Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

I. S. Anureev

USL — THE LANGUAGE OF NATURAL STATE MACHINES

Preprint
114

Novosibirsk 2004

An approach to description of formal operational semantics of modern programming languages is proposed. The approach yields specifications of programming languages syntactically close to specifications in natural languages, but with a formal semantics. The approach is based on natural state machines, a new class of abstract machines. The language of natural state machines called USL is presented and its formal semantics is defined. A library of USL instructions destined for development of programming language semantics is outlined. The examples of application of the approach to the modern programming languages are given.

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

И. С. Ануреев

ЯЗЫК МАШИН ЕСТЕСТВЕННЫХ СОСТОЯНИЙ USL

Препринт
114

Новосибирск 2004

Предложен новый подход к описанию формальной операционной семантики современных языков программирования. Этот подход позволяет задавать спецификации языков программирования в виде текста, близкого к естественному языку, но имеющего формальную семантику. Он основан на машинах естественных состояний, новом классе абстрактных машин. Представлен язык машин естественных состояний, названный USL, и определена его формальная семантика. Дан обзор библиотеки инструкций этого языка, предназначенных для разработки семантики языков программирования. Рассмотрены примеры применения подхода к современным языкам программирования.

1. INTRODUCTION

In his survey of programming language semantics [1], Peter Mosses states that one of the main hindrances to greater use of formal semantics in practical applications is a lack of user-friendliness of most semantic frameworks. The same is true for semantics description languages based on these frameworks.

The languages AsmL [2], Montages [3], VDM-SL [4] and AN2 [5], most applicable to real-life programming languages, have conventional user-friendly features, such as modularity and compositionality, to a greater or lesser extent.

We focus on two extra user-friendly features of a semantics description language, its compactness and closeness of the described semantics to a natural language text. None of the above-mentioned languages turns out to possess these features.

To fill up the gap, we suggest a new very compact language called USL (Unified Semantic Language) [6] that defines a programming language semantics in a form close to the natural language texts.

A semantic framework of the USL language is natural state machines (NSMs), a version of abstract machines in which states are defined in terms of sentences close to natural language sentences.

This research has been partially supported by a gift from Microsoft Research in 2002 and is partially supported by the RFBR grant 04-01-00114a.

2. PRELIMINARIES

2.1. Extended Backus-Naur Formalism

To give syntactic notions, we use a variant of Backus-Naur formalism. A non-terminal is defined as follows:
Non-terminal ::= definition
where constraint$_1$,..., constraint$_n$.

A definition is a sequence of terminals, variables and the symbols ::=, |, (,), [,]. Alternatives are separated by a vertical bar |. Round brackets (and) are used for grouping. Square brackets [and] indicate that the enclosed instruction is optional. If the symbols ::=, |, (,), [,] are used as terminals, they are enclosed in quotes. For example, "(". Terminals and nonterminals are in small letters. Variables are in capital letters.

Constraints are given in any of the three forms

- non-terminal ::= VARIABLE,
- non-terminal ::=
  VARIABLE$_1$, E, VARIABLE$_n$ or
- VARIABLE is predicate.

Predicates are functions that return the value true or false.

Example. The grammar of arithmetical instructions without a priority of the operations + and * is defined as follows:
instruction ::=
V | C | X + Y | X * Y | "(" X ")" where
variable ::= V, constant ::= C, instruction ::= X,Y

variable ::= X where X is variable

constant ::= X where X is constant

2.2. Mathematical notation

Let M be a set and f be a function.

Let 2^M denote a set of all subsets of M, dom of f denote the domain of f and range of f denote the range of f.

3. NATURAL STATE MACHINE SYNTAX

This section contains the main syntactic notions associated with NSMs.

3.1. Natural state machine signatures, sentences

A natural state machine signature Sig is a pair (WS,BPS) of sets. The elements of the sets WS are called words. The set BPS consists of the pairs of elements called left and right brackets, respectively. It includes the pair ((,)). The sets of words and brackets are disjoint.

