UDC 519.681.3

ONTOLOGICAL MODELS IN OTSL

I.S. Anureev

Siberian Division of the Russian Academy of Sciences
A.P. Ershov Institute of Informatics Systems,
630090 Novosibirsk, Russia, ac. Lavrentiev ave., 6,
Fax: +7 383 332 3494; phone: +7 383 330 6360.
E-mail: anureev@iis.nsk.su

OTSL[1] is a language of description of ontological transition systems [1]. Ontological transition systems are formalism for description of semantics of computer systems that combines transition systems with ontological models. In this paper, constructs of OTSL specifying the ontological models of ontological transition systems is presented. Formal semantics of these constructs is defined.

OTSL – язык описания онтологических систем переходов [1]. Онтологические системы переходов – формализм для описания семантики компьютерных систем, который комбинирует системы переходов с онтологическими моделями. В данной работе представлены конструкции OTSL, специфицирующие онтологические модели в онтологических системах переходов. Определена формальная семантика данных конструкций.

Introduction

The state transition systems are a well-known formalism for description of operational semantics of programming languages and program models. A common way to rigorously define the operational semantics, pioneered by Gordon Plotkin in his paper "A Structural Approach to Operational Semantics" [2], is to provide a state transition system for the language of interest.

A state transition system is defined as an abstract machine which consists of a set of states and transitions between states. On the one hand, simplicity of definition of these systems makes them a universal formalism for description of the behavior of systems of different nature (algorithms, programs, program models, computer systems, and so on). On the other hand, it leads to a loss of the conceptual structure of systems in their descriptions.

A natural question is how to enrich the states or/and transitions of transition systems to make these systems more 'conceptually capacious', having preserved their generality.

A logical-algebraic approach to solution of this problem was suggested by Yuri Gurevich, based around the concept of an abstract state machine [3, 4]. ASMs[2], formerly known as evolving algebras, are a special kind of transition systems. The states of ASMs can be arbitrary algebras. The choice of an appropriate algebra signature allows us to adapt ASMs to problem domains. The ASM approach has already proven to be suitable for large-scale specifications of realistic programming languages [5–10]. Other applications of ASMs to various domains can be found in [11].

The ASM theory is the basis for Abstract State Machine Language [12] developed by Microsoft and XASM (Anlauff's eXtensible ASMs) [13], an open source implementation.

In this paper, the ontological approach[3] to solution of this problem is suggested, based around the concept of an ontological transition system. OTSs[4] are a special kind of transition systems. An OTS can be regarded as a transition system which has the following properties:

- there is a conceptual structure (a sets of concepts and a set of relations) which is common for all states of the transition system;
- there is a function of retrieving the content of this conceptual structure from the states of the transition system.

Formally, an ontological transition system consists of a transition system and ontological model. In turn, an ontological model consists of a set of objects, ontology, and meaning function which defines the values of concepts and relations for each state of the transition system.

On the basis of OTSs, the ontological transition system language OTSL has been developed. It includes two sublanguages: a language of actions and a language of formulas. Actions specify the transitions of OTSs. Formulas specify the ontological entities of OTSs. In this paper, the language of formulas is presented. A description of the language of actions can be found in [14].

The paper has the following structure. Section 2 defines the ontological transition systems and related entities.

[1] OTSL is an abbreviation for Ontological Transition System Language.
[2] ASMs are an abbreviation for Abstract State Machines.
[3] This approach is called the method of operational-ontological semantics.
[4] OTSs are an abbreviation for Ontological Transition Systems.

Section 3 sketches out the basic notions of the OTSL language. The base constructs of the language of formulas such as terms, concept expressions and formulas are presented in Sections 4, 5, and 6, respectively. Section 7 presents additional constructs which can be used in formulas. On the one hand, these constructs are reducible to the base formula constructs. On the other hand, they enlarge a conceptual expressiveness of the language of formulas. Sections 8 and 9 define concept declarations and relation declarations, respectively. They are used to specify concepts, relations, and the values of concepts and relations.

This research is partially supported by the grants 06-01-00464a and 08-01-00899-a from RFBR, and by the integration grant 14 from SB RAS.

