to the d-depth bodies of the actor's operands (new objects may be born there).
- Before the change can happen, the d-depth bodies of the actor's operands should be the same during the predefined period of time (h).

The set of all possible worlds for given alphabet A, depth d, and inertness h is called an *(A,d,h)*-chaos language. Denote this set as $UW(A, d, h)$.

Definitions*:*

1. Let w be a world ($w \in UW(A,d,h)$), $z = <x1, ..., xn>$ be an n-tuple of objects, and t',t'' be time ticks ($t',t'' \in T$) such that $t' < t''$, and $\{x1, ..., xn\} \subseteq w(t')$. Then function $traj(w, z, t', t'')(t)$ defined on segment [t', t''] as:

$$traj(w, z, t', t'')(t) = body(\{x1, ..., xn\}, w(t), d)$$

is called a d-depth trajectory of tuple z in world w. Pair $<z, traj(w,z,t',t'')>$ is called a track; z is called the track's system, d is called the track's depth, and segment [t',t''] is called the track's observation period. The moments of time $t_1, ..., t_l$, $t' < t_1 < ,..., < t_l < t''$, such that $traj(w,z,t',t'')(t_i) \neq traj(w,z,t',t'')(t_i - 1)$, are called the track's events, and the number of events l is called the track's length.

Let tr be a track, then denote:

(a) its system as *sys(tr)*,
(b) its observation period as *period(tr)*,
(c) its ordered sequence of events as *events*(*tr*),
(d) its trajectory as *traj*(*tr*)

Denote the set of all possible tracks with depth d and length l in all possible worlds from $UW(A,d,h)$ as $UTR(A,d,h,l,n)$.

A track $tr \in UTR(A,d,h,l,n)$ such that $period(tr) = [0, t']$ is called a starting track. Denote the set of all possible starting tracks with the length less than or equal to l as $USTR(A,d,h,l,n)$. **Note**: If $l < 0$, then $USTR(A,d,h,l,n)$ is considered to be equal to $\varnothing$.

2. Let $tr1$, $tr2$ be tracks such that tr1, $tr2 \in UTR(A,d,h,l,n)$, $sys(tr1) = sys(tr2)$, and let $period(tr1) = [t',t'']$, $period(tr2) = [u', u'']$, $events(tr1) = <t_1, ..., t_l>$, and $events(tr2) = <u_1, ..., u_l>$. We'll call these two tracks event-equivalent if:

$traj(tr1)(t') = traj(tr2)(u'),\ traj(tr1)(t_1) = traj(tr2)(u_1),\ ...,\ traj(tr1)(t_l) = traj(tr2)(u_l),$
$traj(tr1)(t'') = traj(tr2)(u'').$

3. Let $tr1, tr2$ be tracks such that $tr1, tr2 \in UTR(A,d,h,l,n)$, and $period(tr1) = period(tr2)$. And let $sys(tr1) = <x_1,...,x_n>$, $sys(tr2) = <y_1,...y_n>$. We'll call these two tracks object-equivalent if there exists a one to one map of objects:

$$m: O \text{ -> } O$$

such that:

(a) $m(x_1) = y_1, ..., m(x_n) = y_n$,

(b) for any $t \in period(tr1)$:

$def(traj(tr2)(t)) = \{m(o) \mid o \in def(traj(tr1)(t))\}$,

where $def(f)$ is a set of objects for which function f is defined

(c) for any $t \in period(tr1)$, and $o \in def(traj(tr2)(t))$:

$traj(tr2)(t)(o) = \perp$, when $traj(tr1)(t)(o) = \perp$, or
$traj(tr2)(t)(o) = \{p(a_1,...,a_k,m(o_1),...,m(o_{n-k}) \mid$
$p(a_1,...a_k,o_1,...,o_{n-k}) \in traj(tr1)(t)(o'),\ where\ o = m(o')\}$, otherwise

4.Two tracks $tr1, tr2 \in UTR(A,d,h,l,n)$ are called equivalent, denote as $tr1 \Leftrightarrow tr2$, if they are object equivalent, or they are event equivalent, or if there exists a third track $tr3 \in UTR(A,d,h,l,n)$, such that $tr1$ and $tr3$ are event-equivalent, and $tr3$ and $tr2$ are object-equivalent.

