Bull. Nov. Comp. Center, Comp. Science, 34 (2012), 1–21

Program specific transition systems*

I. S. Anureev

Abstract. A new kind of labeled transition systems, program specific transition systems, is proposed. These systems are used to formalize and unify some aspects of program handling. Such aspects as the development of program operational semantics and proof of safety properties of programs are considered, and the appropriate classes of program specific transition systems are defined. Ontological transition systems and operational ontological semantics are defined in terms of program specific transition systems.

1. Introduction

A modern trend in the field of software verification is transition from the development of program verification methods used for small programs in model programming languages to their adaptation to verification of large software systems in industrial programming languages. In this case practically important properties of programs are picked out and specialized methods and techniques of analysis and verification, focused on these properties, are defined. Formalization and unification of processes of specification of these properties and development of methods and techniques for them is an important open problem. Industrial verification is also characterized by a combination of different verification methods. As a result, new hybrid program verification methods arise. Creation of methods and tools that accumulate, analyse and formalize experience in integration of different verification methods is another important open problem.

To solve these problems, we propose to use a new kind of labeled transition systems, program specific transition systems. These systems define the overall framework for description of specialized transition systems that formalize and unify various aspects of program handling.

Three classes of specialized transition systems are considered in this paper. Operational semantics specific transition systems are used for a rapid development of formal specifications of programming languages and software systems prototyping. Safety logic specific transition systems are used to formalize and unify the deductive program verification methods. Ontological transition systems are used to develop operational ontological semantics of programming languages and software systems. They describe the opera-

*Partially supported by RFBR under grant No.11-01-00028-p and SB RAS interdisciplinary integration project No.3.

tional semantics of programming languages and software systems based on their ontology.

2. Preliminary concepts and notation

This section describes the notation of basic data structures (lists, sequences, and functions) used in this paper both on the object level and the meta level.

Lists have the form (a$_1$ ... a$_n$), where a$_i$ are separated by spaces. Let (lists of x) denote the set of all lists of elements of the set x, (lists of x of length n) denote the set of all lists of elements of the set x of length n, (length of a) denote the length of the list a.

Finite sequences have the form a$_1$ ... a$_n$, where a$_i$ are separated by spaces. Let (sequences of x) denote the set of all finite sequences of elements of x and (sequences of x of length n) denote the set of all finite sequences of elements of x of length n.

Let bool = {true, false} and nat be the set of nonnegative integers. Let (union of A$_n$ where (n $\in$ x)) denote the union of the sets A$_n$ for all (n $\in$ x).

Let (f x) denote the application of the function f to the sequence x of arguments (call of the function f on the arguments x). Let undef denote the fact that a function has no value at some argument and ((domain of f) = {x | ((f x) $\neq$ undef)}) denote the domain of f. Let f and g be functions such that (((domain of f) $\cap$ (domain of g)) = $\emptyset$). Then the union (f $\cup$ g) of f and g is a function h such that ((domain of h) = (domain of f) $\cup$ (domain of g)), ((h x) = (f x)) for (x $\in$ (domain of f)) and ((h x) = (g x)) for (x $\in$ (domain of g)).

We also use an alternative attribute notation for (usually finite) functions. If f is a function, f is called an attribute structure, the elements of the domain of f are called attributes, and the object (f a) (denoted by f.a) is called the value of the attribute a of the structure f.

We define the operations of access . and update upd on functions as follows:

- (f.x = (f x));
- if (y $\neq$ x), then (((upd f x e) y) = (f y));
- (((upd f x e) x) = e).

The operations . and upd are extended to sequences and lists, if these structures were considered as functions with positive integer segments as their domains.

We say that a function f can differ from a function g only on a set x and denote this fact (f can differ from g only on x), if ((f a) = (g a)) for each (a $\notin$ x).

A labeled transition system `lts` is a triple `(states labels tr)`, where `states` and `labels` are sets whose elements are called states and labels, respectively, and a logical function `(tr` $\in$ `((states` $\times$ `labels` $\times$ `states)` $\rightarrow$`bool))` is called a transition relation. The system `lts` can transit from a state `s` to a state `ss` by a label `lab`, if `(tr s lab ss)`.