1. Ontological Transition Systems

This section defines ontological transition systems and related entities.

An unlabelled transition system tS is defined as a pair (st, tr). The set st[5] is called a set of states. The function tr $\in$[6] st$\times$[7]st $\rightarrow$[8]bool[9] is called a transition relation. The property tr(St, St') means that there is a transition from the state St to the state St'.

A labelled transition system tS is defined as a triple (l, st, tr). The set l is called a set of labels. The set st is called a set of states. The function tr $\in$ st$\times$st$\rightarrow$(l$\rightarrow$bool) is called a transition relation. The property tr(St, St)(L) means that there is a transition from the state St to the state St' with the label L.

An ontology of a system describes its conceptual structure. It consists of a set of concepts and a set of relations. Concepts define the kinds of sequences of objects of the system. In particular, they define the kinds of objects of the system. Relations define the kinds of interrelations between objects.

Formally, an ontology ont is defined as a pair (co, re). The set co is called a set of concepts of the ontology ont. The set re is called a set of relations of the ontology ont.

An ontological transition system consists of a set of objects, transition system, ontology and meaning function which define the values of concepts and relations for each state of the transition system. The value of a concept is defined as a subset of the set of sequences of objects. The value of a relation is defined as a binary relation on sequences of objects.

Formally, an ontological transition system ots is defined as a quadruple (ob, tr, ont, val). The set ob is called a set of objects. Let se be a set of sequences of objects. The set st of states of the OTS is defined as follows:

$$st = ob \rightarrow se.$$

The function val $\in$ (co$\times$st$\rightarrow$2se[10])$^{\oplus}$[11](re$\times$st$\rightarrow$2se$\times$se) is called a meaning function. The set val(Co, St) is called a value of the concept Co in the state St. A sequence Se $\in$ val(Co, St) is called an instance of the concept Co in the state St. The set val(Re, St) is called a value of the relation Re in the state St. A pair (Se, Se') $\in$ val(Re, St) is called an instance of the relation Re in the state St. The sequence St(Ob) is called a value of the object Ob in the state St. The value of an object defines its structure and objects which interrelate with it. The values of objects are used to retrieve information about the values of concepts and relations.

A triple (ob, ont, val) is called an ontological model of the OTS ots.

2. Ontological Transition System Language

This section sketches out the basic notions of OTSL.

Terminals in grammar of OTSL are divided into keywords and objects. Keywords identify constructs of OTSL. Objects of OTSL represent objects of OTSs.

The set kW of keywords is built in the following way:

kW ::=[12] ? |[13] ! | #… | = | ~ | : | ; | @ | [|] | { | } | (|) | ()

---

[5] In this paper, an entity is often defined by the name of a set which contains all instances of the entity. This name is also used as a nonterminal in grammar rules. Under the agreement, this name starts with a small letter. The name of an element of the set coincides with the name of the set (possibly with additional indexes and strokes) except for the first letter which is capitalized. For example, here `st` is a set of states. It defines the entity 'state'. Then `St`, `St'` and $\mathtt{St}_2$ are states.

[6] $\in$ means 'belongs to'.

[7] $\times$ means 'Cartesian product'.

[8] `X`$\rightarrow$`Y` means the set of all total functions from `X` to `Y`.

[9] `bool` means the set `{true, false}`.

[10] `2X` means the set of all subsets of the set `X`.

[11] X$\rightarrow$Y$^{\oplus}$X'$\rightarrow$Y', where X$\cap$X'=$\varnothing$, means the set of all total functions which act from X to Y and from X' to Y'. Symbols $\varnothing$ and $\cap$ means 'empty set' and 'intersection', respectively.

[12] `::=` means 'has the form' in grammar rules.

| and | or | not | implies | iff | := | +=,

where #… is any sequence of letters and digits from the set {a,…,z, A,…,Z, 0,…,9} starting with # except for #i and #o.

The set ob of objects is an arbitrary set such that ob ∩ kW = ∅.

The set sOb ⊆[14] ob of special objects is built in the following way:

sOb ::= true | new | val | #i | #o | o | s | e | ns | eo.