5. Let $tr1$ be a track, $tr1 \in UTR(A,d,h,l,n)$, and let $[t',t''] = period(tr1)$, and $<e_1, ..., e_l> = events(tr1)$. Then track $tr2 \in UTR(A,d,h,l+1,n)$, such that $sys(tr2) = sys(tr1)$, $period(tr2) = [t', u'']$, where $u'' > t''$, $events(tr2) = < e_1, ..., e_l, e_{l+1}>$, $e_{l+1} < u''$, is called an extension of track $tr1$ if:

$$traj(tr2)(t) = traj(tr1)(t) \text{ for all } t \in period(tr1)$$

Denote the fact of extension as $tr2 >> tr$.

6. Let $tr1$ be a track, $tr1 \in UTR(A,d,h,l,n)$, $l > 0$, and let $[t',t''] = period(tr1)$, and $<e_1, ..., e_l> = events(tr1)$. Then track $tr2 \in UTR(A,d,h,l-1,n)$, $l > m > 0$, such that $sys(tr2) = sys(tr1)$, $period(tr2) = [u', t'']$, where $e_1 \le u' < e_2$, $events(tr2) = <e_2,...,e_l>$, is called a reduction of track $tr1$ if:

$$traj(tr2)(t) = traj(tr1)(t) \text{ for all } t \in period(tr2).$$

Denote the fact of reduction as $tr2 << tr1$.

The notions of track and tracks equivalence were introduced to restrict the amount of information considered by laws:

- The history is considered only for the last *l* changes of the connector's operands.
- The time between the events doesn't matter, only the logical sequence of them is taken into consideration.
- The law doesn't depend on particular objects, only the objects' structure matters.

3.5 Interpretations

Let l be a positive integer, then l-interpretation I of (A,d,h)-chaos language L consists of:

(a) for each type τ in A, a non-empty set D_τ, called the domain of type τ.

(b) for each constant a of type τ in A, an assignment $\mu(a)$ of an element from D_τ.

(c) for each n-ary predicate symbol of type $\tau 1 \times \tau 2 \times ... \times \tau k$, $k < n$, an assignment of two predicates of the form:

$$\pi_c: D_{\tau 1} \times ... \times D_{\tau k} \times (UTR(A,d,h,l\text{-}1,n\text{-}k) \cup USTR(A,d,h,l\text{-}2,n\text{-}k)) \text{ -> } \{true, false\},$$
$$\pi_a: D_{\tau 1} \times ... \times D_{\tau k} \times (UTR(A,d,h,l,n\text{-}k) \cup USTR(A,d,h,l\text{-}1,n\text{-}k)) \text{ -> } \{true, false\},$$

such that:

(1) for any constants $a_1, ..., a_k$ of types $\tau 1, \tau 2, ..., \tau k$ and any two tracks $tr1, tr2 \in UTR(A,d,h, l\text{-}1,n\text{-}k) \cup USTR(A,d,h,l\text{-}2,n\text{-}k)$, such that $tr1 \Leftrightarrow tr2$:

$$\pi_c(\mu(a_1), ..., \mu(a_k), tr1) = \pi_c(\mu(a_1), ..., \mu(a_k), tr2)$$

(2) for any constants $a1, ..., ak$ of types $\tau 1, \tau 2, ..., \tau k$ and any two tracks $tr1, tr2 \in UTR(A,d,h,l,n\text{-}k) \cup USTR(A,d,h,l\text{-}1,n\text{-}k)$ such that $tr1 \Leftrightarrow tr2$:

$$\pi_a(\mu(a_1), ..., \mu(a_k), tr1) = \pi_a(\mu(a_1), ...,\mu(a_k), tr2)$$

(3) Let $a_1, ..., a_k$ be constants of types $\tau 1, \tau 2, ..., \tau k$ and tr1 be a track, $tr1 \in UTR(A,d,h,l,n\text{-}k) \cup USTR(A,d,h,l\text{-}1,n\text{-}k)$ such that $\pi_a(\mu(a_1), ..., \mu(a_k), tr1) = true$. Then for each track $tr2 \in UTR(A,d,h,l\text{-}1,n\text{-}k) \cup USTR(A,d,h,l\text{-}2,n\text{-}k)$ such that $tr2 << tr1$:

