Joint NCC & IIS Bull., Comp. Science, 20 (2004), 1–30

Unified semantic language: syntax, semantics, and pragmatics*

I. S. Anureev

Abstract. A new language of finite state machines called USL is proposed. The language is intended for rapid development of formal verification-oriented operational semantics of modern programming languages. Formal operational semantics of the USL language is defined. The USL-based approach to programming language semantics design is illustrated by the example of a definition of C# jump statements.

1. Introduction

The purpose of this paper is to present an FSM (finite state machines) based formalism for rapid development of formal verification-oriented operational semantics of modern programming languages.

At present, two approaches to FSM-based development of formal operational semantics of programming languages are used.

The first approach is in creation of a language specially destined for description of FSMs. A typical representative of this approach is AsmL (Abstract State Machine Language) [1, 2, 3] based on abstract state machines. AsmL is a general-purpose language for modeling the structure and behavior of discrete systems. The modeling approach behind AsmL is very powerful. AsmL can be used to faithfully capture the abstract structure and step-wise behavior of any discrete system, including very complex ones, such as integrated circuits, software components, and devices that combine both hardware and software. However, it does not take into account of peculiar features of FSMs that define operational semantics of programming languages. Montages [4, 5], a version of ASMs (abstract state machines) [6, 7, 8] specifically tailored for specifying the static and dynamic semantics of programming languages, are a forward step in this direction. Montages combine graphical and textual elements to yield specifications similar in structure, length, and complexity to those in common language manuals, but with a formal semantics. However, they are not verification-oriented and are hardly reducible to logics.

The second approach is in application of general-purpose theorem provers (HOL [9], PVS [10], LCF 2 [11, 12, 13], COQ [14]) to model FSMs, that

*This research was partially supported by Microsoft Research in 2002 and is partially supported by grant 04-01-00114a from RFBR.

define operational semantics of programming languages and prove program properties. This approach provides proving of verification conditions, but FSM models turn out to be unnatural and cumbersome and make verification conditions more complicated. What is more, these systems have a large number of heterogeneous syntactic constructs, semantics, which is difficult to understand and requires a profound knowledge in logic, a wide spectrum of methods and techniques of automatic proving, and numerous libraries of decision procedures for specific theories. Mastering these facilities takes much time.

We present a new language of FSMs called USL (Unified Semantic Language). USL is a simple compact language of FSMs. It combines the high productivity of rapid development of programming language semantics and the possibility of quite effective Refal-like implementation.

A compact formal description of USL, a small-step approach to definition of programming language semantics, and a natural translation of programming language constructs to abstract machine states make definitions of these constructs local and the semantics easy to understand.

Verification-oriented aspect is provided by the possibility to easily insert programming language semantics to first-order logics and by the use of any first-order prover (in particular, a specialized problem-oriented prover) for verification condition proving.

2. Syntax of USL

The USL language consists of two sublanguages, a signature language and an instruction language. This section contains the lexical and syntactic grammars of both sublanguages.

2.1. Extended Backus–Naur formalism

We use a variant of Backus–Naur formalism to describe the grammar of USL. 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 commas. For example, `"("`. Terminals and nonterminals are set in small letters. Variables are set in capital letters.

Constraints are given in any of the three forms

- **non-terminal ::= VARIABLE**,

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

Predicates are functions that return the value **true** or **false**.

Example 1. The grammar of arithmetical instructions without priority of 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. Lexical grammar and signature language

All constructs of the USL language are sequences of symbols separated by one or more delimiters. Sets of symbols and delimiters are implementation dependent. Here we use blanks and new line symbols as delimiters. The following nonterminals define the lexical grammar of USL:

```
symbol ::= X where X is symbol

symbol-sequence ::= X | X Y where
symbol ::= X, symbol-sequence ::= Y
```

The grammar of any constructs of the USL language is defined with respect to a signature. A pair **(Bracket pairs, Library actions)** is said to be a signature if **Bracket pairs** is a subset of the set of pairs of finite symbol sequences, **Library actions** is a subset of the set of finite symbol sequences. The elements of the sets **Bracket pairs** and **Library actions** are called bracket pairs and library actions, respectively. Bracket pairs are used to structure USL constructs. The elements **L** and **R** of a bracket pair **(L,R)** are called the left bracket and right bracket, respectively. Library actions are the names of atomic user-defined actions of the USL language. The mechanisms of addition and execution of library actions are implementation dependent. Further, a signature that includes only brackets **(** and **)** is admissible.

A signature language specifies a signature. The program in the signature language is defined by the nonterminal **signature-language-program** as follows:

```
signature-language-program ::=
[ set of bracket pairs includes X ]
```

```
[ set of library actions includes Y ]
where bracket-pair-sequence ::= X,
action-sequence ::= Y

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

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

Example 2. The signature language program

```
set of bracket pair includes
left(() right())
left({) right(})
left([) right(])
set of library actions includes
action(*) action(&) action(+)
```

specifies that the set **Bracket pair** of the signature includes the bracket pairs `((,))`, `({,})` and `([,])`, the set **Library actions** of the signature includes library actions `*`, `&` and `+`.

2.3. Instruction language

An instruction language defines instructions that point to library and user-defined actions.

A sequence of symbols is said to be an atomic instruction if it does not contain delimiters and brackets. The nonterminal **atomic-instruction** is defined as follows:

```
atomic-instruction ::= X where X is atomic instruction.
```

The nonterminal **instruction** is defined as follows:

```
instruction ::= (X | Y Z | L Y R | L R) where
simple-instruction ::= X, instruction ::= Y, Z,
(L, R) is bracket pair
```

Example 3. Let a signature include the bracket pairs `({, })` and `([, ])`. Then `[ ]`, `X + Y`, `if X > 0 then X else - X`, `{1 , 2 , 3}` are instructions.

Example 4. Let a signature include a set of paired tags of the HTML language as bracket pairs. Then **<html> Hello , World </html>** is an instruction.

Execution of an instruction consists in finding instances of invocation forms in it and applying invocation rules associated with these forms. Invocation forms and invocation rules are instructions of a special form. The nonterminal **invocation-form** is defined as follows:

```
invocation-form ::=
pattern (X) [with parameters (Y)] [ provided Z ]
where instruction ::= X,Y,Z
```

The nonterminal **invocation-rule** is defined as follows

```
invocation-rule ::= associate [ library ] action (X)
with Y where instruction ::= X, invocation-form ::= Y
```

Example 5. The instructions

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

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

are invocation rules that specify a nondeterministic choice of one of the two instructions.

3. Semantics of USL

USL is a language of FSMs. They serve to define executable semantics of programming languages. An FSM includes a set of variables and a set of instructions. Variables possess values and instructions point to actions that change the values of variables. A state of an FSM is defined by a function from variables to their values. The domain of a state is the set of variables whose values are defined in this state. The undefined value is denoted by `()`. A domain of any state of an FSM is finite. In addition to the state change, an action returns a value.

Variables of an FSM and their values, as well as instructions of the FSM, are presented by USL instructions. Execution of an instruction of the USL language consists in finding instances of invocation forms in it and applying invocation rules associated with these forms. We use the ASM (abstract state machine) approach [6, 7, 8] to define semantics of USL instructions.

3.1. Base notions

An instruction is said to be unary if it does not have the form `X Y`, where `X`, `Y` are instructions. A function from unary instructions to instructions is called a substitution. Let `sub` be a substitution and `V` be an instruction. A set of all unary instructions `X` such that `sub(X)` $\neq$ `X` is called a domain of the substitution `sub` and is denoted by `dom of sub`. An application `sub(V)` of the substitution `sub` to the instruction `V` is defined in the following way:

- `sub(V) = sub(X)` if `V` has the form `X`, where `X` is an atomic instruction and `X` belongs to the domain of `sub`;
- `sub(V) = V` if `V` has the form `X`, where `X` is an atomic instruction and `X` does not belong to the domain of `sub`;
- `sub(V) = sub(X) sub(Y)` if `V` has the form `X Y`, where `X`, `Y` are instructions;
- `sub(V) = L sub(X) R` if `V` has the form `L X R`, where `(L,R)` is a pair of brackets, `X` is an instruction;
- `sub(V) = L R` if `V` has the form `L R`, where `(L,R)` is a bracket pair.

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

The length `len of V` of the instruction `V` is defined as follows:

- `len of V = len of X + len of Y` if `V` has the form `X Y`, where `X`, `Y` are instructions;
- `len of V = 1` otherwise.

A partial function from instructions to instructions is said to be a state if it has a finite domain. Let `dom of state` denote a domain of a state `state`. An update `upd(state, X, Y)` of a state `state` is called a state `state'` such that

- `state'(Z) = state(Z)` for each instruction `Z` from `dom of state` except `X`.
- `dom of state'` = `dom of state` $\cup$ `{X}` and `state'(X) = Y` if `Y` does not have the form `()`.
- `dom of state'` = `dom of state` $\setminus$ `{X}` if `Y` has the form `()`.

A set `{val, state, is rule}` is said to be a configuration if `val` is a constant function returning an instruction, `state` is a state, `is rule` is a boolean function that characterizes a finite set of invocation rules.

An invocation form `X` is said to be associated with an invocation rule `Y` if `Y` has the form `associate action (Z) with X`, where `Z` is an instruction.

3.2. Semantics of instructions

Execution of an instruction of the USL language consists in finding instances of invocation forms in it and applying invocation rules associated with these forms.

Let a configuration **conf** have the form **{val, state, is rule}**.

Semantics **sem of I in conf** of an instruction **I** in the configuration **conf** is a set of configurations that we have after execution of the instruction **I**. The formal definition of the construct **sem of I in conf** is given below. Here we introduce this construct to define notions that are mutually recursive.

An instruction **I** is said to be an instance of the invocation form **pattern (X) with parameters (Y) provided Z** in the configuration **conf** with respect to a substitution **sub**, where **Y** has the form $\mathtt{Y}_1$**, ...,** $\mathtt{Y}_n$, if **{**$\mathtt{Y}_1$**,...,**$\mathtt{Y}_n$**}** is a domain of **sub**, **sub(X) = I** and a configuration **conf'={val', _, _}** belongs to **sem of sub(Z) in conf**, where **val'** is not equal to **()**.

Example 6. The instruction **one of two (use processor 1, use processor 2)** is an instance of the invocation form **pattern (one of two(X , Y)) with parameters (X , Y)** in any configuration with respect to the substitution **(X -> use processor 1, Y -> use processor 2)**.

Let **m**, **n**, **k** be nonnegative integers.

A pair **(m, k)** of nonnegative integers is said to be a replacement position in the instruction **V** in the configuration **conf** if an invocation rule **R** and a substitution **sub** exist such that

- **R is rule** is true.
- **V** has the form **X I Z**, where **I** is an instance of the invocation form **F** associated with the rule **R**, in a configuration **conf** with respect to the substitution **sub**, **len of X = m-1** and **len of I = k**.

The instruction **sub(A)**, where **R** has the form **... action (A) ...**, is called a replacement parameter of the position **(m, k)**.

Example 7. Let the function **is rule** from the configuration **conf** be true on the rule

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

Then the pair **(1, 1)** is a replacement position in the instruction **one of two (use processor 1 , use processor 2)** in the configuration **conf**, and **use processor 1** is a replacement parameter of the position.

A choice of instances of invocation forms satisfies a certain strategy. This strategy reduces the definition of semantics **sem of V in conf** of an instruction **V** in a configuration **conf** to the definition of semantics **sem of V in conf in context C** of an instruction **V** in a configuration **conf** in a context **C**, where **C** is defined as a pair **(m,n)** of nonnegative integers and is called an execution context:

1. **sem of V in conf = sem of V in conf in context (1,1).**
2. A configuration **conf″** belongs to **sem of V in conf in context (m,n)** if
   - **m+n-1** $\leq$ **len(V)**.
   - **(m, k)** is a replacement position in the instruction **V** in the configuration **conf**, where **k** $>$ **n**.
   - There is no replacement position **(m, k′)** in the instruction **V** in the configuration **conf** such that **k′** $>$ **k**.
   - An instruction **U** is a replacement parameter of the position **(m, k)**.
   - A configuration **conf′ = {val′, _, _}** belongs to **sem of U in conf**.
   - A configuration **conf″** belongs to **sem of X val′ Y in conf in context (m, len of U + 1)**, where **V = X I Y**, **len of X + 1 = m** and **len of I = k**.
3. **sem of V in conf in context (m,n) = sem of V in conf in context (m+n-1, 1)** if **m+n-1** $\leq$ **len(V)** and for any **k** $\geq$ **n** there is no replacement position **(m,k)** in the instruction **V** in the configuration **conf**.
4. **sem of V in conf in context (m,n) = {conf}** if **m+n-1 > len(V)**.

Example 8. For inversion rules **associate action (0) with pattern (1)** and **associate action (1) with pattern (0)** the instruction **0011** is executed in the state **conf = {val, state, is rule}**, where the function **is rule** is true on these rules, as follows:

```
sem of 0011 in conf = (by 1)
sem of 0011 in conf in context (1, 1) = (by 2)
sem of 1011 in conf in context (1, 2) = (by 3)
sem of 1011 in conf in context (2, 1) = (by 2)
sem of 1111 in conf in context (2, 2) = (by 3)
sem of 1111 in conf in context (3, 1) = (by 2)
sem of 1101 in conf in context (3, 2) = (by 3)
```

```
sem of 1101 in conf in context (4, 1) = (by 2)
sem of 1100 in conf in context (4, 2) = (by 4)
{1100, state, is rule}.
```

4. Example: semantics of C# jump statements

Let us illustrate our approach by defining the semantics of C# jump statements in terms of a USL-based C# abstract machine. The well-known difficulty of execution of C# jump statements [15] is a presence of intervening try statements. In the absence of such try statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening try statements, execution is more complex. If the jump statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of the finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.

To apply instructions of the C# abstract machine, it is necessary to translate the source C# program to the initial state of the machine. Here we omit the full-length translation algorithm **TR**, introducing its separate cases as required.

To define the C# abstract machine, we use Standard Instruction Library [16] of the USL language.

The C# abstract machine deals with the variables of a special form, having the constant value **active**. They are called activities.

There are four types of activities. The activity **execute S in context C** means that the C# construct **S** is ready to be executed in the execution context **C**. The activity **end S in context C** means that the C# construct **S** is ready to be normally terminated in the execution context **C**. The activity **end S in context C with result R** means that the C# construct **S** is ready to be normally terminated in the execution context **C** with the result **R**. The activity **jump S in context C** means that unconditional transfer of control (in particular, propagation of an exception), generated by a jump statement, has achieved the C# construct **S** in the execution context **C**.

The step of the C# abstract machine consists in nondeterministic choice of an activity (if any) and its execution. The machine terminates when and if there is no activity in the current state. In particular, this approach allows us to define interleaving semantics of threads. The instruction defining execution of an activity consists of a sequence of branches in the form of the library conditional instruction **if X then Y** with common semantics. The condition **X** specifies a restriction on the C# construct **S** and the execution context **C** of the activity. The instruction **Y** specifies the necessary actions.

In addition to jump statements, jump activity of C# statements that are targets of the jump statements is specified.

4.1. Throw statement

The case of a throw statement in the translation algorithm **TR** has the form:

```
TR(throw X:expression ;) =
A is throw statement ← true ; argument of A ← TR(X) ;
returns A
```

We use the following notations in the schemes of that form. Let **X:T** stand for the fact that the string **X** matches the nonterminal or literal class **T**; **X ← V ;** stand for assigning the value **V** to the variable **X**; the letters **A**, **B**, **C** denote new atomic instructions coding nonterminal vertices in the syntax tree of the source program; **returns X** denote that the recursive function **TR** returns the value **X**.

The branches

```
if (* S is throw statement) and (* argument of S) = ()
  holds
then
  (execute S in context C := ()) ;
  jump (parental statement of S) in context C := active

if (* S is throw statement) and ((* argument of S) != ())
  and C is jump context holds
then
  (execute S in context C := ()) ;
  (execute (* argument of S) in context
    (* parental context of C) := active) ;
  send ((* C) , C) to garbage