Special objects represent the specific-purpose objects which are common for all OTSs.

Basic nonterminals in grammar of OTSL are divided into sequences, formulas, actions, and OTS declarations. Sequences are the main datatype of OTSL. Formulas specify the values of concepts and relations in OTSs. Concepts and relations of OTSs themselves are represented by objects in OTSL: co⊆ob ∧[15] re⊆ob. Actions specify transitions of OTSs. OTS declarations specify OTSs.

The sets se and nSe of sequences and nonempty sequences, respectively, are built in the following way:

se ::= () | nSe,

nse ::= ob | ob nSe.

Let us note that, by definition, ob⊆se ∧ ob⊆nSe. The sequence () is called the empty sequence.

The concatenation function con ∈ seЧse→se is defined on sequences:

con(NSe, NSe') = NSe NSe' ∧ con((), Se) = con(Se, ()) = Se.

The equality relation = ∈ seЧse→bool, and weak equality relation ~ ∈ seЧse→bool are defined on sequences:

- Se = Se' = true, if Se and Se' are the same sequence, and Se = Se' = false, otherwise.
- The weak equality relation on sequences is defined as follows: Se ~ Se' = true, if the sequence Se is a permutation of the sequence Se', and Se ~ Se' = false, otherwise.

Formulas and actions are built from terms and concept expressions. Semantics of terms, concept expressions, formulas and actions are defined by the meaning function val that has the following form for these constructs:

val ∈ st → (te[16] → se) for terms,

val ∈ st → (coExp[17] → 2se)for concept expressions,

val ∈ st → (fo[18] → bool) for formulas,

val ∈ st Ч st → (act[19] → bool) for actions.

The semantics is defined in the context of an OTS declaration, i.e. the OTS declaration is an implicit argument of the meaning function val for these constructs.

Terms are used to construct sequences of objects. The sequence val(St)(Te) is called the value of the term Te in the state St.

Similar to concepts, concept expressions define the kinds of sequences of objects. They can be considered as anonymous concepts. The set val(St)(CoExp) of sequences of objects is called the value of the concept expression CoExp in the state St. A sequence Se ∈ val(St)(CoExp) is called an instance of the concept expression CoExp in the state St.

The object val(St)(Fo) is called the value of the formula Fo in the state St. A formula Fo is true in a state St, if val(St)(Fo) ≠ (). Otherwise, the formula Fo is false in the state St.

Actions change states of OTSs. They are labels for transition relations. The meaning function val for actions coincides with the transition relation tr.

Terms, concept expressions, and formulas are defined in Sections 4, 5 and 6, respectively. Actions are defined in [14].

There are five basic concepts in OTSL: o, s, e, ns, and eo. Their values depend on no state:

∀[20]St(val(o, St) = ob ∧ val(s, St) = se ∧ val(e, St) = {()} ∧

---

[13] | separates alternatives and means 'or' in grammar rules.

[14] ⊆ means 'is a subset of'.

[15] ∧ means 'and'.

[16] `te` means the set of terms.

[17] `coExp` means the set of concept expressions.

[18] `fo` means the set of formulas.

[19] `act` means the set of actions.

[20] ∀ means 'for all'.

val(ns, St) = nSe ∧ val(eo, St) = ob ∪[21] {()}).

An OTS declaration is a sequence of OTS declaration members. OTS declaration members are divided into concept declarations, relation declarations, and transition declarations:

otsDec[22] ::= otsDecMem[23] | otsDecMem otsDec,

otsDecMem ::= coDec[24] | reDec[25] | trDec[26],

coDec ::= #c ob { fo },

reDec ::= #r ob { fo },

trDec ::= #t { act }.

The concept declaration #c Ob{Fo} defines the concept Ob with specification Fo. The special object #i[27] is used in the specification Fo of the concept Ob to refer to instances of the concept Ob. The value of the concept Ob in a state St is defined as the set of sequences Se such that the formula Fo(#i←Se)[28] is true in the state St.