The set SS of sentences of the signature Sig is defined as follows:
sentence ::= (X | Y Z | L Y R | L R)
where X is word, sentence ::= Y, Z,
(L, R) is bracket pair

Let WordSet(S) be a set of all words of S∈SS. For the set M of sentences, WordSet(M) = $\bigcup_{S\in M}$WordSet(S)

A sentence is said to be unary if it does not have the form X Y where X, Y ∈ SS. The set USS of unary sequences is defined as follows:
unary-sequence ::= X | X , Y where X is unary sentence, unary-sequence ::= Y

3.2. Substitutions

A function from unary sentences to sentences is called a substitution. Let sub be a substitution and V∈SS. A set of all unary sentences X such that sub(X) ≠ X is called the domain of the substitution sub and is denoted by dom of sub. An application sub(V) of the substitution sub to the sentence V is defined by the following:

- sub(V) = sub(X) if V has the form X where X∈WS;
- sub(V) = sub(X) sub(Y) if V has the form X Y where X,Y∈SS;
- sub(V) = L sub(X) R if V has the form L X R where (L,R)∈BPS, X∈SS;
- sub(V) = L R if V has the form L R where (L,R)∈BPS.

For a set X = {X_1,...,X_n}, sub(X) denotes

{sub(X_1),...,sub(X_n)}

For a function f from X to Y, sub(f) denotes a function from sub(X) to sub(Y) such that sub(f)(sub(x)) = sub(f(x)).

Let (X_1->Y_1,...,X_n->Y_n) denote the substitution sub with a domain {X_1,...,X_n} such that sub(X_i) = Y_i for each $1 \leq i \leq n$.

The length len of V of V is defined as follows:

- len of V = len of X + len of Y if V has the form X Y where X,Y∈SS;
- len of V = 1 otherwise.

3.3. States, configurations, natural state machines

A partial function from sentences to sentences is said to be a state if it has a finite domain.

Let StS denote a set of all states, S ∈ SS and state ∈ StS. We postulate that state(S) = () iff S ∉ dom of state. Let upd(state, X, Y) denote the state state′ such that

- state′(Z) = state(Z) for each Z∈(dom of state)\{X}.
- dom of state′ = (dom of state)∪{X} and state′(X) = Y if Y ≠ ().
- dom of state′ = (dom of state)\{X} if Y = ().

A set conf = {val, state, Rules, IWords, GWords} is said to be a configuration if val is a sentence, state is a state, Rules is a finite set of rules, IWords and GWords are finite sets of words such that IWords ∩ GWords = ∅.

Let CNF be a set of all configurations of the signature Sig. Let {m_1 := e_1,...,m_k := e_k}, where m_1,...,m_k ∈ conf, denotes a configuration obtained from conf by replacement of its elements m_1,...,m_k by the values e_1,...,e_k.

An NSM is a pair (Sig,II) where Sig is an NSM signature, II is a partial function of the scheme SS → USS → (CNF→2^{CNF}) called an instruction implementation. The elements of dom of II are called instruction names.

3.4. Matching rules and forms

Matching rules and matching forms are sentences defined in the following way:
matching-rule ::=
associate instruction (X) with Y where
X is sentence, matching-form ::= Y

matching-form ::= pattern (X)
[with parameters (Y)] [provided Z]
where X,Z are sentences, Y is unary sequence

A matching form X is said to be associated with a matching rule Y if Y has the form associate action (Z) with X where Z is an instruction.

3.5. Explanations of key notes

Let M be an NSM of the form (Sig,II) and conf be a configuration.

The signature Sig defines the alphabet of M. Sentences of the signature Sig are syntactic presentations of instructions of M that change configurations of M. The function II defines the semantics of the base instructions of M.

Matching rules allow us to define new instructions. A matching form specifies a class of sentences to which a matching rule associated with this form is applied.

The sentence val is a value of M, state is a state of M, Rules is a set of rules that can be applied in conf, IWords is a set of words appeared in the earlier executed instructions of M, and GWords is a set of words generated by the earlier executed instructions of M.

4. SEMANTICS OF A NATURAL STATE MACHINE

Semantics of natural state machines is defined by three transition relations over configurations: the global transition relation $\rightarrow_S^G$, local transition relation $\rightarrow_S^L$ and constrained transition relation $\rightarrow_S^{(m,n)}$, where m and n are nonnegative integers.

First we introduce some auxiliary notions. These notions are mutually recursive with the local transition relation.

Let a configuration conf have the form

{val, state, Rules, IWords, GWords}

sub be a substitution and S∈SS.

4.1. Notions

Let Y be a sentence of the form $Y_1, \ldots, Y_n$.

The sentence S is said to be an instance of the matching form pattern (X) with parameters (Y) provided Z in conf with respect to sub if dom of sub = $\{Y_1,\ldots,Y_n\}$, sub(X) = S and there exists val′ such that conf $\rightarrow^{L}_{sub(Z)}$ conf′ and val′ ≠ ().

Let m, n, k be nonnegative integers, R be a matching rule of the form associate instruction (U) . . . , and a sentence F be a matching form associated with R.