3. Program specific transition systems

Program specific transition systems (P-STSs) are labeled transition systems used to formalize various aspects of program handling (definition of program semantics, specification of program verification strategies, etc.).

States in P-STSs are algebraic structures of a special kind.

Let `atoms` be a set of objects called atoms. An expression is either an expression list or an atom. Let `expressions` denote the set of all expressions. In the context of the subsequent exposition, we call the elements of sets `(sequences of expressions)` and `(lists of atoms)` programs and symbols, respectively, and denote these sets by `programs` and `symbols`. The elements of the set `(programs = (sequences of expressions))` are called programs. Let `elements` be a set of symbols called elements such that `(expressions` $\subseteq$ `elements)`.

A state `s` w.r.t. `(atoms elements)` is defined as a total function from the set `((lists of atoms)` $\rightarrow$ `((union of (elements`^{n} $\rightarrow$ `elements),` `where (n` $\in$ `nat))` $\cup$ `{undef}))`. The set `elements` is called a carrier of `s`, and its elements are elements of `s`.

The set `{(f` $\in$ `symbols) | ((s f)` $\neq$ `undef)}` is called a signature of `s` and denoted by `(symbols of s)`, and the elements of this set are called symbols of `s`. The function `(s f)` is called an interpretation of the symbol `f` in the state `s`. In contrast to the standard definition of an algebraic system in which signature symbols are atoms, the symbols of a state are atom lists. This brings the description of function calls closer to the description in a natural language. The use of special atoms `_` and `__` in these symbols makes them function call templates. These atoms called argument specifiers designate places for function arguments. The atom `_` indicates that the corresponding argument should be first evaluated, and the atom `__` indicates that this argument does not need to be evaluated. Let `(f` $\in$ `(symbols of s))`. The number of occurences of argument specifiers in `f` is called the arity of `f` and denoted by `(arity of f)`. The following property for `s` should hold: if `(n = (arity of f))`, then `((s f)` $\in$ `(elements`^{n} $\rightarrow$ `elements))`.

For example, for the equality operation `=`, the corresponding symbol `f` has the form `(_ = _)` and `((s f)` $\in$ `((elements` $\times$ `elements)` $\rightarrow$ `bool))`. In this case, `(bool` $\subseteq$ `elements)`.

Let `(u, w` $\in$ `(sequences of expressions))`. Let `typed-args` be a list of structures with the attributes `arg` and `type` with values from `expressions`

and {value, itself}, called typed arguments. The expression e is an instance of f w.r.t. typed-args if and only if (The first proper rule is applied.)

- if (e = ()) and (f = ()), then (typed-args = ());
- if (e = (typed-args.1.arg u)) and (f = (_ w)), then (typed-args.1.type = value) and (u) is an instance of (w) w.r.t. (typed-args.2 ... typed-args.n);
- if (e = (typed-args.1.arg u)) and (f = (__ w)), then (typed-args.1.type = itself) and (u) is an instance of (w) w.r.t. (typed-args.2 ... typed-args.n);
- if (e = (b u)) and (f = (b w)), then (u) is an instance of (w) w.r.t. typed-args;
- false.

The expression e is called an instance of f, if e is an instance of f w.r.t. some typed-args. The same expression can be an instance of several symbols. We consider that there is a function (match $\in$ ((expressions $\times$ states) $\to$ ((symbols $\times$ expressions) $\cup$ {undef}))) which determines a symbol choice:

- if ((match e s) = (f typed-args)), then (f $\in$ (symbols of s)) and e is an instance of f w.r.t. typed-args;
- if ((match e s) = undef), then there are no (f $\in$ (symbols of s)) and typed-args such that e is an instance of f w.r.t. typed-args.

The value (value of e in s) of the expression e in the state s is defined as follows (The first proper rule is applied.):

- if (e $\in$ atoms), then ((value of e in s) = e);
- if e = ((ee)) and (ee $\in$ (sequences of expressions)), then ((value of e in s) = ((value of (ee) in s)));
- if (e $\in$ (symbols of s)), then ((value of e in s) = (s e));
- if ((match e) = (f typed-args)), then ((value of e in s) = ((s f) arg-values));
- ((value of e in s) = undef).