$$\pi_c(\mu(a_1), ..., \mu(a_k), tr2) = true.$$

(4) Let $a_1, ..., a_k$ be constants of types $\tau 1, \tau 2, ..., \tau k$ and $tr1$ be a track, $tr1 \in UTR(A,d,h,l\text{-}1,n\text{-}k) \cup USTR(A,d,h,l\text{-}2,n\text{-}k)$ such that $\pi_c(\mu(a_1), ..., \mu(a_k), tr1) = false$. Then there exists a track $tr2 \in UTR(A,d,h,l,n\text{-}k) \cup USTR(A,d,h,l\text{-}1,n\text{-}k)$ such that $tr2 >> tr1$, and $\pi_a(\mu(a_1), ..., \mu(a_k), tr2) = true$.

Predicate π_c is called a condition predicate, while predicate π_a is called an action predicate, l is called a history length. Pair $<\pi_c,\pi_a>$ is called a meaning of predicate symbol p in interpretation I. Triple $<p, \pi_c, \pi_a>$ is called a law. Statement (1)-(4) above has the following "legal" explanation:

(1) All are equal before the law.
(2) The same punishment for the same offence.
(3) Actions result in satisfaction (restoring the law).
(4) There always is a way to restore the law.

Definitions:

1. Let w be a world, t'' be a time tick, and c be a connector $c=p(a_1, ..., a_{n-k}, x_1,...,x_k)$, where $a_1, ..., a_{n-k}$ are constants and $x_1,...,x_k$ - are objects, and c takes place in $w(t'')$. We say that c is *satisfied* at time t'' in world w in interpretation I, if there is a track $tr \in UTR(A,d,h,l-1,n-k) \cup USTR(A,d,h,l-2,n-k)$, where $sys(tr) = <x_1, ..., x_k>$, and $period(tr) = [t', t'']$, such that:

- $traj(tr)(t) = body(\{x_1, ..., x_k\}, w(t), d)$, for any $t \in [t',t'']$
- $\pi_c(\mu(a_1), ..., \mu(a_{n-k}), tr) = true$, were π_c is the condition predicate of p in I.

Otherwise we say that c is *unsatisfied* in world w at time t in interpretation I.

2. Let e be an event in world w.. We say that e is a legal event in interpretation I, if there exist

- a potential actor of this event $c=p(a_1, ..., a_{n-k}, x_1,...,x_k)$, where $a_1, ..., a_{n-k}$ are constants and $x_1,...,x_k$ - are objects, and
- a track $tr \in UTR(A,d,h,l,n-k) \cup USTR(A,d,h,l-1,n-k)$, where $sys(tr)=<x_1, ..., x_k>$, $period(tr)=[t', t'']$, and $events(tr)=<e_1, ..., e_{l-1},e>$,

such that:

- $traj(tr)(t) = body(\{x_1, ..., x_k\}, w(t), d)$, for any $t \in [t',t'']$, and
- $\pi_a(\mu(a_1), ..., \mu(a_{n-k}), tr) = true$, were π_a is the action predicate assigned to p in I.

3.6 Models

Let I be an l-interpretation of (A,d,h)-chaos language L, H be a positive number ($H >> h$), and $w \in W(A,d,h)$. We say that world w is H-valid in interpretation I, if:

(a) for any connector c, and a time interval $[t', t'']$, such that $t'' - t' \geq H$ and c take place in $w(t)$ for all $t \in [t', t'']$, there is at least one moment of time $t''' \in [t', t'']$, such that c is satisfied in w at t''' in interpretation I.
(b) any event in w is legal in interpretation I.

In (a), (b) above, (a) expresses the principle of *limited lawlessness*, while (b) strengthens the principle of causality - any action should be aimed at restoring one law.

L-interpretation I in which world w is H-valid, is called an (l,H)-model of world w. If for world $w \in UW(A,d,h)$, there exists an (l,H)-model, world w is called a ***(d,l)*-chaos**, or a chaos with depth d, and history length l. An $(1,1)$-chaos is called a *root* chaos. In the root chaos, laws don't take into consideration objects' history, and they don't "see" the objects deeper than one level. **Note**: if w is a *(d,l)*-chaos, than it is also a *(d',l')*-chaos for any $d' \geq d$, and $l' \geq l$.