if (* S is throw statement) and ((* argument of S) != ())
  and not C is jump context holds
then
  (execute S in context C := ()) ;
  (execute (* argument of S) in context C := active)
```

for the activity **execute S in context C** start to execute the throw statement **S** in the context **C**.

The library instruction *** X** returns the value of the instruction **X** in the current state. The instruction **(* argument of S) = ()** and **(* argument of S) != ()** stand for the absence and presence of the argument of **S**, respectively. The library instructions **X := Y**, **X = Y** and **X != Y** are the assignment instruction, equality instruction and inequality instruction with

a common semantics, respectively. The library instructions **and**, **or** and **not** are common logical connectives. The instruction **execute S in context C := ()** eliminates the activity **execute S in context C**. An instruction of the form **X := active** adds the new activity **X**.

The throw statement with no argument, presented by the first branch, can be used only in a catch block, in which case this statement re-throws the exception that is currently being handled by this catch block. This exception is stored in the context **C**. The instruction **parental statement of S** returns the immediately enclosing statement of **S** called a parental statement. If there is no parental statement, it returns **no statement**.

The rest of the branches makes the argument of **S** ready to be executed. If **C** is a context generated by a before-executed jump statement, called a jump context, the second branch eliminates it, transiting to the context *** parental context of C** which preceded the execution of the jump statement. This context is called a parental context of **C**. The instruction **send ((* C) , C) to garbage** makes the instructions **(* C)** and **C** accessible for garbage collection. Semantics of this instruction is implementation-dependent.

The branch

```
let S1 be parental context of C in
  if (* S1 is throw statement) and S = (* argument of S1)
    holds
  then
    (end S in context C with result R := ()) ;
    let S2 be new word in
      (S2 is throw instance := true) ;
      (value of S2 := R) ;
      let C1 be new word in
        (parental context of C1 := C) ;
        (C1 := S2) ;
        jump (parental statement of S1) in context C1 :=
          active
```

for the activity **end S in context C with result R** completes the execution of the throw statement **S1** when and if the execution of its argument **S** has normally terminated with a result **R**.

The instruction **let S2 ...R)** stands for generation of a new propagated exception **S2**, called throw instance, with the value **R** by the throw statement **S1**. The instruction **let C1 ...S2)** stands for creation of a new jump context **C1**, which stores **S2** and has the context **C** as a parent. The library instruction **let X be Y in Z** introduces the abbreviation **X** for the value of the instruction **Y** into the instruction **Z**. The library instruction **new word** returns a new atomic instruction.

The branch

```
if (* S is throw statement) then
  (jump S in context C := ()) ;
  jump (parental statement of S) in context C := active
```

for the activity **jump S in context C** propagates the exception generated by abnormal termination of the argument of the throw statement **S** to the parental statement of **S**.

4.2. Break statement

The case of a break statement in the translation algorithm **TR** has the form:

```
TR(break;) = A is break statement ← true ; returns A
```

The branch

```
if (* S is break statement) holds
then
  (execute S in context C := ()) ;
  let C1 be new word in
    (parental context of C1 := C) ;
    C1 := S ;
    jump (parental statement of S) in context C1 := active
```

for the activity **execute S in context C** completes the execution of the break statement **S** and transfers control to the parental statement of **S**. The instruction **C1** stands for a new jump context generated by the break statement **S**.

4.3. Continue statement

The case of a continue statement in the translation algorithm **TR** has the form:

```
TR(continue;) = A is continue statement ← true ;
returns A
```

The branch

```
if (* S is continue statement) holds
then
  (execute S in context C := ()) ;
  let C1 be new word in
    (parental context of C1 := C) ;
```

```
    C1 := S ;
    jump (parental statement of S) in context C1 := active
```

for the activity **execute S in context C** completes the execution of the continue statement **S** and transfers control to the parental statement of **S**.

4.4. Goto statement

The case of a goto statement in the translation algorithm **TR** has the form:

```
TR(goto X:identifier ;) =
A is goto statement ← true ; label of A ← TR(X) ;
returns A

TR(goto case X:constant-expression ;) =
A is goto statement ← true ;
case expression of A ← TR(X) ;
returns A

TR(goto default ;) =
A is goto statement ← true ; default of A ← true ;
returns A
```

The branch

```
if (* S is goto statement) and
  (((* case expression of S) = ()) or
    (((* case expression of S) != ()) and
      (* case value of S) != ())) holds
then
  (execute S in context C) := () ;
  let C1 be new word in
    (parental context of C1 := C) ;
    C1 := S ;
    jump (parental statement of S) in context C1 := active
```

for the activity **execute S in context C** completes the execution of the goto statement **S** and transfers control to the parental statement of **S**.

The branch

```
if (* S is goto statement) and
  ((* case expression of S) != ()) and
  (* case value of S) = () holds
then
  (execute S in context C := ()) ;
  execute (* case expression of S) in context C := active
```

for the activity **execute S in context C** starts the execution of the goto case statement **S** and makes the case expression of the statement ready to be executed.

The branch

```
let S1 be (parental statement of S) in
  if (S1 is goto statement) and
    (S = * case expression of S1) holds
  then
    (end S in context C with result R := ()) ;
    (* case value of S1 := R) ;
    execute S1 in context C := active
```

for the activity **end S in context C with result R** stores the value of the constant case expression **S** of the goto case statement **S1** in the variable **case value of S1** and makes the statement **S1** ready to be executed with already computed case expression.

4.5. Return statement

The case of a return statement in the translation algorithm **TR** has the form:

```
TR(return X:expression ;) =
A is return statement ← true ; expression of A ← TR(X) ;
returns A
```

The branch

```
if (* S is return statement) and (* expression of S) != ()
  holds
then
  (execute S in context C := ()) ;
  execute (* expression of S) in context C := ()
```

for the activity **execute S in context C** completes the execution of the return statement **S** and transfers control to the parental statement of **S**.