The list (arg-values $\in$ (lists of elements of length n)), where (n = (arity of f)), in the above definition is given as follows:

- if (typed-args.i.type = value), then (arg-values.i = (value of typed-args.i.arg in s));

- if (typed-args.i.type = itself), then (arg-values.i = typed-args.i.arg).

A function ($\sigma \in$ (atoms $\rightarrow$ (sequences of expressions))) is called a substitution. If (domain of σ) = {x_1, ..., x_n}, σ can be written as ((x_1 (σ x_1)) ... (x_n (σ x_n))). A substitution function subst w.r.t. σ is defined as follows (The first proper rule is applied.):

- if (e $\in$ (domain of σ)), then ((subst e σ) = (σ e));
- if (e $\in$ atoms), then ((subst e σ) = e);
- ((subst (e_1 ... e_n) σ) =
  ((subst e_1 σ) ... (subst e_n σ))).

We now describe the main components of a P-STS (p-sts = (states labels tr)).

The state predefined-interpretation defines the interpretation of predefined symbols. The interpretation of these symbols does not change when p-sts transits from one state to another. Let predefined-symbols denote the set (symbols of predefined-interpretation).

The set modifiable-symbols includes modifiable symbols. Their interpretation may change when p-sts transits from one state to another.

The sets predefined-symbols and modifiable-symbols can intersect. A state s of p-sts expands to predefined-symbols so that (((s f) a) = ((predefined-interpretation f) a)) for all (a $\notin$ (domain of (s f))).

The set modifiable-symbols includes special symbols (value _), (history length), and (_ is new-element).

The symbols (value _) are used to specify the intermediate values that appear when p-sts transits from one state to another. For each state s, the intermediate values are the values of expressions (value 1), ..., (value k) in s, where (k = (value of (history length) in s)).

Let ((new elements of s) = {(a $\in$ elements) | (((s (_ is new-element)) a) = true)}). This set specifies the elements of elements that have not been «used» in s and previous states of p-sts. The elements of this set are called new elements in s. For each (s $\in$ states), the symbol (_ is new-element) has the following properties:

- (((s (is new-element)) a) $\in$ bool) for each (a $\in$ elements). This property means the totality of the symbol (_ is new-element);
- if (tr s lab ss), then ((new elements of ss) $\subseteq$ (new elements of s)). This property means monotone nonincreasing of the set (new elements of s) w.r.t. tr;

- if (tr s lab ss), then ((new elements of s) \ (new elements of ss)) is finite. This property means that any transition «uses» only a finite number of new elements;
- (new elements of s) is infinite. This property means that there is a sufficient number of new elements to «be used» in any transition;
- ((value of e in s) ∈ (new elements of s)) for each (e ∈ expressions), and (((new elements of s) ∩ expressions) = ∅). These properties formalize the notion «not to be used».

A labeled transition system p-sts = (states labels tr) is called a P-STS w.r.t. (atoms elements predefined-interpretation modifiable-symbols), if lab = (p | pp) for some (p, pp ∈ programs), and for all (s ∈ states) the following conditions hold:

- s is a state w.r.t. (atoms elements);
- ((symbols of s) = (predefined-symbols ∪ modifiable-symbols)). This property means that the signature of any state contains exactly the symbols of predefined-symbols and modifiable-symbols;
- ((s (history length)) ∈ nat);
- if (tr s lab ss), then ((s (history length)) ≤ (ss (history length))) for each (ss ∈ states). This property means that the set of intermediate values can only be replenished;
- ((value of (value i) in s) = undef) for each (i ≥ (s (history length))). This property means that the set of intermediate values is limited by the constant (history length).

If (tr s (p | pp) ss), we say that p transforms s to ss w.r.t. pp. Thus, programs can be considered as state transformers.

A list (p s) is called a configuration. A configuration should meet the following restriction: if p contains an expression of the form (value a), then (a ∈ nat) and (a ≤ (s (history length))).