The relation declaration #r Ob{Fo} defines the relation Ob with specification Fo. The special objects #i[29] and #o[30] are used in the specification Fo of the relation Ob to refer to instances of the relation Ob. The value of the relation Ob in a state St is defined as the set of pairs (Se, Se') of sequences such that the formula Fo(#i←Se, #o←Se') is true in the state St.

The transition declaration #t {Act} defines a set of transitions labeled by objects. The special object #i is used in the action Act to refer to the object which labels (or initiates) the transition. A transition from a state St to a state St' with a label Ob belongs to this set, if val(St, St')(Act(#i ← Ob)) = true.

3. Terms

Terms are are built in the following way:

te ::= eSe | ob | obV | teCom.

The object value obV and term composition teCom are defined below.
The value of the empty sequence is the empty sequence itself:

val(St)(()) = ().

The value of an object is the object itself:

val(St)(Ob) = Ob.

The object value is defined as follows:

obV ::= ? ob.

The value of the object value ?Ob in a state St is the value of the object Ob in the state St:

val(St)(?Ob) = St(Ob).

The term composition teCom is defined as follows:

teCom ::= te te.

The value of a composition of terms is concatenation of the values of the terms:

val(St)(Te Te') = con(val(St)(Te), val(St)(Te')).

---

[21] ∪ means 'union'.
[22] `otsDec` means the set of OTS declarations.
[23] `otsDecMem` means the set of OTS declaration members.
[24] `coDec` means the set of concept declarations.
[25] `reDec` means the set of relation declarations.
[26] `trDec` means the set of transition declarations.
[27] `i` means 'instance'.
[28] The entities defined by grammar rules (terms, formulas and so on) are represented as labeled trees that correspond to their grammar structure. Therefore in the sequel, the 'tree' terminology (positions, substitutions and so on) is used for these entities. Here $T(L_1 \leftarrow T_1, \ldots, L_n \leftarrow T_n)$ means a tree which is obtained from the tree T by replacement of all occurences of leaves with the labels $L_1, \ldots, L_n$ by the trees $T_1, \ldots, T_n$, respectively.
[29] `i` means 'input'.
[30] `o` means 'output'.

4. Concept expressions

The set coExp of concept expressions is built in the following way:

coExp ::= co | im | preIm,

where the image im and preimage preIm are defined below.

Let OtsDec be an OTS declaration. Let us define semantics of the above-mentioned kinds of concept expressions in the context of OtsDec.

The value val(St)(Co) of the concept Co in the state St is defined as the value of the concept Co:

val(St)(Co) = val[31](Co, St).

The set im of images is defined as follows:

im ::= re < te.

The value of an image Re<Te in a state St is the image of the value of the relation Re for the set {val(St)(Te)}:

val(St)(Re<Te) = {Se|(val(St)(Te), Se) ∈ val[32](Re, St)}.

The set preIm of preimages is defined as follows:

preIm ::= re > te.

The value of a preimage Re>Te in a state St is the preimage of the value of the relation Re for the set {val(St)(Te)}:

val(St)(Re>Te) = {Se | (Se, val(St)(Te)) ∈ val(Re, St)}.

A relation Re is called an attribute of a sequence Se in a state St, if there is at most one sequence St' such that (Se, Se') ∈ val(Re, St). A relation Re is called an attribute of a sequence Se, if Re is an attribute of the sequence Se in each state St. A relation Re is called a mandatory attribute of a sequence Se in a state St, if there is just one sequence St' such that (Se, Se') ∈ val(Re, St). A relation Re is called a mandatory attribute of a sequence Se, if Re is a mandatory attribute of the sequence Se in each state St. An attribute Re of a sequence Se is said to have the value Se' in a state St, if the set val(St)(Re<Se) = {Se'}. The value of an attribute Re of a sequence Se is said to be indeterminate in a state St, if the set val(St)(Re<Se) = ∅.

A relation Re is called an attribute of a concept Co in a state St, if the relation Re is an attribute of each instance of the concept Co in the state St. A relation Re is called an attribute of a concept Co, if the relation Re is an attribute of the concept Co in each state St. A relation Re is called a mandatory attribute of a concept Co in a state St, if the relation Re is a mandatory attribute of each instance of the concept Co in the state St. A relation Re is called a mandatory attribute of a concept Co, if the relation Re is a mandatory attribute of the concept Co in each state St.