A pair (m, k) of nonnegative integers is said to be a replacement position in S in conf if there exist R ∈ Rules and sub such that S has the form X I Z, where I is an instance of F in conf with respect to sub, len of X = m-1 and len of I = k.

The pair (R,sub) is called by a replacement parameter of the position (m, k).

4.2. Transition relations

Let R be a matching rule of the form associate instruction (U) . . . with parameters (V).

The transition relations $\rightarrow^{G}_{S}$, $\rightarrow^{L}_{S}$ and $\rightarrow^{(m,n)}_{S}$ are defined as if-then rules in the following way:

1. If conf $\rightarrow^{G}_{S}$ conf′ and conf′ $\rightarrow^{G}_{S}$ conf″, then
   conf $\rightarrow^{G}_{S}$ conf″
2. If
   {sub(val), sub(state), sub(Rules),
   IWords ∪ WordSet(S), sub(GWords)} $\rightarrow^{L}_{S}$ conf′,
   where sub is a substitution that replaces variables of the set GWords ∩ WordSet(S) by new variables that do not belong to the set IWords ∪ WordSet(S) ∪ GWords, then conf $\rightarrow^{G}_{S}$ conf′
3. If conf $\rightarrow^{(1,1)}_{S}$ conf′, then conf $\rightarrow^{L}_{S}$ conf′

4. If
   - m+n-1 $\leq$ len of S
   - (m, k) is a replacement position in S in conf with the replacement parameter (R,sub), where k $\geq$ n
   - there is no replacement position (m, k′) in S in conf such that k′ > k
   - if U $\in$ dom of II then conf′ $\in$ II(U)(sub(V))(conf) else conf $\rightarrow^{L}_{sub(U)}$ conf′
   - conf $\rightarrow^{(m,\ len\ of\ sub(U)\ +\ 1)}_{X\ val'\ Y}$ conf″, where S = X I Y, len of X + 1 = m and len of I = k

   then conf $\rightarrow^{(m,n)}_{S}$ conf″
5. If conf $\rightarrow^{(m+n-1,\ 1)}_{S}$ conf′ where m+n-1 $\leq$ len of S and for any k $\geq$ n there is no replacement position (m,k) in S in conf, then conf $\rightarrow^{(m,n)}_{S}$ conf′
6. If m+n-1 > len of S, then conf $\rightarrow^{(m,n)}_{S}$ conf

4.3. Explanations of key notes

Let S be a sentence of the form $S_1 \ldots S_k$, where $S_1, \ldots, S_k \in$ USS.

The constrained transition relation $\rightarrow^{(m,n)}_{S}$ applies matching rules only to subsentences of S of the form $S_m \ldots S_l$, where l $\geq$ n. This relation defines a strategy of application of the local transition relation $\rightarrow^{L}_{S}$. This strategy is analogous to function calls in the functional programming languages.

The global transition relation $\rightarrow^{G}_{S}$ is different from the local transition relation $\rightarrow^{L}_{S}$ only in renaming the words of the sets IWords and GWords to avoid confusion.

5. THE USL LANGUAGE

The USL language is a language of description of NSMs. It consists of two sublanguages, a signature language and an instruction language.

All constructs of the USL language are sequences of symbols separated by one or more delimiters. The sets of symbols and delimiters are implementation-defined. Here we use blanks and new line symbols as delimiters.

5.1. Signature language

The signature language specifies the sets BPS and dom of INS of an NSM. A program of the signature language is defined in the following way:

signature-language-program ::=
[set of bracket pairs includes X]
[set of instructions includes Y]
where bracket-pair-sequence ::= X,
instruction-sequence ::= Y

bracket-pair-sequence ::=
left "(" L ")" right "(" R ")" |
left "(" L ")" right "(" R ")" X
where L,R are symbol sequences,
bracket-pair-sequence ::= X

instruction-sequence ::=
action "(" X ")" |
instruction "(" X ")" Y
where X is symbol sequence,
instruction-sequence ::= Y

Example. A signature language program
set of bracket pair includes
left({) right(}) left([) right(])
set of instructions includes
instruction(*) instruction(&) instruction(+)
specifies that BPS = {((,)), ({,}), ([,])} and dom of INS =
{*, &, +}.

Mechanisms of addition and execution of instructions defined by INS are implementation-defined.

The set WS is a set of all sequences of symbols that do not contain delimiters and brackets.