If (tr s (p | pp) ss), we say that p-sts can transit from (p s) to (pp ss). The configuration (p s) is called final, if there is no configuration (pp ss) such that (tr s (p | pp) ss). The state s is called final, if (p s) is final for each program p. The configuration is called a branchpoint, if p-sts can transit from it to more than one configuration.

A trace is a finite or infinite sequence of configurations (p_1 s_1) (p_2 s_2) ... such that p-sts can transit from (p_i s_i) to (p_{i+1} s_{i+1}).

A special atom backtrack specifies a dummy trace of program execution. A configuration (p s) is called a backtracking configuration, if (p.1 =

(backtrack)). The transition relation tr should satisfy the following condition: if (p s) is a backtracking configuration, then (p s) is final. A finite trace with a backtracking configuration as its last element is called dummy. Backtracking means that if p-sts reaches a backtracking configuration (p s), it backtracks to the nearest branchpoint and starts to perform another trace. If there are no such traces, p-sts backtracks to the previous branchpoint. If there are no such points, all traces are dummy (except for, possibly, the trace from the initial configuration).

A function (io-sem ∈ (programs → ((states × states) → bool))) is called an input-output semantics w.r.t. p-sts, if ((io-sem p) s ss) if and only if there is a finite non-dummy trace (p_1 s_1) ... (p_n s_n) such that ((p_1 s_1) = (p s)), (s_n = ss) and (p_n s_n) is final.

The transition relation tr is a union of transition relations, each of which is characterized by the kind of the expression p.1. In turn, these expressions are divided into regular and irregular ones. The transition relation for a regular expression p.1 is given by transition rules. These rules are also called the rules of operational semantics of p.1. The transition relation for each kind of irregular expressions is defined by a specific way.

A transition rule r has the form (if sam var x hvar w then c), where (sam ∈ expressions), (x ∈ (sequences of expressions)), (w ∈ symbols), and (c ∈ programs).

In the case of ((length of x) = 0) or ((length of w) = 0), the corresponding components var x and hvar w can be omitted.

The expression sam is called a sample of r. The sample sam defines the set of expressions p.1 to which r can be applied.

The elements of x are called sample variable specifiers. If a specifier u is an atom, it specifies a sample variable u of the type exp. If u has the form (seq v), where (v ∈ atoms), it specifies the sample variable v of the type seq. The result of matching p.1 with sam is stored in the sample variables. The variables of the type exp store expressions, and the variables of the type seq store programs. Let u.var denote the variable specified by u, and u.type denote the type of u.var.

The elements of w are called history variables. They are used to store intermediate values in (value _). Transition with the help of r increases the value of (history length) by (length of w).

The program c is called the body of r. The result cc of modification of c according to the values of sample and history variables replaces p.1 in p, resulting in pp. The body c should not include instances of the symbols (value _) and (history length).

The sets of modifiable symbols, kinds of irregular expressions, their semantics, and semantics of transition rules may be different for different kinds of P-SPS.

4. Operational semantics specific transition systems

Operational semantics specific transition systems (OS-STSs) is a special kind of P-STSs used for description of operational semantics of programs.

The set `modifiable-symbols` includes a special symbol `(value)` which is used to specify the value that an OS-STS can return when it transits from one state to another. We say that `p` returns the value `v` w.r.t. `pp`, if `(tr s (p | pp) ss)` for some `ss` and `((ss (value)) = v)`. We say that `p` returns the value `v`, if `p` returns the value `v` w.r.t. some `pp`.

Irregular expressions in OS-STSs are of five kinds.

The irregular expression `(stop)` is called a stop and has the following semantics: `(tr s ((stop) p | pp) ss)` if and only if `(ss = s)` and `pp` is an empty sequence.

The irregular expression `(assume a)` is called a continuation condition and has the following semantics: `(tr s ((assume a) p | pp) ss)` if and only if `(ss = s)` and (The first proper rule is applied.)

- if `((value of a in s) = true)`, then `(pp = p)`;
- `(pp = (backtrack) p)`.

