Bull. Nov. Comp. Center, Comp. Science, 26 (2007), 19–38
© 2007 NCC Publisher

A language of actions in ontological transition systems

Igor S. Anureev

Abstract. Ontological transition systems are a method of specification of computer systems which integrates operational and ontological approaches to specification of these systems. In the framework of development of a language of ontological transition systems OTSL, a sublanguage of actions is defined. Actions are used to specify transitions in ontological transition systems. Examples of formal semantics of C# statements illustrate the method of ontological transition systems.

1. Introduction

Ontological transition systems (OTSs) [1] are an extension of transition systems with ontologies.

An ontology of a computer system describes the conceptual structure of the system (its concepts and the relations between them). In the context of this paper, an ontology 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.

An ontological transition system consists of a set of objects, a transition system, an ontology and a function, called content retrieval, which defines the content of concepts and relations for each state of the transition system. The content of a concept is defined as a subset of the set of sequences of objects. The content of a relation is defined as a binary relation on sequences of objects.

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 transitions of OTSs. Formulas specify ontological entities of OTSs. In this paper, the language of actions is presented. A description of the language of formulas can be found in [1]. The language of actions is a result of further development of ideas presented in [2, 3, 4].

The paper has the following structure. Section 2 presents the language of actions and defines their semantics. Actions are used to specify transition relations of OTSs. Section 3 presents additional constructs which can be used in actions. On the one hand, these constructs are reducible to the basic action constructs. On the other hand, they enlarge a conceptual capacity of the OTSL language. In Section 4, the OTSL language is used to define formal semantics for a number of C# statements.

The paper does not contain definitions of ontological transition systems and related notions, as well as preliminary notions and denotations, since all necessary notions and denotations are covered in paper [1] of this issue.

This research is partially supported by the grant 06-01-00464a from RFBR, and by the integration grant 14 and the grant for young scientists from SB RAS.

2. Actions

This section defines actions and related entities.

Actions. The set *act* of actions is built in the following way:

$$act ::= ass \mid gu \mid obAct \mid sel \mid qAct \mid blo \mid actCom.$$

The sets *ass*, *gu*, *obAct*, *sel*, *loc*, *qAct*, *blo*, and *actCom* of assignments, guards, object actions, selections, quantified actions, blocks and action compositions, respectively, are defined below.

Actions change states of OTSs. They are used as labels for transition relations. Therefore, semantics of an action is given by definition of the transition relation *tr* with this action as a label. Semantics of actions is defined in the context of an OTS declaration.

OTS declarations. OTS declarations are used to specify OTSs. The sets *otsDec* and *otsDecMem* of OTS declarations and OTS declaration members, respectively, are built in the following way:

$$otsDec ::= otsDecMem \mid otsDecMem\ otsDec.$$

$$otsDecMem ::= coDec \mid reDec \mid trDec.$$

The definitions of the sets *coDec* and *reDec* of concept declarations and relation declarations, respectively, can be found in [1]. The set *trDec* of transition declarations is defined below.

Transition Declarations. The set *trDec* of transition declarations is defined as follows:

$$trDec ::= \#t\ \{\ act\ \}.$$

Let *OtsDec* be an OTS declaration. Let us define different kinds of actions in the context of *OtsDec*.

Assignments. The set *ass* of assignments is built in the following way:

$$ass ::= sAss \mid cAss.$$

The sets $sAss$ and $cAss$ of simple assignments and compound assignments, respectively, are defined below. An assignment changes the content of objects.

Simple Assignments. The set $sAss$ of simple assignments is built in the following way:

$$sAss ::= ob\ \ :=\ \ te\ ;\ \mid obCon\ \ :=\ \ te\ ;\ .$$

A simple assignment replaces the content of an object, which is the value of the left-hand part of the assignment, by the value of its right-hand part:

$$\begin{array}{l} tr(St, St')(T\ \ :=\ \ T';) \Leftrightarrow \\ val(St)(T) \in ob \wedge St' = upd(St, val(St)(T), val(St)(T')). \end{array}$$

Compound Assignments. The set $cAss$ of compound assignments is built in the following way:

$$cAss ::= ob \mathrel{+=}\ te\ ;\ \mid obCon\ \ :=\ \ te\ ;.$$

A compound assignment replaces the content of an object, which is the value of the left-hand part of the assignment, by a concatenation of the old content of the object with the value of the right-hand part of the assignment:

$$\begin{array}{l} tr(St, St')(T\ \ :=\ \ T';) \Leftrightarrow \\ val(St)(T) \in object \wedge \\ St' = upd(St, val(St)(T), con(St(val(St)(T)), val(St)(T')). \end{array}$$