```
if (* S is return statement) and (* expression of S) = ()
  holds
then
  (execute S in context C := ()) ;
  let C1 be new word in
    (parental context of C1 := C) ;
    C1 := S ;
    jump (parental statement of S) in context C1 := active
```

for the activity **execute S in context C** starts the execution of the return statement **S** and makes the expression of the statement ready to be executed.

The branch

```
let S1 be parent of C in
  if (* S1 is return statement) and S = (* expression of S1)
  holds
  then
    (end S in context C with result R := ()) ;
    let S2 be new word in
      (S2 is return instance := true) ;
      (value of S2 := R) ;
      let C1 be new word in
        (parental context of C1 := C) ;
        C1 := S2 ;
        jump (parental of S1) in context C1 := active
```

for the activity **end S in context C with result R** completes the execution of the return statement **S1** and transfers control to the parental statement of **S1** with a new jump execution context **C1** that stores the computed return value **R** of the expression **S** of the return statement **S1**.

4.6. Try statement

The case of a try statement in the translation algorithm **TR** has the form:

```
TR(try X:block Y:catch-clause-list Z:finally-clause) =
A is try statement ← true ;
block of A ← TR(X) ;
catch clause list of A ← TR(Y) ;
finally clause of A ← TR(Z) ;
returns A
```

The branch

```
if * S is try statement holds then
  (execute S in context C := ()) ;
  execute (* block of S) in context C := active
```

for the activity **execute S in context C** makes the block of the try statement **S** ready to be executed.

The branches

```
let S1 be parental statement of S in
  if (* S1 is try statement) and (S = (* block of S1)) and
    (* finally clause of S1) != () holds
```

```
  then
    (end S in context C := ()) ;
    execute (* finally clause of S1) in context C := active

let S1 be parental statement of S in
  if (* S1 is try statement) and (S = (* block of S1)) and
    (* finally clause of S1) = () holds
  then
    (end S in context C := ()) ;
    end S1 in context C := active
```

for the activity **end S in context C** continue the execution of the try statement **S1** when and if the execution of the block **S** of the statement **S1** has normally terminated. The first branch holds if the try statement **S1** has the finally clause. It makes the clause ready to be executed. The second branch holds if there is no finally clause in the try statement **S1**. It terminates the execution of **S1**.

The branches

```
let S1 be parental statement of S in
  if (* S1 is try statement) and
    (S = (* finally clause of S1))
    and not C is jump context holds
  then
    (end S in context C := ()) ;
    end S1 in context C := active

let S1 be parental statement of S in
  if (* S1 is try statement) and
    (S = (* finally clause of S1))
    and C is jump context holds
  then
    (end S in context C := ()) ;
    jump (parental statement of S1) in context C := active
```

for the activity **end S in context C** complete the execution of the try statement **S1** when and if the finally clause **S** of the statement has normally terminated.

The branches

```
let S1 be parental statement of S in
  if (* S1 is try statement) and
    S = (* catch clause list of S1) and
    (* finally clause of S1) != () holds
  then
    (end S in context C in context R := ()) ;
```

```
    execute (* finally clause of S1) in context C := active

let S1 be parental statement of S in
  if (* S1 is try statement) and
    S = (* catch clause list of S1) and
    (* finally clause of S1) = () and
    R = exception has caught holds
  then
    (end S in context C with result R := ()) ;
    end S1 in context C := active

let S1 be parental statement of S in
  if (* S1 is try statement) and
    S = (* catch clause list of S1) and
    (* finally clause of S1) = () and
    R = exception has not caught holds
  then
    (end S in context C with result R := ()) ;
    jump (parental statement of S1) in context C := active
```

for the activity `end S in context C with result R` continue the execution of the try statement `S1` when and if the execution of the catch clause list `S` of the statement `S1` has normally terminated. The first branch holds if the try statement `S1` has the finally clause. It makes the clause ready to be executed. The rest of branches hold if there is no finally clause in the try statement `S1`. The second branch holds if the exception has caught by one of the catch clauses of the list `S`. It terminates the execution of `S1`. The third branch holds if no catch clause of the list `S` has caught the exception. It propagates the exception to the parental statement of the try statement `S`.

The branches

```
if (* S is try statement) and (C is jump context) and
  (not (* C) is throw instance) and
  (* finally clause of S) != () holds
then
  (jump S in context C := ()) ;
  execute (* finally clause of S) in context C := active

if (* S is try statement) and (C is jump context) and
  (not (* C) is throw instance) and
  (* finally clause of S) = () holds
then
  (jump S in context C := ()) ;
  jump (parental statement of S) in context C := active

if (* S is try statement) and (* (* C) is throw instance)
```

```
  and ((* catch clause list of S) = ()) and
  (* finally clause of S) = () holds
then
  (jump S in context C := ()) ;
  jump (parental statement of S) in context C := active

if (* S is try statement) and (* (* C) is throw instance)
  and ((* catch clause list of S) = ()) and
  (* finally clause of S) != () holds
then
  (jump S in context C := ()) ;
  execute (* finally clause of S) in context C := active

if (* S is try statement) and (* (* C) is throw instance)
  and (* catch clause list of S) != () holds
then
  (jump S in context C := ()) ;
  execute (* catch clause list of S) in context C := active
```

for the activity `jump S in context C` continue to execute the try statement `S` when and if the block of the statement `S` has abnormally terminated.

The first and second branches hold if control is transferred by any jump statement except throw statement. They differ on the presence or absence of the finally clause in the statement `S`. The rest of branches hold if an exception is propagated. They differ on the presence or absence of the finally clause and catch clause list in the statement `S`.

4.7. Catch clause list

The case of a catch clause list in the translation algorithm `TR` has the form:

```
TR(X1:catch-clause ...XN:catch-clause) =
A is catch clause list ← true ;
1 th of A ← TR(X1) ;
...;
N th of A ← TR(XN) ;
returns A
```

The branch

```
if * S is catch clause list holds then
  (execute S in context C := ()) ;
  execute (* 1 th of S) in context C := active
```

for the activity `execute S in context C` makes the first catch clause of the catch clause list `S` ready to be executed.

The branches