5.2. Instruction language

The instruction language defines instructions that are executed by NSMs specified by signature language programs. A program of the instruction language is defined in the following way:
instruction-language-program ::=
X | X Y where instruction ::= X,
instruction-language-program ::= Y

instruction ::= execute instruction X

where sentence ::= X

Let PS be a set of all instruction language programs, P, P′ ∈ PS and S ∈ SS. Semantics of the program P is defined by the transition relation $\rightarrow_P$ over CNF as follows:

- if P has the form S and conf $\rightarrow_S^G$ conf′, then conf $\rightarrow_P$ conf′
- if P has the form S P′ and conf $\rightarrow_S^G$ conf′ $\rightarrow_{P'}$ conf″, then conf $\rightarrow_P$ conf″

6. STANDARD INSTRUCTION LIBRARY

This section contains the Standard Instruction Library of the USL language. Library instructions are defined in the form:
matching rule

- ...
- ...
- ...

} formal definition

The matching rule specifies the instruction name and the matching form F associated with this rule, the formal definition defines the transition relation $\rightarrow_I^L$ for the instance I of the form F in the initial configuration conf with respect to the substitution sub. For the derived instructions, the formal definition is omitted.

6.1. Notions

The section contains notions used in the library instruction definitions.

Binders are sentences defined in the following way:
binder ::= binder of variables (X)
[of patterns (Y)] [provided Z]
where
unary-sentence-list ::= X,
sentence-list ::= Y, Z is sentence

unary-sentence-list ::= X | (X , Y)
where X is unary sentence,
unary-sentence-list ::= Y

sentence-list ::= X | (X , Y) where
X is sentence, sentence-list ::= Y

The binders

binder ::= binder of variables (X)
provided Z
and
binder ::= binder of variables (X) of
patterns (Y)
are short forms for
binder ::= binder of variables (X) of
patterns (X) provided Z
and
binder ::= binder of variables (X) of
patterns (Y) provided true,
respectively.

Let B denote a binder of the form
binder of variables (X) of patterns (Y)
provided Z
where X = $X_1,\ldots,X_n$ and Y = $Y_1,\ldots,Y_m$.

A substitution sub is said to be a syntactic candidate of B in conf if dom of sub = $\{X_1,\ldots,X_n\}$ and sub(Y_i) $\in$ dom of state for each $1 \leq i \leq m$.

A substitution sub is said to be a semantic candidate of B in conf if

- sub is a syntactic candidate of B in conf
- there exists val′ such that conf $\rightarrow^{L}_{sub(Z)}$ {val′, ...} and val′ does not have the form ().

A function logsub(B, conf) is said to be a characteristic function of B in conf if

- the domain dom of logsub(B,conf) of logsub(B,conf) is a set of syntactic candidates of B in conf.
- A range of logsub(B, conf) is a set of all subsets from the sentences true and false.
- if sub is a semantic candidate of B in conf, then true $\in$ logsub(B, conf)(sub).
- If conf $\rightarrow^{L}_{sub(Z)}$ {(), ...}, then false $\in$ logsub(B, conf) (sub) .

A sequence sub_1, ..., sub_k of pairwise different substitutions sub_i is called a series of candidates of B in conf if

- $sub_i \in$ dom of logsub(B,conf) for each $1 \leq i \leq k$
- if logsub(B,conf)(sub) = {true}, then sub = sub_i for some $1 \leq i \leq k$
- if logsub(B,conf)(sub) = {false}, then there is no $1 \leq i \leq k$ such that sub = sub_i.

Let serialize candidates of B in conf denote a set of all series of candidates of B in conf.

6.2. Execution instructions

associate instruction (*) with pattern (* X)
with parameters (X)

- conf $\rightarrow^{L}_{sub(*\ X)}$ {val := state(sub(X))}

associate instruction (&) with pattern (& X)
with parameters (X)

- conf $\rightarrow^{L}_{sub(\&\ X)}$ {val := sub(X)}

associate instruction (&&) with pattern (&& X)
with parameters (X)

- if X has the form (Y) where Y $\in$ SS, then conf $\rightarrow^{L}_{sub(\&\&\ X)}$ {val := sub(Y)}
- conf $\rightarrow^{L}_{sub(\&\&\ X)}$ {val := sub(X)} otherwise

6.3. Structural instructions

associate instruction (X) with pattern ((X))
with parameters (X)

associate instruction (;) with pattern (X ; Y)
with parameters (X , Y)

- if conf $\rightarrow^{L}_{sub(X)}$ conf$'$ $\rightarrow^{L}_{sub(Y)}$ conf$''$, then conf $\rightarrow^{L}_{sub(X\ ;\ Y)}$ conf$''$

associate instruction (,) with pattern (X , Y)
with parameters (X , Y)

- if conf $\rightarrow^{L}_{sub(X)}$ conf$'$ $\rightarrow^{L}_{sub(Y)}$ conf$''$, then conf $\rightarrow^{L}_{sub(X\ ,\ Y)}$ {val$''$:= val$'$, val$''$}.