Guards. The set gu of guards is built in the following way:

$$gu ::= fo\ ;.$$

A guard defines a condition of execution of a transition. If the formula defined in the guard is true, execution of the transition is possible. Otherwise, execution of the transition is impossible:

$$tr(St, St')(F;) \Leftrightarrow St = St' \wedge val(St)(F) \neq ().$$

Object Actions. The set $obAct$ of object actions is built in the following way:

$$obAct ::= te\ ;;.$$

An object action $Ob;;$ performs a transition initiated by the object Ob:

$$tr(St, St')(Ob; ;) \Leftrightarrow \exists TrDec \in OtsDec(tr(St, St')(TrDec(\#i \leftarrow Ob))).$$

The special object $\#i$ is used in the transition declaration $TrDec$ to refer to the object which initiates the transition.

The definition of an object action is extended to the empty sequence

$$tr(St, St')((); ;) \Leftrightarrow St = St',$$

an object content

$$tr(St, St')(?Ob; ;) \Leftrightarrow tr(St, St')(St(Ob); ;),$$

and a term composition

$$\begin{array}{l} tr(St, St')(T\ T'; ;) \Leftrightarrow \\ \exists St''(tr(St, St'')(T; ;) \wedge tr(St'', St')(T'; ;)). \end{array}$$

The Selection. The sets sel, $casSe$, cas, $eCasSe$, $eCas$, and $casSt$ of selections, case sequences, cases, else case sequences, else cases, and case stops, respectively, are built in the following way:

$$\begin{array}{c} sel ::= \{\ cSe\ eCSe_{opt}\ \} \\ casSe ::= cas \mid cas\ casSe \\ cas = @\ act \\ eCasSe ::= eCas \mid eCas\ eCasSe \\ eCas ::= \#\ act \\ caseSt ::= \#\ ;. \end{array}$$

A selection performs a choice from the set of cases and else cases. It is a generalization of selection statements of programming languages. Each case and each else case is associated with a condition of its applicability. This condition is an action and is defined by the function $cond$ (see below). A case or an else case with a condition P is applicable in a state St, if the formula $\exists St'(tr(St, St')(P))$ is true.

Selection is executed as follows. First, a nondeterministic choice of an applicable case from the set of cases is performed and the chosen case is executed. If none of cases is applicable, then the nearest (in the sequence order) applicable else case is chosen and executed. If none of cases and else cases is applicable, execution of the selection is impossible:

$$\begin{array}{l} tr(St, St')(\{A\}) \Leftrightarrow \\ \quad \exists q(A_q \in cas \wedge tr(St, St')(A_q)) \vee \\ \quad \exists q(A_q \in eCas \wedge tr(St, St')(A_q) \wedge \\ \quad\quad \forall q'(q' \prec_c q \Rightarrow \neg tr(St, St')(cond(A_{q'})))). \end{array}$$

***The function** cond.* The property A of a case or else case is computed by the function $cond \in cas \cup eCas \to act$. The function $cond$ eliminates the parts of A which follow the case stops $\#;$, if these case stops do not occur in formulas or selections which are parts of A:

$$cond(A) = \begin{cases} cond(A[P]_q), & \text{if } \begin{array}{l} \exists q((A_q = P\ \#;\ P')\wedge \\ \quad \nexists q'(q \prec q' \wedge A_{q'} \in sel \cup fo)); \end{array} \\ A, & \text{otherwise.} \end{cases}$$

Case Sequences. Semantics of case sequences is defined as follows:

$$\begin{array}{l} tr(St, St')(CasSe\ CasSe') \Leftrightarrow \\ \exists St''(tr(St, St'')(CasSe) \wedge tr(St'', St')(CasSe')). \end{array}$$

Cases. Semantics of cases is defined as follows:

$$tr(St, St')(@\ Act) \Leftrightarrow tr(St, St')(Act).$$

Else Case Sequences. Semantics of else case sequences is defined as follows:

$$\begin{array}{l} tr(St, St')(ECasSe\ ECasSe') \Leftrightarrow \\ \exists St''(tr(St, St'')(ECasSe) \wedge tr(St'', St')(ECasSe')). \end{array}$$

Else Cases. Semantics of else cases is defined as follows:

$$tr(St, St')(\#\ Act) \Leftrightarrow tr(St, St')(Act).$$

Case Stops. The case stop marks the end of a case condition and does not influence execution of an action:

$$tr(St, St')(\#;) \Leftrightarrow St = St'.$$

Quantified Actions. The set $qAct$ of quantified actions is built in the following way:

$$\begin{array}{c} qAct ::= \{\ ?\ (\ bin\ )\ act\ \} \mid \{\ !\ (\ bin\ )\ act\ \} \\ bin ::= ob\ :\ coExp \mid bin\ bin. \end{array}$$

The elements of the set bin are called bindings. A quantified action

$$\{*\ (Ob_1 : CoExp_1 \ldots Ob_n : CoExp_n)Act\},$$

where $* \in \{?, !\}$, defines the parameters $Ob_1, \ldots, Ob_n$ for the embedded action Act.