5. Formulas

This section defines formulas and the related entities.

The set fo of formulas is built in the following way:

fo ::= aFo | pFo | qFo | dFo | bFo.

The sets aFo, pFo, qFo, dFo, and bFo of atomic formulas, propositional formulas, quantified formulas, dynamic formulas, and bracketed formulas, respectively, are defined below.

The set aFo of atomic formulas is built in the following way:

aFo ::= te | mem | eq | wEq.

The membership mem, equality eq and weak equality wEq are defined below.

The set mem of memberships is built in the following way:

mem ::= te:coExp.

A membership Te:CoExp is true in a state St, if the value of the term Te is an instance of the concept expression CoExp:

val(St)(Te:CoExp) = true, if val(St)(Te) ∈ val(St)(CoExp),

val(St)(Te:CoExp) = (), otherwise.

The set eq of equalities is built in the following way:

eq ::= te = te.

An equality Te=Te' is true in a state St, if the values of terms Te and Te' in the state St are equal:

[31] As shown in Section 8, the value of `val(Co, St)` depends on `OtsDec`.

[32] As shown in Section 9, the value of `val(Re, St)` depends on `OtsDec`.

val(St)(Te = Te') = true, if val(St)(Te) = val(St)(Te'),

val(St)(Te = Te') = (), otherwise.

The set wEq of weak equalities is built in the following way:

eq ::= te ~ te.

An equality Te ~ Te' is true in a state St, if the values of terms Te and Te' in the state St are weakly equal:

val(St)(Te ~ Te') = true, if val(St)(Te) ~ val(St)(Te'),

val(St)(Te ~ Te') = (), otherwise.

The set pFo of propositional formulas is built with the help of logical connectives: negation not, conjunction and, disjunction or, implication implies, and equivalence iff:

pFo ::= not fo | fo and fo | fo or fo | fo implies fo | fo iff fo.

with their usual semantics:

val(St)(not Fo) = true, if val(St)(Fo) = (),

val(St)(not Fo) = (), otherwise.

val(St)(Fo and Fo') = true, if val(St)(Fo) ≠ () ∧ val(St)(Fo') ≠ (),

val(St)(Fo and Fo') = (), otherwise.

val(St)(Fo or Fo') = true, if val(St)(Fo) ≠ () ∨ val(St)(Fo') ≠ (),

val(St)(Fo or Fo') = (), otherwise.

val(St)(Fo implies Fo') = true, if val(St)(Fo) ≠ () ⇒[33] val(St)(Fo') ≠ (),

val(St)(Fo implies Fo') = (), otherwise.

val(St)(Fo iff Fo') = true, if val(St)(Fo) ≠ () ⇔[34] val(St)(Fo') ≠ (),

val(St)(Fo iff Fo') = (), otherwise.

The set qFo of quantified formulas is built with the help of existential (?) and universal (!) quantifiers

qFo ::= (? (bin) fo) | (! (bin) fo)

bin ::= ob : coExp | bin bin

with their usual semantics:

val(St)((?(Ob1:CoExp1 … Obn:CoExpn)Fo)) = true, if

∃[35](A1∈val(St)(CoExp1), …, An∈val(St)(CoExpn))(val(St)(Fo(Ob1←A1, …, Obn←An)) ≠ ()),

val(St)((?(Ob1:CoExp1 … Obn:CoExpn)Fo)) = (), otherwise.

val(St)((!(Ob1:CoExp1 … Obn:CoExpn)Fo)) = true, if

∀[36](A1∈val(St)(CoExp1), …, An∈val(St)(CoExpn))(val(St)(Fo(Ob1←A1, …, Obn←An)) ≠ ()),

val(St)((!(Ob1:CoExp1 … Obn:CoExpn)Fo)) = (), otherwise.

The elements of the set bin are called bindings.