6.4. Conditional instructions

associate instruction (if holds then else) with pattern (if X holds then Y else Z) with parameters (X , Y , Z)
provided not X is binder

- if conf $\rightarrow^{L}_{sub(X)}$ conf$'$, val$' \neq$ () and conf $\rightarrow^{L}_{sub(Y)}$ conf$''$, then conf $\rightarrow^{L}_{sub(if\ X\ then\ Y\ else\ Z)}$ conf$''$

- if conf $\rightarrow^{L}_{sub(X)}$ conf′, val′ = () and conf $\rightarrow^{L}_{sub(Z)}$ conf″, then conf $\rightarrow^{L}_{sub(if\ X\ then\ Y\ else\ Z)}$ conf″

associate instruction (if binder holds then else) with
pattern (if U holds then Y else Z) with parameters (U , Y , Z) provided U is binder

- if conf $\rightarrow^{L}_{sub'(sub(Y))}$ conf′ and
  true ∈ logsub(sub(U), conf) (sub′),
  then conf $\rightarrow^{L}_{sub(if\ U\ holds\ then\ Y\ else\ Z)}$ conf′
- if conf $\rightarrow^{L}_{sub(Z)}$ conf′ and false ∈ logsub(sub(U), conf) (sub′) for each sub′ ∈ dom of logsub(sub(U), conf), then
  conf $\rightarrow^{L}_{sub(if\ U\ holds\ then\ Y\ else\ Z)}$ conf′

associate instruction (if X then Y else ())
with pattern (if X then Y) with parameters (X , Y)
provided not Y contains else

6.5. Choice instructions

associate instruction (X) with pattern (one of two (X , Y)) with parameters (X , Y)

associate instruction (Y) with pattern (one of two (X , Y)) with parameters (X , Y)

associate instruction (execute for each) with pattern
(execute X for each U) with parameters (X , U)

- if conf $\rightarrow^{L}_{sub_1(sub(X));\ ...;\ sub_k(sub(X))}$ conf′ and $sub_1, \ldots,$ sub_k ∈ serialize candidates of sub(U) in conf, then
  conf $\rightarrow^{L}_{sub(execute\ X\ for\ each\ U)}$ conf′
- conf $\rightarrow^{L}_{sub(execute\ X\ for\ each\ U)}$ conf otherwise

associate instruction (execute for some) with pattern
(execute X for some U) with parameters (X , U)

- if conf $\rightarrow^{L}_{sub'(sub(X))}$ conf′ and
  true ∈ logsub(sub(U), conf) (sub'), then
  conf $\rightarrow^{L}_{sub(execute\ X\ for\ some\ U)}$ conf′
- if false ∈ logsub(sub(U), conf)(sub′) for each sub′ ∈ dom of logsub(sub(U), conf), then
  conf $\rightarrow^{L}_{sub(execute\ X\ for\ some\ U)}$ conf

6.6. Update rules

associate instruction (:=) with pattern (X := Y)
with parameters (X , Y)

- if conf $\rightarrow^{L}_{sub(X)}$ conf′ $\rightarrow^{L}_{sub(Y)}$ conf″, then
  conf $\rightarrow^{L}_{sub(X:=Y)}$ {state″ := upd(state″, val′, val″)}

associate instruction (new word) with pattern
(new word)

- if w ∈ WS \ (IWords ∪ GWords), then conf $\rightarrow^{L}_{sub(new\ word)}$ {val := w, GWords := GWords ∪ {w}}

associate instruction (add rule) with pattern(X)
with parameters (X) provided X is rule

- conf $\rightarrow^{L}_{sub(X)}$
  {val := sub(X), Rules := Rules ∪ {sub(X)}}

6.7. Abbreviation rules

associate instruction (let be in) with pattern (let X
be Y in Z) with parameters (X , Y , Z) provided X is
unary instruction

- if conf $\rightarrow^{L}_{sub(Y)}$ conf″ $\rightarrow^{L}_{sub'(sub(Z))}$ conf″ and
  sub′ = (sub(X)->val′), then
  conf $\rightarrow^{L}_{sub(let\ X\ be\ Y\ in\ Z)}$ conf″

associate instruction (execute X for some U) with
pattern (let U holds in X) with parameters (U , X)