3.7 Scheme properties: Passive vs. Active Schemes

A scheme defined by n-ary predicate symbol p of type $\tau 1 \times \tau 2 \times ... \times \tau k$, $k < n$, is called (purely) *passive* in l-interpretation I if:

(a) For any constants $a_1, ..., a_k$ of types $\tau 1, \tau 2, ..., \tau k$ and track $tr \in UTR(A,d,h,l\text{-}1,n\text{-}k) \cup USTR(A,d,h,l\text{-}2,n\text{-}k)$: $\pi_c(\mu(a_1), ..., \mu(a_k), tr) = true$ (i.e. condition predicate is always true).

(b) For any constants $a_1, ..., a_k$ of types $\tau 1, \tau 2, ..., \tau k$ and track $tr \in UTR(A,d,h,l,n\text{-}k) \cup USTR(A,d,h,l\text{-}1,n\text{-}k)$: $\pi_a(\mu(a_1), ..., \mu(a_k), tr) = false$ (i.e. action predicate is always false).

Otherwise p is called an *active* scheme. Passive schemes only express facts, while active schemes produce actions.

3.8 Scheme properties: Deterministic vs. Nondeterministic Schemes

N-ary predicate symbol p of type $\tau 1 \times \tau 2 \times ... \times \tau k$, $k < n$, defines a *scheme with deterministic firing* in l-interpretation I if:

For any constants $a_1, ..., a_k$ of types $\tau 1, \tau 2, ..., \tau k$ and track $tr \in UTR(A,d,h,l\text{-}1,n\text{-}k) \cup USTR(A,d,h,l\text{-}2, n\text{-}k)$, such that $\pi_c(\mu(a_1), ..., \mu(a_k), tr) = true$, there is no track $tr' \in UTR(A,d,h,l,n,k) \cup USTR(A,d,h,l\text{-}1,n\text{-}k)$, such that $tr' >> tr$, and $\pi_a(\mu(a_1), ..., \mu(a_k), tr') = true$.

Otherwise p is called a *scheme with nondeterministic firing*. The latter can cause an action without any internal reason.

N-ary predicate symbol p of type $\tau 1 \times \tau 2 \times ... \times \tau k$, $k < n$, defines a *scheme with deterministic action* in l-interpretation I, if:

For any constants $a_1, ..., a_k$ of types $\tau 1, \tau 2, ..., \tau k$ and a track $tr \in UTR(A,d,h,l\text{-}1,n\text{-}k) \cup USTR(A,d,h,l\text{-}2,n\text{-}k)$, such that $\pi_c(\mu(a_1), ..., \mu(a_k), tr) = false$, all tracks $tr' \in UTR(A,d,h,l,n\text{-}k) \cup USTR(A,d,h,l\text{-}1,n\text{-}k)$, such that $tr' >> tr$, and $\pi_a(\mu(a_1), ..., \mu(a_k), tr') = true$, are equivalent.

Otherwise p is called a scheme with a *nondeterministic action*. The latter needs external help to choose an action.

Scheme p with both deterministic firing and action is called a (fully) *deterministic* scheme, otherwise p is said to be a nondeterministic scheme. A connector with nondeterministic scheme is called an *external* connector, it needs help from the outer world in deciding when or/and how to act.

4. Procedural Semantics

Consider the task of building a formal procedure that given an initial state of world *w* (at time 0) produces an *H*-valid chaos in *l*-interpretation *I*. We call such procedure a Chaos Generator (*CG*). When defining *CG*, we'll consider only chaoses for which the cardinality of set $CON(w(t))$ (the set of all connectors taking place in world *w* at time *t*) is limited for any $t \in T$:

$$|\,CON(w(t))\,| \leq M, \quad \text{where } M \geq H/h$$

The above ensures that the principle of limited lawlessness is observed.