The irregular expression `(modify a)` is called an update condition and has the following semantics: `(tr s ((modify a) p | pp) ss)` if and only if (The first proper rule is applied.)

- if `((value of a in s wrt ss) = true)`, then `(pp = p)` and `(ss can differ from s only on x)`, where `x` is a set of modifiable symbols for which there is an example `(e)` such that `(: e)` occurs in `a`;
- `(ss = s)` and `(pp = (backtrack) p)`.

The value `(value of e in s wrt ss)` of the expression `e` in `s` w.r.t. `ss` is defined as follows (The first proper rule is applied.):

- if `(e ∈ atoms)`, then `((value of e in s wrt ss) = e)`;
- if `e = ((ee))` and `(ee ∈ (sequences of expressions))`, then `((value of e in s wrt ss) = ((value of (ee) in s wrt ss)))`;
- if `(e = (: ee))` and `((ee) ∈ (symbols of s))`, then `((value of e in s wrt ss) = (ss (ee)))`;
- if `(e = (: ee))` and `((match (ee) ss) = (f typed-args))`, then `((value of e in s wrt ss) = ((ss f) arg-values))`;
- if `(e.1 ≠ :)` and `(e ∈ (symbols of s))`, then `((value of e in s wrt ss) = (s e))`;

- if `(e.1` $\neq$ `:)` and `((match e s) = (f typed-args))`, then `((value of e in s wrt ss) = ((s f) arg-values))`;
- `((value of e in s wrt ss) = undef)`.

The list `arg-values` in the above definition is given as in the definition of `(value of e in s)`. A special atom `:` in the expression `(: ee)` means that the symbol for which `(ee)` is an instance is interpreted in `ss`.

The irregular expression `(modify a else b)` is called an update condition with alternative and has the following semantics: `(tr s ((modify a else b) p | pp) ss)` if and only if (The first proper rule is applied.)

- if `((value of a in s wrt ss) = true)`, then `(pp = p)` and `(ss can differ from s only on x)`, where `x` is a set of modifiable symbols for which there is an example `(e)` such that `(: e)` occurs in `a`;
- `(ss = s)` and `(pp = b p)`.

The irregular expression `(a ::= b)` is called a symbol update and has the following semantics: `(tr s ((a ::= b) p | pp) ss)` if and only if (The first proper rule is applied.)

- if `((match a s) = (f typed-args))` and `(f` $\in$ `modifiable-symbols)`, then `ss = s` and `(pp = (modify ((: f) = (updv f args b))) p)`
- `(ss = s)` and `(pp = (fail) p)`.

The predefined symbol `(updv _ _ _)` has the following interpretation: `((value of (updv g (`x_1 `...` x_n`) y) in s) = (upd (value of g in s) ((value of` x_1 `in s) ...` `(value of` x_n `in s)) (value of y in s)))`.

The list `args` $\in$ `(lists of expressions of length (arity of f))` is defined as follows:

- if `(typed-args.i.type = itself)`, then `(args.i = (quote typed-args.i.arg))`;
- if `(typed-args.i.type = value)`, then `(args.i = typed-args.i.arg)`.

The predefined symbol `(quote __)` has the following interperetation: `((value of (quote a) in s) = a)`.

We now define the semantics of the transition rule `r` in `p-sts`.

Let σ be a substitution on the sample variables of `r` such that `((`σ `x.i.var)` $\in$ `expressions)` for `(x.i.type = exp)` and `((`σ `x.i.var)` $\in$

(sequences of expressions)) for (x.i.type = seq), δ is a substitution on the history variables of r such that (δ w.j) = (value ((value of (history length) in s) + j)) and (γ = ($\sigma \cup \delta$)).

Let (cc $\in$ programs) be defined as follows: ((length of cc) = (length of c)) and (cc.k = (del* (subst c.k γ) s)) for all (1 $\leq$ k $\leq$ (length of c)).

The function del* evaluates expressions marked by a special atom * in the body c of r, and is defined as follows (The first proper rule is applied.):

- ((del* (* a) s) = (value of (del* a) in s));
- ((del* (a_1 ... a_n) s) = ((del* a_1 s) ... (del* a_n s)));
- ((del* a s) = a).