6.8. Equality instructions

associate instruction (=) with pattern (X = Y)
with parameters (X , Y)

- if conf $\rightarrow^{L}_{sub(X)}$ conf′ $\rightarrow^{L}_{sub(Y)}$ conf″ and val′ = val″, then
  conf $\rightarrow^{L}_{sub(X\ =\ Y)}$ {val := true}
- conf $\rightarrow^{L}_{sub(X\ =\ Y)}$ {val := ()} otherwise

associate instruction (not X = Y) with pattern (X != Y) with parameters
(X , Y)

6.9. Propositional instructions

associate instruction (not) with pattern (not X)
with parameters (X)

- if conf $\rightarrow^{L}_{sub(X)}$ {(), ...}, then conf $\rightarrow^{L}_{sub(not\ X)}$ {val := true}
- conf $\rightarrow^{L}_{sub(not\ X)}$ {val := ()} otherwise

associate instruction (if X holds then (if Y holds then & true)) with pattern (X and Y) with parameters (X , Y)

associate instruction (if X holds then & true else
(if Y holds then & true)) with pattern (X or Y)
with parameters (X , Y)

associate instruction (if X and Y holds then & true
else (if (not X) and (not Y) holds then & true))
with pattern (X xor Y) with parameters (X , Y)

associate instruction ((not X) or Y) with pattern
(X implies Y) with parameters (X , Y)

associate instruction ((X implies Y) and (Y implies X)) with pattern (X iff Y) with parameters (X , Y)

6.10. Quantified instructions

associate instruction (for each holds) with pattern
(for each U holds) with parameters (U)

- if true $\in$ logsub(sub(U), conf)(sub$'$) for each sub$'$ $\in$ dom of logsub(sub(U), conf), then conf $\rightarrow^{L}_{sub(for\ each\ U\ holds)}$ {val := true}
- if false $\in$ logsub(sub(U), conf)(sub$'$) for some sub$'$ $\in$ dom of logsub(sub(U), conf), then conf $\rightarrow^{L}_{sub(for\ each\ U\ holds)}$ {val := ()}

associate instruction (for some holds) with pattern
(for some U holds) with parameters (U)

- if true $\in$ logsub(sub(U), conf)(sub$'$) for some sub$'$ $\in$ dom of logsub(sub(U), conf), then conf $\rightarrow^{L}_{sub(for\ some\ U\ holds)}$ {val := true}

- if false $\in$ logsub(sub(U), conf)(sub$'$) for each sub$'$ $\in$ dom of logsub(sub(U), conf), then conf $\rightarrow^{L}_{sub(for\ some\ U\ holds)}$ {val := ()}

associate instruction (for unique holds) with pattern (for unique U holds) with parameters (U)

- if serialize candidates of sub(U) in conf contains a singleton series, then conf $\rightarrow^{L}_{sub(for\ unique\ U\ holds)}$ {val := true}
- if serialize candidates of sub(U) in conf is an empty set or contains a series of some elements, then conf $\rightarrow^{L}_{sub(for\ unique\ U\ holds)}$ {val := ()}

6.11. Recognition instructions

associate instruction (is binder) with pattern (X is binder) with parameters (X)

- if sub(X) is a binder, then conf $\rightarrow^{L}_{sub(X\ is\ binder)}$ {val := true}
- conf $\rightarrow^{L}_{sub(X\ is\ binder)}$ {val := ()} otherwise

associate instruction (contains) with pattern (X contains Y) with parameters (X , Y)

- if sub(X) has the form U Y V where U, V $\in$ SS, then conf $\rightarrow^{L}_{sub(X\ contains\ Y)}$ {val := true}
- conf $\rightarrow^{L}_{sub(X\ contains\ Y)}$ {val := ()} otherwise

associate instruction (is rule) with pattern (X is rule) with parameters (X)

- if sub(X) is a matching rule, then conf $\rightarrow^{L}_{sub(X\ is\ rule)}$ {val := true}
- conf $\rightarrow^{L}_{sub(X\ is\ rule)}$ {val := ()} otherwise

associate instruction (is unary instruction) with pattern (X is unary sentence) with parameters (X)

- if sub(X) is a unary sentence, then conf $\rightarrow^{L}_{sub(X\ is\ unary\ sentence)}$ {val := true}
- conf $\rightarrow^{L}_{sub(X\ is\ unary\ sentence)}$ {val := ()} otherwise

associate instruction (length) with pattern (length of X) with parameters (X)