Our chaos generator works in steps. At each step it produces a chaos history which is an extension of the history produced at the previous step. Besides the history produced so far, the state of the chaos generator is defined by 5-tupple $GEN[i]$:

$$GEN[i] = \langle SCON[i], ECON[i], ACON[i], etime[i], atime[i] \rangle,$$

where:

$SCON[i]$	- a set of satisfied connectors,
$ECON[i]$	- a set of excited connectors,
$ACON[i]$	- a set of active connectors,
$etime[k]: ECON[i] \cup ACON[i] \text{ -> } T$	- time when became excited,
$atime[k]: ACON[i] \text{ -> } \mathrm{T}$	- time when became active.

Sets $SCON[i]$, $ECON[i]$, and $ACON[i]$ do not intersect, and $SCON[i] \cup ECON[i] \cup ACON[i] = CON(his[i](i))$, where $his[i]$ is the chaos history produced up to the step *i*.

SCON defines a set of connectors that are satisfied and don't need to do anything with their operands. *ECON* defines the set of connectors:

- whose schemes are of active type and whose operands (more precisely their *d*-depth bodies) have been changed on one of the previous steps, or
- whose schemes happened to be of nondeterministic firing type, and they manifested the desire to change their operands without any internal reason.

ACON defines a set of connectors that have already got access to their operands and started their law enforcement work.

The idea of *CG* can be described as connectors circulation in nature:

- At step *i*, *CG* chooses a connector from *ACON* that has finished its work, and it introduces changes made by this connector in its operands into the new state of the world ("*Commit*"). In order to choose such connector, *ACON* should contain at least one *ready* connector. The readiness is defined by expression $(i - atime[i](c)) \geq h$. The chosen connector becomes the actor of a new event. If there is no ready connectors in *ACON*, the world doesn't change at step *i*.

- The actor becomes satisfied and is moved to *ECON*.
- Some connectors from *SCON* may become exited, e.g., because their operands have been changed by the actor, consequently they are moved to *ECON*.
- Some connectors from *ACON* can have their operands changed by the actor. They are moved back to *ECON* ("*Rollback*").
- Some connectors from *ECON* have got access to their operands and start working. They are moved to *ACON*.

By choosing different algorithms of transferring connectors from one set to another, we can model different ways of solving possible conflicts between connectors that hang on common objects. Three basic types of *CG* functioning are possible:

- Sequential execution, where *ACON* always consists of no more than one connector.
- Pessimistic strategy, where *ACON* never contains connectors that are linked to each other via common objects. In the worst case, when all connectors are linked to each other, the pessimistic strategy is reduced to the sequential execution.
- Optimistic strategy, where as many excited connectors as possible are moved to *ACON* (even the ones that are linked to each other). In the worst case, when too many rollbacks have happened to some connectors and they become overdue, the optimistic strategy is reduced to the pessimistic one. (Connector c is overdue at time i if $(i - etime[i](c)) \geq H - M*h$, where M is the maximum number of connectors taking place at time i.)

Initial state of *CG* is defined as:

$GEN[0] = <SCON[0] = PSCON[w(0)], ECON[0] = ASCON[w(0)] - \{c\}, ACON[0] = \{c\}, etime[0], atime[0]>$,

where:

$w[0]$ - the given initial state of the world;
$PSCON[w(0)]$ - the set of connectors with passive schemes (in the initial state of the world);
$ASCON[w(0)]$ - the set of connectors with active schemes (in the initial state of the world);
$c \in ASCON[w(0)]$ - arbitrary chosen connector with active scheme (from the initial state);
$etime[0](c') = 0$ for any $c' \in ASCON[w(0)]$;
$atime[0](c) = 0$.

The Chaos Generator described above can be considered as a model of a connectors scheduler. This model gives a natural explanation to such values as h, H, and l:

- h - is the time which a connector needs to introduce changes in its operands.
- H-h - is the maximum time an excited connector can wait until it gets access to its operands to restore the law. The relation $h*M <= H$ ensures the principle of limited lawlessness in the worst case: there exist M connectors, and all of them are linked to each other via common objects.

- *l* - the number of events that should be stored for each object in order to continue chaos generation (full history is not needed).

Three different algorithms of activating connectors that were mentioned above can be used as a basis for implementing the connector scheduler in three different environments:

- uniprocess environment,
- environment with a limited number of available processes - one per each active connector plus one for the scheduler,
- environment with unlimited number of available processes - one per each connector and one per each object.