Dynamic formulas are built with the help of dynamic logic modalities

dFo ::= (? { act } fo) | (! { act } fo)

with the usual semantics:

val(St)((?{Act}Fo)) = true, if ∃St'(tr(St,St')(Act)∧val(St')(Fo)≠()),

val(St)((?{Act}Fo)) = (), otherwise.

val(St)((!{Act}Fo)) = true, if ∀St'(tr(St,St')(Act)⇒val(St')(Fo)≠()),

val(St)((!{Act}Fo)) = (), otherwise.

The set bFo of bracketed formulas is built in the following way:

bFo ::= (fo).

Brackets are used to define the order of computation of subformulas in formulas:

---

[33] ⇒ means 'implication'.

[34] ⇔ means 'equivalence'.

[35] ∃[X1, …, Xn](X) means ∃X1 … ∃Xn(X), where ∃X(Y) means 'there is X such that Y'.

[36] ∀[X1, …, Xn](X) means ∀X1 … ∀Xn(X).

val(St)((Fo)) = val(St)(Fo).

The order of computation is also specified by priority and associativity of operations. Operations are listed below in the descending order:

= ~ not and or implies iff.

Example. The formula A or B and C = D is equivalent to the formula A or (B and (C = D)).
In addition, the operations and and or are left associative.
Example. The formula A and B and C is equivalent to the formula (A and B) and C.

6. Additional formula constructs

This section presents additional constructs which can be used in formulas. On the one hand, these constructs are reducible to the basic formula constructs. On the other hand, they enlarge the conceptual expressiveness of the OTSL language. These constructs include anonymous objects and anonymous sequences. They are used in place of terms. To introduce them, the set te of terms is redefined.

The set te of terms is built in the following way:

te ::= eSe | ob | obC | teCom | anOb | anSe,

The sets anOb and anSe of anonymous objects and anonymous sequences, respectively, are defined below.
The set anOb of anonymous objects is built in the following way:

anOb ::= (= te) | (te) | (~ te).

An anonymous object (= Te) represents any object Ob with the value equal to the value of the term Te in the state St, i.e. St(Ob) = val(St)(Te). An anonymous object (Te) is a synonym for (= Te).

An anonymous object (~ Te) represents any object Ob with the value weakly equal to the value of the term Te in the state St, i.e. St(Ob) ~ val(St)(Te).

An anonymous object (⊙[37] Te) is implicitly bound by the existential quantifier ?, i.e. any formula Fo such that Foq[38]=(⊙ Te) is equivalent to the formula (?(Ob:o)(Fo[Ob]q[39] and ?Ob ⊙ Te)). This requirement guarantees existence of at least one object which satisfies the formula Fo and has the value defined by the term Te.

The set anSe of anonymous sequences is built in the following way:

anSe ::= * : coExp.

An anonymous sequence *:CoExp represents any sequence Se such that Se ∈ val(St)(CoExp). An anonymous sequence Se is implicitly bound by the existential quantifier ?, i.e. any formula Fo such that Foq = *:CoExp is equivalent to the formula (?(Ob:CoExp)Fo[Ob]q). This requirement guarantees existence of at least one sequence Se∈val(St)(CoExp) which satisfies the formula Fo.

Anonymous objects and anonymous sequences can be reducible to other constructs. Their semantics is defined by the reduction function red ∈ aFo→qFo. This function normalizes atomic formulas, eliminating anonymous objects and anonymous sequences:

red(Fo) = (?(Ob:o)red(?Ob=Te) and red(Fo[Ob]q)), if Foq = (=Te),

red(Fo) = (?(Ob:o)red(?Ob=Te) and red(Fo[Ob]q)), if Foq = (Te),

red(Fo) = (?(Ob:o)red(?Ob~Te) and red(Fo[Ob]q)), if Foq = (~Te),

red(Fo) = (?(Ob:CoExp)red(Fo[Ob]q)), if Foq = (*:CoExp),

red(Fo) = Fo, otherwise.

Example. Let an OTS specify a referral database. The formula (*:s surname Smith *:s) is equivalent to the formula (?(X:o Y:s Z:s)?X = Y surname Smith Z). It means that there exists at least one object with the surname Smith in the state of the referral database.