Let ((length of w) = m) and ((length of p) = n). The transition relation tr for a regular expression p.1 is defined by transition rules as follows: (tr s (p | pp) ss) if and only if there is a substitution σ such that

- ((subst a σ) = p.1);
- (ss can differ from s only on {(history length)}) and ((ss (history length)) = ((ss (history length)) + m));
- (pp = cc p.2 ... p.n).

As an example of application of rules of OS-STSs, we define operational semantics of expressions often used in OS-STSs.

The expression (new element) called a new symbol generator is defined by the rule

```
(if (new element) then (modify (((: value) is new-element) and
  ((: _ is new-element) =
   (updv (_ is new-element) ((: value)) false)))))
```

A special atom fail denotes the incorrect termination of the program and is called an unsafe termination. A configuration (p s) is called unsafe, if (p.1 = (fail)). Otherwise, (p s) is called safe. The transition relation tr should satisfy the following condition: if (p s) is unsafe, then (p s) is final. A finite trace with an unsafe configuration as its last element is called unsafe.

The expression (assert a) called a safety condition is defined by the rules

```
(if (assert a) var a then (assume a))

(if (assert a) var a then (assume (not a)) (fail) (stop))
```

The expression `(restart)` called a restart is defined by the rule

```
(if (restart) then (modify (((: value) = undef) and
  (forall x ((: value x) = undef)) and ((: history length) = 0))))
```

5. Safety logic specific transition systems

Let `os-sts` be an OS-STS. Let us define the notion of program safety in `os-sts`.

A program `p` is called safe in `os-sts` w.r.t. a state `s`, if there is no finite trace $(p_1\ s_1)\ \ldots\ (p_n\ s_n)$ such that $((p_1\ s_1) = (p\ s))$ and $(p_n\ s_n)$ is unsafe.

A program `p` is called safe in `os-sts` w.r.t. a precondition `(pre ∈ expressions)`, if `p` is safe in `os-sts` w.r.t. `s` for each `s` such that `((value of pre in s) = true)`.

Safety logic specific transition systems (SL-STSs) are P-STSs of a special kind, and they are used to check safety properties of programs. Let `sl-sts` be an SL-STS. Transition rules of `sl-sts` reduce checking of safety of a program `p` in `os-sts` w.r.t. `pre` to proving a set of expressions called verification conditions. If all these verification conditions are true, then `p` is safe in `os-sts` w.r.t. `pre`.

A P-STS `sl-sts` is called a safety logic specific transition system w.r.t. `os-sts`, if it satisfies the following properties.

The set `modifiable-symbols` of `sl-sts` includes special symbols `(precondition)` and `(version of _)`. The symbol `(precondition)` stores a precondition. The set `(domain of (s (version of _)))` is the same for any state `s` of `sl-sts` and coincides with the set `modifiable-symbols` of `os-sts`. This set is denoted by `(domain of version)`. The element `((value of (version of f) in s) ∈ nat)` is the number of the previous states of `os-sts` in which the interpretation of `f` could be changed. This element is called a version of `f` in `s`.

Since `(value)` and `(_ is new-element)` belong to `modifiable-symbols` of `os-sts`, `sl-sts` uses `(value*)` and `(_ is new-element*)` instead of it.

Irregular expressions in SL-STSs are of seven kinds.

The irregular expression `(stop)` is defined as the same for OS-STSs.

The irregular expression `(modify a)` is called an update condition and has the following semantics: `(tr s ((modify a) p | pp) ss)` if and only if `(pp = p)` and (The first proper rule is applied.)

- `(ss can differ from s only on {(precondition), (version of _)})`,
- `((ss (precondition)) = ((s (precondition)) and (add version to a wrt s)))`,

- if (f ∈ x), where x is a set of modifiable symbols for which there is an instance (e) such that (: e) occurs in a, then ((ss (version of f)) = ((s (version of f)) + 1)),
- if (f ∈ (modifiable symbols) \ x), then ((ss (version of f)) = (s (version of f))).