5. CHAOS Programming Environment

5.1 Basic Elements

The model described in the previous sections consists of:

- objects,
- connectors, and
- constants.

Objects and connectors are the main elements of the model, they serve to represent complex structures. Constants are foreigners in this world, they are used to incorporate external data types. Constants may be of simple types, like integer, floats, or strings, or of some complex types, but in any case they are treated in the model as unstructured entities (so called *blobs*).

Connectors express objects' properties and relations. An *n*-ary connector with only one operand expresses properties of this operand with the help of its *n-1* constants (if any). If the connector's scheme is passive, it just assigns properties to the object. If the scheme is active, the connector forces the object's structure and behavior to comply with those properties.

An *n*-ary connector with *k* operands expresses a relation between his *k* operands, where remaining *n-k* constants (if any) describe properties of this relation. If the connector's scheme is passive, it just expresses the fact that the relation between the objects exists, and possibly assigns some properties to the relation. If the scheme is active, the connector forces the objects' structure and behavior to be in sync with each other and, possibly, with the properties of the relation.

5.2 Classes

The notion of class is absent from our model. Objects are primary elements of it. A class may be represented as a one operand connector that forces the object on which it hangs to have a predefined structure. Moreover, a class may be represented as a two-operands connector where one operand serves as a template of the structure that should be imposed on the other. A chain of class connectors, one hanging on the template of the next one in the chain, may represent the notion of class hierarchy.

With our concept of class, an object can change its class during its lifetime. This happens when one class connector is substituted with another one. After that, it's up to the second connector to rearrange the structure of the object so that it satisfies the requirements of the second class.

As several connectors can hang on the same object, several classes can be assigned to it which may serve as a model of multiple inheritance. It's up to class connectors to solve the inheritance conflicts in one way or another. If the connectors are defined in an incompatible way, hanging them on the same object can lead to unstable behavior where one connector will always undo the changes done by the others.

5.3 Communication with External World

External connectors, the ones with nondeterministic schemes, are the primary mechanism to model communication with the external world, e.g., people, or electronic devices. An external connector may be thought of as having a display where a human being can help the connector to do its job.

If the connector's scheme is with deterministic firing but with nondeterministic action, it will be up to the connector to request the human assistance by beeping, blinking, etc. After providing assistance, the human being may return to his other occupations, drinking coffee, for example, until the connector calls him again. Action nondeterminsim may vary from allowing any possible structural changes in the connector's operands to selection from a list of predefined choices.

If the connector has a scheme with nondeterministic firing but with deterministic action, it can be represented as a big button which the user presses from time to time based on observations that lie beyond the boundaries of our system. Then it's up to the connector to do the rest of the job.

All other situations of nondeterminism may be considered as combination of the above two cases.

5.4 Programming Language

The chaos runtime environment consists of a connector scheduler, and a connector executor. The connector scheduler decides when and which connectors should get access to their operands and start their law-enforcement work. The generation procedure described in section 4 outlines the main principles of the scheduler. The executor is a device that given the text of the law, changes the connector's operands according to it. To facilitate the work of the executor we need a programming language for writing laws.

A law language should have operation of two types:

- on constants (external data types), like arithmetic operations, string operations, etc.
- on object-connectors structures (internal data types), like *add connector*, *delete connector*, etc.

Different approaches to constructing a law language can be used. For example, it may be built as a relatively law-level language with basic operations, like *add/delete connector*. However, for dealing with complex structures, a pattern matching and transformation technique would be more appropriate. The sophisticated pattern matching technique has been developed in AI for LISP, PROLOG, and other AI languages. This technique could be used for creating a high-level chaos programming language. Building such language is beyond the scope of this work. To demonstrate the principles of chaos programming, below we present two simple examples written in would-be low level chaos programming language. It follows C-syntax, and has loop operations, set operations, and simple pattern matching operations.

Two examples presented in the next subsection are classical: a shallow mirror law (correspond to the normal *shallow_copy* method/function), and a deep mirror law (correspond to the normal *deep_copy* method/function). They belong to the root chaos as they don't take into consideration any historic data, and don't see the objects deeper than one level. Nevertheless, those examples demonstrate the basic principles of chaos programming.