If $* = ?$, then semantics of the quantified action is a union of semantics of actions which is obtained from the action Act by replacement of the parameters $Ob_1, ..., Ob_n$ by arbitrary values from sets defined by the concept expressions $CoExp_1, ..., CoExp_n$, respectively. Thus, this case models a nondeterministic choice from specializations of the action Act defined by the values of its parameters:

$$tr(St, St')(\{?(Ob_1 : CoExp_1 ... Ob_n : CoExp_n)Act\}) \Leftrightarrow \\ \exists[A_1 \in val(St)(CoExp_1), ..., A_n \in val(St)(CoExp_n)] \\ (tr(St, St')(Act(Ob_1 \leftarrow A_1, ..., Ob_n \leftarrow A_n))).$$

If $* = !$, then semantics of the quantified action defines a set of pairs (St, St') such that a transition from the state St to the state St' is possible for any specialization of the action Act defined by the values of its parameters:

$$tr(St, St')(\{!(Ob_1 : CoExp_1 ... Ob_n : CoExp_n)Act\}) \Leftrightarrow \\ \forall[A_1 \in val(St)(CoExp_1), ..., A_n \in val(St)(CoExp_n)] \\ (tr(St, St')(Act(Ob_1 \leftarrow A_1, ..., Ob_n \leftarrow A_n))).$$

Blocks. The set *blo* of blocks is built in the following way:

$$blo ::= \{\ act\ \}.$$

Blocks are used to structure actions. Semantics of a block coincides with semantics of the action defined in the block:

$$tr(St, St')(\{A\}) \Leftrightarrow tr(St, St')(A).$$

Action Compositions. The set *actCom* of action compositions is built in the following way:

$$actCom ::= act\ act.$$

An action composition $Act\ Act'$ is defined as the sequential execution of actions Act and Act':

$$tr(St, St')(Act\ Act') \Leftrightarrow \exists St''(tr(St, St'')(Act) \wedge tr(St'', St')(Act')).$$

3. Additional action constructs

This section presents additional constructs which can be used in actions. On the one hand, these constructs are reducible to the basic action constructs. On the other hand, they enlarge the conceptual capacity of the OTSL language. These constructs include localization and new objects. Localizations are used in place of actions. New objects are used in place of terms. To introduce them, the sets *act* and *te* of actions and terms, respectively, are redefined.

Redefining Actions. The set *act* of actions is built in the following way:

$$act ::= ass \mid gu \mid obAct \mid sel \mid qAct \mid blo \mid actCom \mid loc.$$

The set *loc* of localizations is defined below.

Localizations. The set *loc* of localizations is built in the following way:

$$loc ::= \{\ :\ \ (\ obList\ )\ act\ \}\ |\ \{\ :\ \ ob\ act\ \}$$
$$obList ::= ob\ |\ ob\ obList.$$

A localization $\{: (Ob_1\ \dots\ Ob_n)\ Act\}$ guarantees that the change of the content of the objects $Ob_1, \dots, Ob_n$ by the action Act is local, i.e. the content is restored after termination of the action Act. Its semantics is defined by the reduction function $red : act \to act$ which normalizes actions, eliminating localizations:

$$red(Act) = \begin{cases} red(Act[\{?(X_1 : s\ \dots\ X_n : s) & \\ \quad X_1 = ?Ob_1;\ \dots\ X_n = ?Ob_n;\ Act' & \\ \quad Ob_1 := X_1; \dots Ob_n := X_n;\}]_q), & \text{if } \exists q(Act_q = \{: (Ob_1\ \dots\ Ob_n)\ Act'\}); \\ Act, & \text{otherwise.} \end{cases}$$

The localization $\{: Ob\ Act\}$ is short for $\{: (Ob)\ Act\}$.
Redefining Terms. The set te of terms is built in the following way:

$$te ::= eSe\ |\ ob\ |\ obC\ |\ teCom\ |\ newOb.$$

The set $newOb$ of new objects is defined below.
New Objects. The set $newOb$ of new objects is built in the following way:

$$newOb ::= (\ te\ )\ |\ *\ :\ coExp.$$

A new object (Te) in a state St represents any object Ob taken from the content of a special object new in the state St such that the content of the object Ob is equal to the value of the term Te in the state St, i.e.