- if k is a number of sentences of the form I th element of sub(X), where I is an integer belonging to dom of state, then
$$\text{conf} \rightarrow^{L}_{sub(length\ of\ X)} \{\text{val} := \text{k}\}$$

6.12. Integer instructions

associate instruction (is integer) with pattern
(X is integer) with parameters (X)

- if conf $\rightarrow^{L}_{sub(X)}$ conf′ and val′ is an integer, then
$$\text{conf} \rightarrow^{L}_{sub(X\ is\ integer)} \{\text{val} := \text{true}\}$$
- conf $\rightarrow^{L}_{sub(X\ is\ integer)}$ {val := ()} otherwise

associate instruction (+) with pattern (X + Y) with
parameters (X , Y) provided (X is integer) and ((X ; Y) is integer)

- if conf $\rightarrow^{L}_{sub(X)}$ conf′, conf $\rightarrow^{L}_{sub(Y)}$ conf″ and sum is a sum of val′ and val″, then conf $\rightarrow^{L}_{sub(X\ +\ Y)}$ {val := sum}

associate instruction (<) with pattern (X < Y) with
parameters (X , Y) provided (X is integer) and ((X ; Y) is integer)

- if conf $\rightarrow^{L}_{sub(X)}$ conf′, conf $\rightarrow^{L}_{sub(Y)}$ conf″ and val′ is less than val″, then conf $\rightarrow^{L}_{sub(X\ <\ Y)}$ {val := true}
- conf $\rightarrow^{L}_{sub(X\ <\ Y)}$ {val := ()} otherwise

associate instruction (not ((X < Y) or (X = Y))) with
pattern (X > Y) with parameters (X , Y) provided
(X is integer) and ((X ; Y) is integer)

associate instruction (not X > Y) with pattern
(X <= Y) with parameters (X , Y) provided
(X is integer) and ((X ; Y) is integer)

associate instruction (not X < Y) with pattern
(X >= Y) with parameters (X , Y)
provided (X is integer) and ((X ; Y) is integer)

7. THE EXAMPLE: DESIGN OF C# SEMANTICS

Let us illustrate our approach to description of formal semantics of real-life programming languages by the example of design of a C# NSM. The

C# NSM is one of the main constituents of the three level approach to C# program verification [8].

The process of designing the C# NSM is divided into three stages. At the first stage, the set INS is built from atomic executable C# constructs (for instance, the names of the base operators with type qualifiers to avoid overloading). At the second stage, the algorithm of generation of initial configurations of the C# NSM, including information about C# program to be executed and execution environment, is developed. At the third stage, the C# NSM instruction that executes C# programs given by initial configurations is defined in the USL language.

7.1. Generation of initial C# NSM configurations

Translation of C# programs into initial C# NSM configurations is performed by recursive descendant along the program abstract syntax tree in one-to-one correspondence with the C# grammar [7]. For example, the C# program fragment public int sum(int x, int y) {return x+y;} is translated to the initial configuration conf. The state state of conf is defined in the following way:
state(E1 is method declaration) = true,
state(name of E1) = sum,
state(parameter list of E1) = E2,
state(1 th element of E2) = E3,
state(E3 is parameter) = true,
state(type of E3) = int,
state(name of E3) = x,
state(2 th element of E2) = E4,
state(type of E4) = int,
state(name of E4) = y,
state(return type of E1) = int,
state(modifier set of E1) = E5,
state(public belongs to E5) = true,
state(body of E1) = E6,
state(E6 is statement list) = true,
state(1 th element of E6) = E7,
state(E7 is return statement) = true,
state(expression of E7) = E8,
state(E8 is expression) = true,
state(operation of E8) = +,

state(operand list of E8) = E9,
state(1 th element of E9) = x,
state(2 th element of E9) = y.

The words E_i, coding nonterminals in the abstract syntax tree for the fragment, form the set GWords. The set IWords = WordSet(dom of state $\cup$ range of state) $\setminus$ GWords. The set Rules = $\emptyset$. The sentence val = ().

The translation preserves a logical structure of the source program that allows us to establish feedbacks with the source program code during execution of instructions of the C# NSM.

In addition to C# source program, the elements of an execution environment can also be added to initial configurations. Here we only extend dom of state by the sentence application argument list such that state(application argument list) is an argument list of the method Main.