5.5 Examples

Example 1. The *shallow_mirror(o1,o2)* scheme enforces object *o2* to contain the same connectors as object *o1* with one exception. If connector *c* from *o1* has *o1* as one of its operands, connector *c1* is included in *o2* instead of *c*, where *c1* is derived from *c* by substituting operand *o1* with *o2*.

```
deterministic  scheme shallow_mirror( read-only object o1, object o2 )
{
/* Delete from o2 all connectors that do not belong to o1. Loop over all connectors in o2 */

        for each c in o2
        {
                c1 = subst(c, o2/o1)
/*  Operation subst(c, x/y) converts connector c in connector c1 by substituting any occurrence of
term x ( object or a constant) by term y */
                if c1 not in o1
                {
                        delete(o2, c);
/* Operation delete(o,c) deletes connector c from the body of object o */
                }
        }
/* Add to o2 all connectors from o1 that are not in o2. Loop over all connectors in o1 */
        for each c in o1
        {
                c1 = subst(c, o1/o2)
                if c1 not in o2
                {
                        add(o2, c1);
/* Operation add(o,c) adds connector c to the body of object o */
                }
        }
}
```

Example 2. The *deep_mirror(o1,o2, buffer)* scheme enforces object *o2* to contain images of objects and connectors from the body of object *o1.* The third operand is used as a buffer for recursive decomposition defined as follows:

(a) For each object *o´* found in the body of *o1* (except *o1* itself), there should be one and only one connector *deep_mirror(o´, o´´, buffer´)* in *buffer.* This connector links *o´* to its image *o´´ (o´* $\neq$ *o´´).*
(b) *Buffer* should not contain any other connectors than described in (a).
(c) For any connector *c* from *o1, o2* should contain its image *c´*. Connector *c´* has the same scheme as *c*, but it hangs on the images of the original connector's operands. If one of the operands is *o1* itself, then *o2* is used as an operand for *c´.*
(d) Object *o2* should not contain any other connectors than described in (c).

Note: The code below won't handle properly the situation when an object serves as its own subobject on the level more the one.

deterministic scheme **deep_mirror(** *read-only object* **o1,** *object* **o2,** *object* **buffer)**
{
/* Delete from **buffer** all connectors which are not **deep_mirror**, or whose first operand is not included in the body of **o1**, or which are duplicates ***/**

```
    for each c in buffer
    {
        if scheme(c) ≠ deep_mirror
        {
            delete(buffer, c)
        }
        else if  term(c,1) not in guts(o1, 1)
        {
            delete(buffer, c)
        }
        else
        {
            let connector(c1) where  scheme(c1) = deep_mirror && c1≠ c &&
               term(c1,1) =  term (c,1)
            if c1 in buffer
            {
                delete(buffer, c)
            }
        }
    }
```

/* Ensure that for each object in **o1** except **o1** itself, there is a connector **deep_mirror** in **buffer** */

```
    for each x in guts(o1,1) where x ≠ o1
    {
        let object( y)
        let object( z)
```

```
                    if deep_mirror(x,y,z) not in buffer
                    {
                              add(buffer, deep_mirror(x, new, new))
/* Operation new creates a new object with empty body */
                    }
          }
/* Delete from o2 all non-image connectors */

          for each c in o2
          {
                    let connector(c1) where  scheme(c) = scheme(c1) &&
                              for each t in terms(c)
                              {
                                        let t1 where t1 in terms(c1) && pos(c1, t1) = pos(c, t)
                                        if  constant(t)
                                        {
                                                  t = t1
                                        }
                                        else if  t = o2
                                        {
                                                  t1 = o1
                                        }
                                        else
                                        {
                                                  let object(z)
                                                  deep_mirror(t1, t, z) in buffer
                                        }
                              }
                    if c1 not in o1
                    {
                              delete(o2, c)
                    }
          }
/* Ensure that for each connector in o1 there is its image in o2 */
          for each connector c in o1
          {
                    let connector(c1)  where scheme(c)= scheme(c1) &&
                              for each t in terms(c)
                              {
                                        let t1 where t1 in terms(c1) && pos(c1, t1) = pos(c, t)
                                        if  constant(t)
                                        {
                                                  t = t1
                                        }
                                        else if t = o1
                                        {
                                                  t1 = o2
                                        }
```

```
                    else
                    {
                            let object(z)
                            deep_mirror(t, t1, z) in buffer
                    }
            }
        if c1 not in o2
        {
            add(o2, c1)
        }
    }
}
```