$$Ob \in St(new) \wedge St(Ob) = val(St)(Te).$$

A new object $* : CoExp$ in a state St represents any object Ob taken from the content of the special object new in the state St such that the content of the object Ob belongs to the value of the concept expression $CoExp$ in the state St, i.e.

$$Ob \in St(new) \wedge St(Ob) \in val(St)(CoExp).$$

After taking the object Ob, this object is removed from the content of new. The object Ob is implicitly bound by an existential quantifier ?, that guarantees existence of at least one new object in the content of new.
Elimination of new object constructs. Semantics of new object constructs is defined by the reduction function

$$red \in ass \cup obAct \to qAct.$$

This function normalizes assignments and object actions, eliminating new objects from them:

$$red(Act) = \begin{cases} \{?(Ob{:}o\ Se{:}s)\ ?new \sim Ob\ Se; & \\ \quad new := Se;\ red(Ob := Te;) & \\ \quad red(Act[Ob]_q)\}, & \text{if } Act_q = (Te), \\ \{?(Ob{:}o\ Se{:}s\ Se'{:}CoExp) & \\ \quad ?new \sim Ob\ Se;\ new := Se; & \\ \quad red(Ob := Se';)\ red(Act[Ob]_q)\}, & \text{if } Act_q = *{:}CoExp, \\ Act, & \text{otherwise.} \end{cases}$$

4. Examples of formal semantics of C# statements

This section illustrates the method of OTSs by an example of development of formal semantics of a number of C# statements.

The concept `statement` is defined by the concept declaration

```
#dc statement {#i:o and (#i:block or #i:ifStatement or otherStat)}
```

The record `otherStat` denotes disjunction of the other kinds of C# statements.

Semantics of blocks is defined in detail to understand a general idea of application of the method of OTSs. Semantics of the rest statements is given without additional comments.

4.1. Blocks

Semantics of blocks is defined by the following declarations:

```
#c block {#i:o and ?#i = block *:s}
#r statements {#i:block and #o:s and ?#i ~ (= statements #o) *:s}
#t {#i:block;
 {?(A:statements<#i)
  {@val:jump or A:e;
   #A;; {:blockStop blockStop := (statements A);
          blockStop;;}}}}
```

The first declaration defines the concept `block`.

The formula `#i:o` means “The block `#i` of the concept `block` is an object”.

The formula `?#i = block *:s` means “The content of `#i` starts with the object `block`”. The object `block` can be considered as a tag which marks all instances of the concept `block`.

The second declaration defines the relation `statements`.

The formula `#i:block` means “The input `#i` of instances of the relation `statements` is a block”.

The formula `#o:s` means “The output `#o` of instances of the relation `statements` is a sequence”.

The formula

```
?#i ~ (= statements #o) *:s}
```

means “The output `#o` is the statement list of the block `#i`, if there is an object `X` such that `X` is contained in the content of the block `#i` and the content of `X` starts with the object `statements` and `#o` follows `statements`.” The object `X` can be considered as a container which contains the statement list of the block `#i`. The object `statements` can be considered as a tag which marks the container X.

Also, the object `statements` can be considered as an attribute of instances of the concept `block`. The value of this attribute for the block `#i` in a state `St` is defined as the only element of the set `St(statements<#i)`.

The third declaration defines transitions which are initiated by blocks.

The guard `#i:block;` means “Only if `#i` is a block, a transition is possible”.

The quantified action {`?(A:statements<#i) X`} means “Let `A` be a statement list of the block `#i` in the action `X` (`A` is the value of the attribute `statements` for `#i`)”.

The case `@val:jump or A:e;` of the selection action means “If the object `val` is an instance of the concept `jump` or the statement list `A` is empty, then do nothing”.

The concept `jump` describes a kind of objects which are initiated by jump statements of the C# language:

```
#c jump {#i:breakJump or #i:continueJump or #i:returnJump or
 #i:exceptionJump or #i:labelJump or #i:caseJump or
 #i:defaultJump}
```

Instances of the concept `breakJump` are initiated by break statements. Instances of the concept `continueJump` are initiated by continue statements. Instances of the concept `returnJump` are initiated by return statements. Instances of the concept `exceptionJump` are initiated by throw statements and execution environment. Instances of the concepts `labelJump`, `caseJump` and `defaultJump` are initiated by different kinds of goto statements.

The object `val` is a special object. It is used to keep the values of C# expressions and instances of the concept `jump`.