```
let S1 be parental statement of S in
  if (* S1 is catch clause list) and
    (R = exception has caught) holds
  then
    (end S in context C with result R := ()) ;
    end S1 in context C with result R := active
```

```
let S1 be parental statement of S in
  if (* S1 is catch clause list) and
    (R = exception has not caught) and
    (* ((number of S in S1) + 1) th of S1) != () holds
  then
    (end S in context C with result R := ()) ;
    execute (* (I+1) th of S1) in context C := active
```

```
let S1 be parental statement of S in
  if (* S1 is catch clause list) and
    (R = exception has not caught) and
    (* ((number of S in S1) + 1) th of S1) = () holds
  then
    (end S in context C with result R := ()) ;
    end S in context C with result R := active
```

for the activity **end S in context C with result R** transfer activity from the last executed catch clause **S** of the catch clause list **S1** to the next catch clause of the list. The instruction **number of S in S1** stands for the number of **S** in **S1**. The branches differ on the presence or the absence of the next catch clause and by the fact whether the exception has caught or not in **S**.

The branch

```
if (* S is catch clause list) holds then
  (jump S in context C := ()) ;
  jump (parental statement of (parental statement of S))
    in context C := active
```

for the activity **jump S in context C** completes the execution of the try block and propagates an exception to the parental statement of the try block when and if the catch clause list **S** of this try statement has abnormally terminated, having thrown the exception.

4.8. Catch clause

The case of a catch clause in the translation algorithm **TR** has the form:

```
TR(catch (X:type Y:identifier) Z:block) =
A is catch clause ← true ;
type of A ← TR(X) ;
parameter of A ← TR(Y) ;
block of A ← TR(Z) ;
returns A
```

The branches

```
let S1 be * C in
  if (* S is catch clause) and
    ((type of (* value of S1)) is subtype of (* type of S))
    and (* parameter of S) != () holds
  then
    (execute S in context C) := () ;
    (let V be new word in
      (V is local variable := true) ;
      (type of V := * type of S) ;
      (name of V := * parameter of S) ;
      (scope of V := * block of S) ;
      (invocation instance of V :=
        invocation instance of C) ;
      (value of V := * value of S1)) ;
    execute (* block of S) in context C := active

let S1 be * C in
  if (* S is catch clause) and
    ((type of (* value of S1)) is subtype of (* type of S))
    and (* parameter of S) = () holds
  then
    (execute S in context C := ()) ;
    execute (* block of S) in context C := active

let S1 be * C in
  if (* S is catch clause) and ((* type of S) != ()) and
    not (type of (* value of S1)) is subtype of (* type of S)
    holds
  then
    (execute S in context C := ()) ;
    end S in context C
      with result exception has not caught := active

if (* S is catch clause) and (* type of S) = () holds
```

```
then
  (execute S in context C := ()) ;
  execute (* block of S) in context C := active
```

for the activity `execute S in context C` start to execute the catch clause `S`, transferring activity to the block of `S`.

The instruction `X is subtype of Y` stands for the fact that the type `Y` coincides with the type `X` or `Y` is a base type of `X`. The instruction `(* type of S) != ()` stands for the fact that `S` is a specific catch clause.

The first and second branches hold if the specific catch clause `S` is a first matching catch clause. They differ on the presence or absence of a parameter of `S`. The first branch declares a local exception variable `V` with the name `* parameter of S`. The exception object `* value of S1` is assigned to this variable. The type, scope and invocation instance of this variable are defined. The invocation instance specifies the current function member invocation.

The third branch holds if the specific catch clause `S` is not a matching catch clause. It completes the execution of the catch clause `S`. The forth branch deals with the general catch clause `S`.

The branches

```
let S1 be parental statement of S in
  if (* S1 is catch clause) and (S = (* block of S1)) and
    (C is jump context) holds
  then
    (end S in context C := ()) ;
    (end S1 in context (* parental context of C)
      with result exception is caught := active) ;
    send ((* C) , C) to garbage

let S1 be parental statement of S in
  if (* S1 is catch clause) and (S = (* block of S1)) and
    not C is jump context holds
  then
    (end S in context C := ()) ;
    end S1 in context C
      with result exception is caught := active
```

for the activity `end S in context C` completes the execution of the catch clause `S1` when and if the block `S` of the catch clause `S1` has normally terminated. Moreover, the first branch eliminates the catched exception if this exception has not yet been eliminated.

The branch

```
let S1 be parental statement of S in
```

```
if (* S1 is catch clause list)
then
  (jump S in context C := ()) ;
  jump S1 in context C := active
```

for the activity `jump S in context C` completes the execution of the catch clause `S` and propagates an exception to the catch clause list `S1` when and if the exception has been thrown during the execution of the block of `S`.

4.9. Finally clause

The case of a finally clause in the translation algorithm `TR` has the form:

```
TR(finally X:block) =
A is finally clause ← true ;
block of A ← TR(X) ;
returns A
```

The branch

```
if * S is finally clause holds then
  (execute S in context C := ()) ;
  execute (* block of S) in context C := active
```

for the activity `execute S in context C` starts to execute the finally clause `S`, transferring activity to the block of the clause `S`.

The branch

```
let S1 be parental statement of S in
  if (* S1 is finally clause) and S = (* block of S1)
    holds
  then
    (end S in context C := ()) ;
    end S1 in context C := active
```

for the activity `end S in context C` completes the execution of the finally clause `S1` when and if the block `S` of this clause has normally terminated.

The branch