7.2. C# NSM instruction

C# NSM instruction is defined, with almost one-to-one preservation of the terminology and the algorithm structure of C# language specification [7], by the program
execute instruction
application startup ;
execution of C# constructs

The execution environment calls a designated method which is referred to as the application's entry point with the help of the instruction
application startup
defined by the matching rule
associate instruction (
  if binder of variables (M) provided
    (M is method-declaration) and
    ((* name of M) = Main) and
    (((* parameter list of M) = ( )) or
    let L be (* parameter list of M) in
      (length of L) = 1 and
      ((type of * 1 th element of L) =
      (array of (string))))
  holds
  then
    let C be new word in
      (C := executing entry point

method) ;
((C is context) := true) ;
((executing M in context C) :=
active))
with pattern (application startup)
The rule creates the first active C# construct executing M in context C and specifies the context C in which the construct must be executed. The first active construct is an entry point method with the name Main and the given signature. It is possible for more than one of the classes or structs to contain a method called Main whose definition qualifies it to be used as an application entry point. The choice of an entry point is implementation dependent. The choice is done nondeterministically in this rule to cover all possible implementations.

The instruction execution of C# constructs executes active C# constructs in a nondeterminate manner according to the rule
associate instruction(
let A be executing E in context C
in
if binder of variables (E , C)
in patterns (A)
provided ((* A) = active) holds
then ((execute E in context C) ;
execution of C# constructs)
) with pattern (execution of C# constructs)

The instruction execute E in context C executes the C# construct E in the context C according to the rule
associate instruction(
if . . . then . . . ; . . . ; if . . . then . . .
) with pattern (execute E in context C)
with parameters (E , C)

In the branches if . . . then . . . , all possible C# constructs with all possible contexts are listed.

7.3. Examples of branches

The branches
if ((* C) = executing entry point method) and
((* body of E) = ())

```
then (application return value) := void

if ((* C) = executing entry point method) and
((* body of E) != ()) and
((* application argument list) = ())
then
   executing E in context C := () ;
   let C1 be new variable in
      C1 is context := true ;
      parent of C1 := C ;
      C1 := executing body of E ;
      (executing (* body of E) in
      context C1) := active

if ((* C) = executing entry point method) and
((* body of E) != ()) and
((* application argument list) != ())
then
   let B be (* body of E) in
   let A be (* application argument
         list) in
      extension of local variables set of
      block B to parameters (* parameter
      list of E) with values A ;
   executing E in context C := () ;
   let C1 be new variable in
      C1 is context := true ;
      parent of C1 := C ;
      C1 := executing body of E ;
      (executing B in context C1) :=
      active
```

define execution of the entry point method. The instruction extension of local variables set of block B to parameters (* parameter list of E) with values A needs, in its turn, a matching rule defining it and so on.

Thus, our approach to the description of formal programming language semantics is characterized by almost one-to-one translation of informal programming language specifications into NSMs and presentation of the se-

mantics in a form close to natural language texts. This speeds up the development of programming language semantics and makes it easy to understand.

8. CONCLUSION

The paper presents the following results of our research:

- a new class of abstract machines, natural state machines
- a language of natural state machines called USL and Standard Instruction Library of the language
- formal semantics of both USL and the library
- the example of C# semantics design, illustrating the USL-based approach to description of programming language semantics.

We are intended to apply the USL language to the description of semantics of .NET programming languages.

REFERENCES

1. Peter D. Mosses. *The Varieties of Programming Language Semantics and Their Uses.* Proceedings of Perspectives of System Informatics (PSI'01), Springer LNCS, P. 165–190, 2001.
2. AsmL Web Page. http://research.microsoft.com/fse/asml.
3. Philipp W. Kutter and Alfonso Pierantonio. *Montages: Specifications of Realistic Programming Languages.* Journal of Universal Computer Science, Vol. 3, N 5, P. 416–442, 1997.
4. P. G. Larsen, B. S. Hansen, H. Brunn N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. *Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language.* December 1996.
5. S. B. Lassen, P. D. Mosses, and D. A. Watt. *An introduction to AN-2: The proposed new version of action notation.* In Proceedings of the Third International Workshop on Action Semantics, N NS-00-6 in BRICS Notes Series, pages 19-36, 2000.
6. Anureev I.S. *The USL language. Syntax, Semantics and Pragmatics.* Joint Novisibirsk Computer Center and A.P. Ershov Institute of Informatics Systems Bulletin, Series Computer Science, N 20, 2004. (to appear).
7. ECMA-334. *C# Language Specification.* December 2001, http://www.ecma.ch.
8. V. A. Nepomniaschy, I. S. Anureev., I. V. Dubranovsky, A. V. Promsky. *A three level approach to C# program verification.* Joint Novisibirsk Computer Center and A.P. Ershov Institute of Informatics Systems Bulletin, Series Computer Science, N 20, 2004. (to appear).