The else case

```
#A;; {:blockStop blockStop := (statements A); blockStop;;}
```

of the selection action means ”Otherwise, execute the statement list `A` and exit the block by the procedure `blockStop`”.

The localization

```
{:blockStop blockStop := (statements A); blockStop;;}
```

locally sets the content of the procedure `blockStop` (which can be considered as the argument of `blockStop`) to the statement list `A`, executes `blockStop`, and restores the old value of the content of `blockStop`.

The procedure `blockStop` is defined by the following declarations:

```
#c blockStop {#i = blockStop}
#r statements {#i:blockStop and #o:ns and ?#i = (statements #o)}
#t {#i:blockStop;
 {@val:labelJump;
   {?(A:gotoLabel B:value<A B:value<val
      C:statements<#i D:s)
    C = *:s A D; #; D;; blockStop;;}
  #}}
```

The first declaration defines the concept `blockStop`.

The content declarator `#i = blockStop` means "The content of the concept `blockStop` is a set consisting of the only object `blockStop` in any state".

The second declaration defines the relation `statements`. As shown above, the object `statements` can be considered as an attribute of the instance `blockStop` of the concept `blockStop`.

The third declaration defines transitions which are initiated by the procedure `blockStop`. For simplicity, elimination of local variables of blocks is not considered.

The quard `#i:blockStop;` means "Only if `#i` is a block stop, this transition is possible".

The case

```
@val:labelJump;
 ...
```

of the selection action means "If the object `val` is an instance of the concept `labelJump` (`val:labelJump;`), the statement list `C` of the block `#i` contains the goto label `A` (`C = *:s A D;`), the value `B` of `A` (`B:value<A`) coincides with the value of the attribute `value` for the object `val`, then execute the sequence `D` of statements (`D;;`) which follows `A` (`C = *:s A D;`) and then execute the procedure `blockStop` again (`blockStop;;`).

The concept `labelJump` describes a kind of objects which are generated by goto statements. These objects have the attribute *value*. The value of the attribute is the label of a goto statement. The value of the attribute for an object `Ob` in a state *St* is defined as `St(value<Ob)`:

```
#c labelJump {#i:o and ?#i = labelJump *:s}
#r value {#i:labelJump and #o:o and ?#i ~ (value #o) *:s}
```

The else case `#` of the selection statement means "Otherwise, do nothing".

4.2. Empty statements

Semantics of empty statements is defined by the following declarations:

```
#c emptyStatement {#i = emptyStatement}
#ru {.:emptyStatement;}
```

4.3. Labeled statements

Semantics of labeled statements is reduced to semantics of labels:

```
#c gotoLabel {#i:o and ?#i = gotoLabel *:s}
#r value {(gotoLabel i)}{?#i ~ (gotoLabel #o) *:s}
#t {.:gotoLabel;}
```

4.4. Declaration statements

Semantics of local variable declarations is defined by the following declarations:

```
#c localVariableDeclaration
 {#i:o and ?#i = localVariableDeclaration *:s}
#r type
 {#i:localVariableDeclaration and #o:s and ?#i ~ (type #o) *:s}
#r localVariableDeclaratorList
 {#i:localVariableDeclaration and #o:ns and
  ?#i ~ (localVariableDeclaratorList #o) *:s}
#c type {#i = type}
#r value {(type i)}{#i:type and #o:o and ?#i ~ (value #o) *:s}
#t {#i:localVariableDeclaration;
 {@val:jump;
  #{?(A:type<#i B:localVariableDeclaratorList<#i)
    {:type type = A; B;;}}}}
```

Formal semantics of local variable declarators is defined by the following declarations:

```
#c localVariableDeclarator
 {#i:o and ?#i = localVariableDeclarator *:s}
#r name
 {#i:localVariableDeclarator and #o:identifier and
  ?#i ~ (= name #o) *:s}
#r initializer
 {#i:localVariableDeclarator and
  (#o:expression or #o:arrayInitializer) and
  ?#i ~ (= initializer #o) *:s}
```

```
#c localVariable {#i:o and ?#i = localVariable *:s}
#r name
 {#i:localVariable and #o:identifier and ?#i ~ (name #o) *:s}
#r type
 {#i:localVariable and #o:type and ?#i ~ (type #o) *:s}
#r location
 {#i:localVariable and #o:location and ?#i ~ (location #o) *:s}
#r value
 {#i:localVariable and #o:o and (?(A:location<#i) #o:value<A)}


#c location {#i:o and ?#i = location *:s}
#r value {#i:location and #o:o and ?#i ~ (= value #o) *:s}

#t {#i:localVariableDeclarator;
 {@val:jump;
  #{?(A:name<#i B:value<type C:o)
    {:val val := (location); C = ?val;
     val := (localVariable (name A) (type B) (location C));}
    {@{?(D:initializer<#i) D;;
       {@val:jump;
        #C += (value ?val);}}
     #}}}}
```

Semantics of local constant declarations is defined in a similar way.