6. Concluding Remarks

The object-oriented model presented in the previous sections is the main result of our own CHAOS project started back in 1984, where CHAOS stands for Concurrent Human Assisted Object Systems. The ultimate goal of the project was to create methods and programming environment for design and implementation of distributed management automation systems. The main idea was to develop human-assisted systems as opposite to human-assisting systems. The former works on its own appealing to the human help only when necessary. The latter represent the traditional view of business system development, it's a human being who does the work, the system only helps him to perform certain functions.

Our theoretical model was first tested in practice in 1989-90 when we were developing an application for supporting sales and marketing activities of a trading company. The system was called *DealDriver* to highlight that it helps the workers to "drive" their deals from the beginning to the end which is receiving payment. Our objective was to develop a system able to assist the office workers in all aspects of their everyday work, including registering all the activities performed, and helping to plan the new ones. For this project, we devised a practical model for representing business processes that may be regarded as a *specialization* of our formal model. During the project we also created home-made development tools that included Hi-base - an object-oriented historical database, and Navi - an object oriented user-interface system. Hi-base is able to store all the previous states of the objects and show them on demand. Navi allows the end-user to easily navigate along the links connecting various objects to each-other in presence and past (history).

The tools developed during the project were later successfully used for building other applications. The results of our practical work are summarized in brochures [6,17].

References

1. Rambaugh J., et al. Object-Oriented Modeling and Design, Prentice Hall, Inc., Englewood Cliffs, NJ, 1991.
2. Coad P. and Yourdon E. Object-Oriented Analysis. Yourdon Press, Englewood Cliffs, NJ, 1990.
3. Graham I. Migrating to Object Technology, Addison-Wesley, Wokingham, England, 1995.
4. Bider I. and Khomyakov M. Object-Oriented Model for Representing Software Production Processes. In Jan Bosch and Stuart Mitchell, editors, ECOOP'97 Workshop Reader, Springer 1997, LNCS 1357 (under print)
5. Bider I. and Khomyakov M. One Practical Object-Oriented Model of Business Processes. In Klimov H., Rumpe B., Simmonds I., eds., OOPSLA’97 Workshop on Object-oriented Behavioral Semantics. Institute Für Informatic. Technische Universität München, 1997, TUM-19737, pp. 25- 31.
6. Bider I. Developing Tool Support for Process Oriented Management. In Data Base Management 26-01-30, Auerbach, RIA Group, 1997.
7. Lloyd J.W. Foundation of Logic Programming. Springer-Verlag, 1993
8. Kifer M., Lausen G., Wu J. Logical Foundation of Object-Oriented and Frame-Based Languages. Journal of the ACM 42(4):743-843, July 1995.
9. Falcone Sampaio P.R. and Paton N.W. Deductive Object-Oriented Database Systems: A survey. In: Greppet A. Berndtsson M. Eds. Rules in Database Systems. RIDS’97 Proceedings. Springer 97, LNCS 1312.
10. Hintikka, J. Knowledge and Belief. Cornell University Press: Ithaca, NY, 1962
11. Kripke, S. Semantical analysis of modal logic. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik, 9:67-96, 1963.
12. Thom R. Stabilité structurelle et morphogenèse. Intereditions, Paris, 1977.
13. Montague R. Universal Grammar. In R. Thomason, ed. Formal Philosophy: Selected papers of Richard Montague, Yale University Press, 1974.
14. Turchin V.F. The concept of a supercompiler, ACM Transactions on Programming Languages and Systems, 8 (3): pp.292-325, 1986.
15. Turchin V.F. Refal-5: Programming Guide and Reference Manual, New England Publishing Co. Holyoke, MA, 1989
16. Borshchev V. and Khomiakov M. Club systems. In: Automatic Documentation and Linguistics. Vol. 10, N 3. Allerton Press, 1976
17. Bider I. Object Driver: A Method for Analysis, Design, and Implementation of Interactive Applications. In Data Base Management. 32-10-25, Auerbach, RIA Group, 1997.