```
if (* S is finally clause) holds
then
  (jump S in context C := ()) ;
  jump (parental statement of (parental statement of S))
    in context C := active
```

for the activity **jump S in context C** completes the execution of the try statement with the finally clause **S** and propagates an exception to the parental statement of the try statement when and if the block **S** of this clause has abnormally terminated, having thrown the exception.

4.10. No statement

The branches

```
let S1 be * C , C1 be parental context of C ,
  S2 be * C1 in
  if (* S1 is throw instance) and
    (* S2 is invocation instance) holds
  then
    (jump no statement in context C := ()) ;
    (end (* generating expression of S2)
      in context (* parental context of C1)
      with result (* value of S1) := active) ;
    send (S1 , C , S2 , C1) to garbage

let C1 be parental context of C , S1 be * C ,
  S2 be * C1 in
  if (* S1 is throw instance) and
    (* S2 is thread instance) holds
  then
    (jump no statement in context C := ()) ;
    send (S1 , C , S2 , C1) to garbage

let S1 be * C , C1 be parental context of C ,
  S2 be * C1 in
  if (* S1 is return statement) and
    (* S2 is invocation instance) holds
  then
    (jump no statement in context C := ()) ;
    (end (* generating expression of S2)
      in context (* parental context of C1) := active) ;
    send (S1 , C , S2 , C1) to garbage

let S1 be * C , C1 be parental context of C ,
  S2 be * C1 in
  if (* S1 is return instance) and
    (* S2 is invocation instance) holds
  then
    (jump no statement in context C := ()) ;
    (end (* generating expression of S2)
      in context (* parental context of C1)
      with result (convert implicitly (* value of S1) to
```

```
      * return type of S2) := active) ;
    send (S1 , C , S2 , C1) to garbage
```

for the activity **jump no statement in context C** hold if for some statement **S** the instruction **parental statement of S** returns the value **no statement**, i. e. there is no enclosing statement of **S**.

The first branch holds if an exception handler was not located in the current function member invocation. In this case, the function member invocation is terminated and exception propagation is then performed for the caller of the function member with a throw point corresponding to the statement from which the function member was invoked.

The second branch holds if the exception processing terminates all function member invocations in the current thread, indicating that the thread has no handler for the exception. In this case, the thread is terminated itself.

The other branches hold if the context **C** is a jump context that stores the return statement **S1** with no returning value (the third branch) or the return instance **S1** with returning value *** value of S1** (the forth branch). The returning value (if any) is converted to the return type of the containing function member by an implicit conversion. The result of the conversion becomes the value returned to the caller.

4.11. Iteration statements

Let us consider only a jump activity **jump S in context C** of a while statement. The jump activity of the other iteration statements is defined in a similar way.

The case of the while statement in the translation algorithm **TR** has the form:

```
TR(while (X:boolean-expression) Y:embedded-statement) =
A is while statement ← true ;
condition of A ← TR(X) ;
statement of A ← TR(Y) ;
returns A
```

The branch

```
if (* (* C) is break statement) and
  * S is while statement holds
then
  (jump S in context C := ()) ;
  (end S in context (* parental context of C) := active) ;
  send ((* C) , C) to garbage
```

completes the execution of the while statement **S** when and if the jump context **C** stores the before-executed break statement. The jump context **C** is eliminated and its parental context is restored.

The branch

```
if (* (* C) is continue statement) and
  * S is iteration statement holds
then
  (jump S in context C := ()) ;
  (end (* statement of S)
    in context (* parental context of C) := active) ;
  send ((* C) , C) to garbage
```

transfers control to the end point of the embedded statement of the while statement **S**, starting the new iteration of the statement **S**, when and if the jump context **C** stores the before-executed continue statement. The jump context **C** is eliminated and its parental context is restored.

4.12. Switch statement

The case of a switch statement in the translation algorithm **TR** has the form:

```
TR(switch (X:expression) Y:switch-block) =
A is switch statement ← true ;
switch expression of A ← TR(X) ;
switch block of A ← TR(Y) ;
returns A

TR(X1-switch-section ...
  XN:switch-section):switch-block =
A is switch block ← true ;
switch section set of A ← B ;
TR(X1) in B ← true ;
...;
TR(XN) in B ← true ;
returns A

TR(X1:switch-label ... XN:switch-label
  Y:statement-list):switch-section =
A is switch statement ← true ;
label set of A ← B ;
TR(X1) in B ← true ;
...;
TR(XN) in B ← true ;
statement list of A ← TR(Y) ;
returns A
```

```
TR(case X:constant-expression):switch-label = TR(X)

TR(default):switch-label = default
```

The branch

```
let S1 be * C in
  if (* S is switch block) and (* S1 is goto statement)
    and (* case value of S1) != () holds
  then
    (jump S in context C := ()) ;
    send (S1) to garbage ;
    (C := * case value of S1) ;
    execute S in context C := active
```

transfers control from the goto case statement `S1`, stored in the jump context `C`, to the switch block `S` with the case label `(* case value of S1)`.

The branch

```
let S1 be * C , S2 be * switch section set of S in
  if binder of variables (S3) in patterns ((S3 in S2))
    provided (* S is switch block) and
    (* S1 is goto statement) and
    ((* default of S1) != ()) and (* S3 in S2) and
    * default in (* switch label set of S3) holds
  then
    (jump S in context C := ()) ;
    (execute (* statement list of S3)
      in context (* parental context of C) := active) ;
    send (S1 , C) to garbage
```

transfers control from the goto default statement `S1`, stored in the jump context `C`, to the statement list of the switch section `S3` from the switch section set `S2` of the switch block `S` such that `C` includes the label `default`. The library instruction `X in Y` stands for the fact that the instruction `X` belongs to the set `Y`. The library instruction