4.5. Expression statements

Semantics of local variable declarations is defined by the following declarations:

```
#c expressionStatement {#i:o and ?#i = expressionStatement *:s}
#r expression {#i:expressionStatement and #o:expression and
 ?#i = (expression #o) *:s}
#t {#i:expressionStatement; {?(A:expression<#i) A;;}}
```

4.6. If statements

Semantics of if statements is defined by the following declarations:

```
#c ifStatement {#i:o and ?#i = ifStatement *:s}
#r condition {#i:ifStatement and #o:expression and
 ?#i ~ (condition #o) *:s}
#r thenStatement {#i:ifStatement and #o:statement and
 ?#i ~ (thenStatement #o) *:s}
```

```
#r elseStatement {#i:ifStatement and #o:statement and
 ?#i ~ (elseStatement #o) *:s}
#t {#i:ifStatement;
 {@val:jump;
  #{?(A:condition<#i) A;;
    {@val:jump;
     #{@?val = true; #; {?(B:thenStatement<#i) B;;}
       @val = false;
        {@{?(B:elseStatement<#i) #; B;;}
         #}}}}}}
```

4.7. Switch statements

Semantics of switch statements is defined by the following declarations:

```
#c switchStatement {#i:o and ?#i = switchStatement *:s}
#r switchExpression {#i:switchStatement and #o:expression and
 ?#i ~ (switchExpression #o) *:s}
#r switchBlock {#i:switchStatement and #o:switchBlock and
 ?#i ~ (switchBlock #o) *:s}

#c switchBlock {#i:o and ?#i = switchBlock *:s}
#r statementList {#i:switchStatement and #o:expression and
 ?#i ~ (statementList #o) *:s}

#c caseLabel {#i:o and ?#i = caseLabel *:s}
#r value {#i:caseLabel and #o:o and ?#i ~ (value #o) *:s}

#c defaultLabel {#i = defaultLabel}

#c governingTypeConversion {#i = governingTypeConversion}
#r value {#i:governingTypeConversion and #o:type and
 ?#i ~ (value #o) *:s}

#t {#i:switchStatement;
 {@val:jump;
  #{?(A:switchExpression<#i) A;;
    {@val:jump;
     #{?(B:switchBlock<#i C:statementList<B)
       {:governingTypeConversion
        governingTypeConversion := (value ?val);
        governingTypeConversion;}
        {@{@{?(D:caseLabel D:value>?val E:s)
             ?C = *:s D E; #; E;;}
```

```
          #{?(D:s) ?C = *:s *:defaultLabel D; #; D;;}}
         #; {:switchBlockStop
             switchBlockStop := (statements C);
             switchBlockStop;;}}
        #}}}}}}
```

The procedure `switchBlockStop` is defined as follows:

```
#c switchBlockStop {#i:o and ?#i = switchBlockStop *:s}
#r statements {#i:switchBlockStop and #o:ns and
 ?#i ~ (statements #o) *:s}
#t {#i:switchBlockStop;
 {@val:breakJump; #; val := ();
  #val:labelJump;
    {?(A:label<val B:gotoLabel B:value>A C:s
     D:statementList<#i) D = *:s B C; #; C;; #i;;}
  #val:caseJump; #;
    {?(A:value<val B:caseLabel B:value>A C:s
     D:statementList<#i) D = *:s B C; #; C;; #i;;}
  ##defaultJump(val); #;
    {?(A:s B:statementList<.)
     B = *:s *:defaultLabel A; A;; #i;;}
  #}}
```

5. While statements

Semantics of switch statements is defined by the following declarations:

```
#c whileStatement {#i:o and ?#i = whileStatement *:s}
#r condition {#i:whileStatement and #o:expression and
 ?#i ~ (condition #o) *:s}
#r body {#i:whileStatement and #o:statement and
 ?#i ~ (body #o) *:s}
#t {#i:whileStatement;
 {@val:jump;
  #{?(A:condition<#i B:body<#i)
    {:loop loop := (condition A) (body B); loop;;}}}}
```

Semantics of the procedure `loop` is defined as follows:

```
#c loop {#i = loop}
#r condition {#i:loop and #o:expression and
 ?#i ~ (condition #o) *:s}
#r body {#i:loop and #o:statement and ?#i ~ (body #o) *:s}
#t {#i:loop;
```

```
 {@val:breakJump; #; val := ();
  #val:continueJump; #; val := (); loop;;
  #val:jump;
  #{?(A:condition<#i) A;;
    {@not val:jump and ?val = true; #; {?(B:body<#i) B;; .;;}
     #}}}}
```

Semantics of dowhile statements, for statements and foreach statements are defined in a similar way.