Example. Let an OTS specify a database of vacancies. The formula *:employer is equivalent to the formula (?(X:o)X:employer). It means that there exists at least one employer in the state of the database of vacancies.

7. Concept Declarations

This section presents concept declarations which are used to define concepts of OTSs, as well as the values of concepts.

Let CoDec be a concept declaration of the form #c Ob {Fo}. The object Ob is called a concept declared in the concept declaration CoDec. The formula Fo is called a specification of the concept Ob.

[37] ⊙ ∈ {=, ~}.
[38] X_q means a subtree of the tree X in the position q.
[39] $X[Y]_q$ means a tree which is obtained from the tree X by replacement of a subtree in the position q by the tree Y.

The set co(OtsDec) is called a set of concepts declared in the OTS declaration OtsDec, if

$$co(OtsDec) = \{Co \in ob \mid \exists[CoDec \in OtsDec, Fo](CoDec = \#c\ Co\ \{Fo\})\}.$$

Thus, co(OtsDec) is the set of concepts declared in concept declarations which are members of OtsDec.

The function $val \in co(OtsDec)ЧSt \rightarrow 2ob$ is called a meaning function declared by the OTS declaration OtsDec, if

$$val(Co,St) = \{Ob \mid \exists[CoDec \in OtsDec, Fo](CoDec = \#c\ Co\ \{Fo\} \wedge val(St)(Fo(\#i \leftarrow Ob)) \neq ())\}.$$

The special object #i is used in the specification Fo of the concept Co to refer to the sequences from the value of the concept Co which is declared in the concept declaration CoDec.

Example. The declaration #c emptyConcept {()} defines the concept emptyConcept. Its value is the empty set in any state.

Example. The declaration #c object {#:o} defines the concept object. The value of the concept object is the set of all objects in any state.

Example. The declaration #c document {#i:o and ?documents ~ #i *:s } defines the concept document. Instances of the concept document are defined as objects from the value of the object documents. For example, if the state St is defined by Table 1, then val(document, St) = {A, B}.

Example. The declaration #c document {#i:o and ?#i ~ document *:s} defines the concept document. Instances of the concept document are defined as objects such that their values include the object document. For example, if the state St is defined by Table 2, then val(document, St) = {A, B}.

Table 1

Object	Value
Documents	A B
A	B
B	A

Table 2

Object	Value
A	document B
B	C document A
C	B

Example. The declaration #c document {#i:o and ? #i = document *:s} defines the concept document. Instances of the concept document are defined as objects such that their values include the object document as the first element. For example, if the state St is defined by Table 1, then val(document, St) = {A}.

Example. The declaration #c makeReport {#i=makeReport} defines the concept makeReport. The value of the concept makeReport consists of one object makeReport for any state. Concepts of this form are used to represent procedures. Arguments of these procedures are specified by the values of the concepts for each state.

8. Relation Declarations

This section presents relation declarations which are used to define relations of OTSs, as well as the values of relations.

Let ReDec be a relation declaration of the form #r Ob {Fo}. The object Ob is called a relation declared in the relation declaration ReDec. The formula Fo is called a specification of the relation Ob.

The set re(OtsDec) is called a set of relations declared in the OTS declaration OtsDec, if

$$re(OtsDec) = \{Re \in ob \mid \exists[ReDec \in OtsDec, Fo](ReDec = \#c\ Re\ \{Fo\})\}.$$

Thus, re(OtsDec) is the set of relations declared in the relation declarations which are members of OtsDec.

The function $val \in re(OtsDec)ЧSt \rightarrow 2ob$ is called a meaning function declared by the OTS declaration OtsDec, if

$$val(Re, St) =$$

$$\{(Ob,Ob') \mid \exists[ReDec \in OtsDec,\ Fo](ReDec = \#r\ Re\ \{Fo\} \wedge val(St)(Fo(\#i \leftarrow Ob, \#o \leftarrow Ob')) \neq ())\}.$$

The special objects #i and #o are used in the specification Fo of the relation Re to refer to the first and the second components of pairs from the value of the relation Re which is declared in the relation declaration ReDec.