```
if binder of variables (X1 , ..., XN)
in patterns ((Y1) , ..., (YM)) provided Z holds then U
```

stands for the pattern matching. The statement `U` is executed when and if there exist values `V1`, ..., `VN` of the bound variables `X1`, ..., `XN` such that instances of the patterns `Y1`, ..., `YM` w.r.t. the substitution (`X1` $\rightarrow$ `V1`, ..., `XN` $\rightarrow$ `VN`) belong to the domain of the current state. The values of the bound variables is substituted to the statement `U` before its execution.

The branch

```
if (* (* C) is break statement) and
  * S is switch statement holds
then
  (jump S in context C := ()) ;
  (end S in context (* parental context of C) := active) ;
  send ((* C) , C) to garbage
```

completes the execution of the switch statement `S` when and if the jump context `C` stores the before-executed break statement. The jump context `C` is eliminated and its parental context is restored.

4.13. Block statement

The branch

```
let S1 be * C in
  if (* S is block statement) and (C is jump context) and
    not ((* S1 is goto statement) and
      (find labelled statement
        with label (* label of S1) in S) != ()) holds
  then
    (jump S in context C := ()) ;
    end S in context (* parental context of C) := active
    (delete locals for S in context C) ;
    send ((* C) , C) to garbage
```

completes the execution of the block statement `S` when and if `C` is a jump context that stores a jump statement except for the special form of the goto statement. The jump context `C` is eliminated and its parental context is restored.

The instruction **`find labelled statement with label X in Y`** returns the labelled statement with the label `X` that belongs to the statement list of the block statement `Y`. If there is no that labelled statement, the instruction returns `()`.

The instruction **`delete locals for S in context C`** eliminates local variables and local constants of the block `S`. It is defined in the following way:

```
associate instruction (
  execute
    (V is local variable) := () ;
    (type of V) := ( ) ;
    (name of V) := ( ) ;
    (generating declaration of V) := () ;
```

```
    (invocation instance of V) := () ;
    (value of V) := ( )
  for each binder of variables (V)
    in patterns ((generating declaration of V))
    provided
    ((* V is local variable) or (* V is local constant))
    and ((* invocation instance of V) =
         (invocation instance of C)) and
    (parent of (* generating declaration of V)) = S
) with pattern (delete locals for S in context C)
  with parameters (S , C)
```

The branch

```
let S1 be * C , S2 be (find labelled statement
  with label (* label of S1) in S) in
  if (* S is block statement) and
    (* S1 is goto statement) and S2 != () holds
  then
    (jump S in context C := ()) ;
    end S in context (* parental context of C) := active
    send ((* C) , C) to garbage
```

transfers control from the goto statement `S1`, stored in the jump context `C`, to the labelled statement `S2`.

5. Conclusion

The following results have been presented in this paper:

- A new language of finite state machines called USL (Unified Semantic Language) intended for rapid development of formal verification-oriented operational semantics of modern programming languages has been presented.
- Formal operational semantics of the USL language has been defined.
- The USL-based approach to programming language semantics design has been applied to the definition of C# jump statements.

We plan to apply the USL language to the development of formal semantics of .NET programming languages.

References

[1] AsmL: The Abstract State Machine Language. Reference Manual. — 2002. — http://research.microsoft.com/fse/asml/doc/AsmL2_Reference.doc.

[2] Introducing AsmL: A Tutorial for the Abstract State Machine Language. — 2001. — http://research.microsoft.com/fse/asml/doc/AsmL2_Tutorial.doc.

[3] Wolfgang Grieskamp and Nikolai Tillmann. AsmL Standard Library Reference. 2002. http://research.microsoft.com/fse/asml/doc/AsmLStandardLibraryReference.doc.

[4] Kutter Ph. W., and Pierantonio A. Montages: Specifications of Realistic Programming Languages // J. of Universal Computer Science. — 1997. — Vol. 3, N 5. — P. 416–442.

[5] Anlauff M., Kutter Ph. W., Pierantonio A. Enhanced Control Flow Graphs in Montages // Proc. of Perspectives of System Informatics (PSI'99). — Lect. Notes Comput. Sci. — 1999. — Vol. 1755. — P. 40–53.

[6] Gurevich Yu. Evolving Algebras. Lipari Guide // Specification and Validation Methods. — Oxford University Press, 1995. — P. 9–36.

[7] Gurevich Yu. Draft of the ASM Guide. — Michigan, 1997. — (Tech. Rep. / Univ. of Michigan EECS Department; CSE-TR-336-97).

[8] Huggins J. Abstract State Machines Web Page. — http://www.eecs.umich.edu/gasm.

[9] Gordon M.J.C., Melham T.F. Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. — Cambridge University Press, 1993.

[10] Owre S., Shankar N., Rushby J.M. User Guide for the PVS Specification and Verification System. — Computer Science Laboratory, SRI International, Menlo Park, CA, 1993.

[11] Boyer R.S., Moore J.S. A Computational Logic Handbook. 2-nd Edition. — New York: Academic Press, 1997.

[12] M. Kaufmann, P. Manolios, J.S. Moore. Computer-Aided Reasoning: An Approach. — Boston, MA: Kluwer Academic Press, 2000.

[13] Moore J.S. Proving Theorems about Java and the JVM with ACL2. Working material for the lectures. International Summer School on Models, Algebras, and Logic of Engineering Software. — Marktoberdorf, 2002.

[14] Dowek G., Felty A., Herbelin H., Huet G. Paulin, C., Werner B. The Coq proof assistant user's guide, Version 5.6. — Rocquencourt, 1991. — (Tech. Rep. / INRIA; TR 134).

[15] ECMA-334. C# Language Specification. December 2001. — http://www.ecma.ch.

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