5.1. Break statements

Semantics of break statements is defined by the following declarations:

```
#c breakStatement {#i = break}
#c breakJump {#i:o and ?#i = breakJump *:s}
#t {#i:breakStatement;
{@val:jump;
 #val := breakJump;}}
```

5.2. Continue statements

Semantics of continue statements is defined by the following declarations:

```
#c continueStatement {#i = continue}
#c continueJump {#i:o and ?#i = continueJump *:s}
#t {#i:continueStatement;
{@val:jump;
 #val := continueJump;}}
```

5.3. Goto statements

Semantics of goto statements is defined by the following declarations:

```
#c gotoLabelStatement {#i:o and ?#i = gotoLabel *:s}
#r label {#i:gotoLabelStatement and #o:o and
 ?#i ~ (= label #o) *:s}
#c labelJump {#i:o and ?#i = labelJump *:s}
#r value {#i:labelJump and #o:o and ?#i ~ (= value #o) *:s}
#t {#i:gotoLabelStatement;
 {@val:jump;
  #{?(A:label<#i) val := labelJump (value A);}}}

#c gotoCaseStatement {#i:o and ?#i = gotoCase *:s}
#r expression {#i:gotoCaseStatement and #o:expression and
```

```
 ?#i ~ (= expression #o) *:s}
#r value {#i:gotoCaseStatement and #o:o and
 ?#i ~ (= value #o) *:s}
#c caseJump {#i:o and ?#i = caseJump *:s}
#r value {#i:caseJump and #o:o and ?#i ~ (= value #o) *:s}
#t {#i:gotoCaseStatement;
{@val:jump;
 #{?(A:value<#i) val := caseJump (value A);}}}

#c gotoDefaultStatement {#i = gotoDefault}
#c defaultJump {#i = defaultJump}
#t {#i:gotoDefaultStatement;
{@val:jump;
 #val := defaultJump;}}
```

5.4. Return statements

Semantics of return statements is defined by the following declarations:

```
#c returnStatement {#i:o and ?#i = return *:s}
#r expression {#i:returnStatement and #o:o and
 ?#i ~ (= expression #o) *:s}
#c returnJump {#i:o and ?#i = returnJump *:s}
#r value {#i:returnJump and #o:o and ?#i ~ (= value #o) *:s}

#c conversion {#i = conversion}
#r value {#i:conversion and #o:o and ?#i ~ (= value #o) *:s}
#r type {#i:conversion and #o:type and ?#i ~ (= type #o) *:s}
#c returnType {#i = returnType}
#r value {#i:returnType and #o:type and ?#i ~ (= value #o) *:s}
#t {#i:returnStatement;
{@val:jump;
 #{?(A:expression<#i) #; A;;
   {@val:jump;
    {:conversion {?(B:value<returnType) conversion := ?val B}
     conversion;;}
    #val := returnJump (value ?val);}}
 #val := returnJump;}}
```

5.5. Throw statements

Semantics of throw statements is defined by the following declarations:

```
#c throwStatement {#i:o and ?#i = throw *:s}
#r expression {#i:throwStatement and #o:expression and
 ?#i ~ (= expression #o) *:s}
#c exceptionJump {#i:o and ?#i = exceptionJump *:s}
#r exception {#i:exceptionJump and #o:o and
 ?#i ~ (= exception #o) *:s}
#c catchedException {#i = catchedException}
#c nullReferenceException {#i = nullReferenceException}
#t {#i:throwStatement;
{@val:jump;
 #{?(A:expression<#i) #; A;;} #;
  {@val:jump;
   #val = null; #; val := nullReferenceException;
   #val := exceptionJump (exception ?val);}
 #val := exceptionJump (exception ?catchedException);}}
```

5.6. Try statements

Semantics of try statements is defined by the following declarations:

```
#c tryStatement {#i:o and ?#i = try *:s}
#r block {#i:tryStatement and #o:statement and
 ?#i ~ (= block #o) *:s}
#r catchClauseList {#i:tryStatement and #o:ns and
 ?#i ~ (= catchClauseList #o) *:s}
#t {#i:tryStatement;
{@val:jump;
 #{(A:block<#i B:catchClauseList<#i) A;;
   {:catchedException catchedException := (); B;;}}}}
```

The value of the attribute `catchClauseList` is a sequence of specific catch clauses, a general catch clause, and a finally clause.