Example. The declaration #r synonym {#i:word and #o:word and ?synonymGroup ~ #i #o *:s} defines the relation synonym on words. According to this declaration, two words are synonyms, if they are both included in the value of the concept synonymGroup.

Example. The declaration

#r title {#i:document and #o:text and ?#i = source *:o title #o *:s}

defines the relation title. According to this declaration, the text #o is the title of the document #i in the state St, if source B title #o is a prefix of the value of #i for some object B. This form of representation of the relation value is used for concepts with a fixed order of attributes. In this example, source is the first attribute with the value B, title is the second attribute with the value #i, the rest of attributes represented by C follows title.

Example. The declaration #r title {#i:document and #o:text and ? #i □(= title #o) *:s} defines the relation title. According to this declaration, the text #o is the title of the document #i in the state St, if there is an object B such that B ∈ St(#i) and St(B) = title #o.

Example. The declaration

#r expression {#i:expressionStatement and #o:expression and ?#i = #o ;}

defines the relation expression. According to this declaration, the expression #o is an expression of the expression statement #i in the state St, if the value of #i in the state St has the form #o.

Conclusion

This paper is further development of the OTSL language intended for description of ontological transition systems. This language is divided into two sublanguages: a language of formulas and a language of actions. The language of actions which specify the transitions of OTSs was presented in [14]. The language of formulas which specify the ontological models of OTSs was presented in this paper.

The advantages of the language of formulas are as follows:

It has a formal semantics.

It defines a conceptual structure of states of OTSs.

It provides the use of natural intuitive terminology in specifications.

1. *Anureev I.S.* Ontological Transition Systems // Joint NCC&IIS Bulletin, Series Computer Science. – 2007. – Vol. 26. – P. 13 – 24.
2. *Gordon D.* Plotkin. A Structural Approach to Operational Semantics. (1981) Tech. Rep. DAIMI FN–19, Computer Science Department, Aarhus University, Aarhus, Denmark.
3. *Gurevich Y.* Abstract state machines: An Overview of the Project in "Foundations of Information and Knowledge Systems". Lect. Notes Comput. Sci. – 2004. – Vol. 2942. – P. 6–13.
4. *Gurevich Y.* Evolving Algebras. Lipari Guide // Specification and Validation Methods. – Oxford University Press, 1995. – P. 9–36.
5. *Stὸrk R., Bцrger E.* Java and the Java Virtual Machine: Definition, Verification, Validation. Springer–Verlag, 2001.
6. *Bцrger E., Fruja N.*, Vincenzo Gervasi, R. Stдrk: A high–level modular definition of C#. Theor. Comput. 2005. – Sci. 336(2/3). – P. 235–284.
7. *ITU*–T Recommendation Z.100 Annex F: SDL Formal Semantics Definition, International Telecommunications Union (ITU), Geneva, 2000.
8. *Gurevich Y., J. Huggins.* The Semantics of the C Programming Language. // Lect. Notes Comput. Sci. – 1993. – Vol. 702. – P. 274–309.
9. *Bцrger E., Rosenzweig D.* A Mathematical Definition of Full Prolog // Science of Computer Programming. – 1994. – Vol. 24. – P. 249–286.
10. *Kutter P., Pierantonio A.* The Formal Specification of Oberon // J. of Universal Computer Science. – 1997. – N 3(5). – P. 443–503.
11. *Huggins J.* Abstract State Machines Web Page. – http://www.eecs.umich.edu/gasm.
12. *AsmL*: The Abstract State Machine Language. – Reference Manual. 2002. – http://research.microsoft.com/fse/asml/doc/AsmL2_Reference.doc.
13. *XasM* – An Extensible, Component–Based Abstract State Machines Language. – http://xasm.sourceforge.net/XasmAnl00/XasmAnl00.html
14. *Anureev I.S.* A Language of Actions in Ontological Transition Systems // Joint NCC&IIS Bulletin, Series Computer Science. – 2007. – Vol. 26. P. 16 – 25.