Semantics of specific catch clauses is defined as follows:

```
#c specificCatchClause {#i:o and ?#i = specificCatchClause *:s}
#r type {#i:specificCatchClause and #o:type and
 ?#i ~ (= type #o) *:s}
#r name {#i:specificCatchClause and #o:identifier and
 ?#i ~ (= name #o) *:s}
#r block {#i:specificCatchClause and #o:block and
 ?#i ~ (= block #o) *:s}
#r subtype {#i:type and #o:type and ...}
```

```
#t {#i:specificCatchClause;
{@not val:exceptionJump or ?catchedException;
 #{?(A:type<#i B:i C:i) B = ?val;
   {:type type := B; type;;} C = ?val;
   {@C:subtype<A; #; catchedException := B;
     {@{?(D:name<#i) #;
       val := (localVariable D C (location B));}
      #}
     val := (); {?(E:block<#i) E;;}
    #}}}}
```

Semantics of general catch clauses is defined as follows:

```
#c generalCatchClause {#i:o and ?#i = generalCatchClause *:s}
#r block {#i:generalCatchClause and #o:block and
 ?#i ~ (= block #o) *:s}
#t {#i:generalCatchClause;
{@not val:exceptionJump or ?catchedException;
 #catchedException := ?val; val := (); {?(A:block<#i) A;;}}}
```

Semantics of finally clauses is defined as follows:

```
#c finallyClause {#i:o and ?#i = finallyClause *:s}
#r block {#i:finallyClause and #o:block and
 ?#i ~ (= block #o) *:s}
#c exceptionBeforeFinally {#i = exceptionBeforeFinally}
#t {#i:generalCatchClause;
{:exceptionBeforeFinally
 {@val:jump; exceptionBeforeFinally := ?val;
  #?catchedException;
   exceptionBeforeFinally := exceptionJump ?catchedException;
  #exceptionBeforeFinally := ();}
 val := (); {?(A:block<#i) A;;}
 {@val:jump;
  #val := ?exceptionBeforeFinally;}}}
```

5.7. Checked and unchecked statements

Semantics of checked and unchecked statements is defined as follows:

```
#c checkedStatement {#i:o and ?#i = checkedStatement *:s}
#r block {#i:checkedStatement and #o:block and
 ?#i ~ (= block #o) *:s}
#t {#i:checkedStatement;
{@val:jump;
 #{?(A:block<#i)
```

```
   {:checkedContext {checkedContext := true; A;;}}}}}

#c uncheckedStatement {#i:o and ?#i = uncheckedStatement *:s}
#r block {#i:uncheckedStatement and #o:block and
 ?#i ~ (= block #o) *:s}
#t {#i:uncheckedStatement;
{@val:jump;
 #{?(A:block<#i)
   {:uncheckedContext {checkedContext := (); A;;}}}}}
```

6. Conclusion

This paper is further development of the language OTSL destined for description of ontological transition systems. This language is divided into two sublanguages: a language of formulas and a language of actions. The sublanguage of formulas which specify the ontological entities in OTSs was presented in [1]. A sublanguage of actions which specify the transitions in OTSs was presented in this paper.

The advantages of the language of actions are as follows:

- the language of actions has a formal operational semantics;
- this language is a verification-oriented language, since the operational semantics of this language is easily inserted to a first-order program logics;
- integration of languages of actions and formulas in ontological transition systems allows us to change the content of concepts and relations during transitions.

At present, we use our method to define a complete operational semantics of the sequential subset of C# language in the framework of the project of C# program verification [5].

References

[1] Anureev I.S. Ontological Transition Systems // Bull. Novosibirsk Comp. Center. Ser. Computer Science. — 2007. — IIS Special Iss. 26. — P. 1–17.

[2] Anureev I.S. Unified Semantic Language: Syntax, Semantics, and Pragmatics // Joint NCC & IIS Bull. Ser. Computer Science. — 2004. — Iss. 20. — P. 1–30.

[3] Anureev I. The language of natural state machines USL. — Novosibirsk, 2004. — 25 p. — (Prep. / IIS SB RAS; N 114).

[4] Anureev I.S. An Approach to Formal Human-Oriented Specifications of Programming Languages // Proc. Workshop on Concurrency, Specification and Programming (CS&P'2006), Humboldt University, Berlin, 2006.

[5] Nepomniaschy V.A., Anureev I.S., Dubranovsky I.V., Promsky A.V. A Three-Level Approach to C# Program Verification // Joint NCC&IIS Bulletin. Ser. Computer Science. — 2004. — Iss. 20. — P. 61–85.