A special atom : in (: e) means that the symbol for which (e) is an instance increases its version by one when sl-sts transits from s to ss.

Let the symbol (concatenate _ and _) be interpreted as a concatenation of atoms in the case when both arguments are atoms. Let (f ∈ (domain of version)). The expression (add version to a wrt s) adds versions for all symbols of (domain of version) in a, and has the following semantics (The first proper rule is applied.):

- if (a ∈ atoms), then ((add version to a wrt s) = a);
- if a = (: b)) and (b) is an instance of f, then ((add version to a wrt s) = ((concatenate : and ((value of (version of f) in s) + 1)) (add version to b wrt s)));
- if a = (b) and a is an instance of f, then ((add version to a wrt s) = ((concatenate : and (value of (version of f) in s)) (add version to b wrt s)));
- if a = (b), then ((add version to a wrt s) = ((add version to b wrt s)));
- ((add version to a_1 ... a_n s) = (add version to a_1 wrt s) ... (add version to a_n wrt s)).

The irregular expression (assume a) is called a continuation condition and has the following semantics: (tr s ((assume a) p | pp) ss) if and only if (pp = p), (ss can differ from s only on {(precondition)}) и ((ss (precondition)) = ((s (precondition)) and (add version to a wrt s))).

The irregular expression (a ::= b) is called a symbol update and has the following semantics: (tr s ((a ::= b) p | pp) ss) if and only if (The first proper rule is applied.)

- if ((match a s) = (f typed-args)) and (f ∈ modifiable-symbols), then ss = s and (pp = (modify ((: f) = (updv f args b))) p)
- (ss = s) and (pp = (fail) p).

Semantics of the irregular expressions (assume* a), (modify* a), and (a ::=* b) called an operational continuation condition, an operational update condition, and operational symbol update, respectively, coincides with

semantics of expressions `(assume a)`, `(modify a)`, and `(a ::= b)` in OS-STSs.

Semantics of transition rules of SL-STSs coincides with the semantics of transition rules of OS-STSs.

As an example of application of the rules of SL-STSs, we define the operational semantics of expressions often used in SL-STSs.

The expression `(fail)` called an unsafe termination is defined by the rule

```
(if (fail)
 then ((value*) ::=* (quote (not (* precondition)))) (stop))
```

The expression `(new element)` called a new element generator is defined by the rule

```
(if (new element) then (modify (((: value) is new-element) and
  ((: _ is new-element) =
   (updv (_ is new-element) ((: value)) false)))))
```

The expression `(new element*)` called an operational new element generator is defined by the rule

```
(if (new element*) then (modify* (((: value) is new-element*) and
  ((: _ is new-element*) =
   (updv (_ is new-element*) ((: value)) false)))))
```

The expression `(assert a)` called a safety condition is defined by the rule

```
(if (assert a) var a then (assume a))

(if (assert a) var a then (assume (not a)) (fail) (stop))
```

The expression `(assert* a)` called an operational safety condition is defined by the rule

```
(if (assert* a) var a then (assume* a))

(if (assert* a) var a then (assume* (not a)) (fail) (stop))
```

The expression `(restart with precondition a)` called a restart is defined by the rule

```
(if (restart with precondition a) var a
 then (modify* (((: value*) = undef) and
  (forall x ((: value x) = undef)) and ((: history length) = 0) and
  (forall x ((: version of x) = 0)) and
  ((: precondition) = (add null version to a)))))
```

The expression `(add null version to a)` puts in `a` the null version of all symbols of `(domain of version)` and has the following definition (The first proper rule is applied.):

- if `(a` $\in$ `atoms)`, then `((add null version to a) = a)`;
- if `a = (b)` and `a` is an instance of `(f` $\in$ `(domain of version))`, then `((add null version to a) = (:0 (add null version to b)))`;
- if `a = (b)`, then `((add null version to a) = ((add null version to b)))`;
- `((add null version to` a_1 `...` a_n `s) = (add null version to` a_1 `wrt s) ...` `(add null version to` a_n `wrt s))`.

A set of verification conditions generated by `sl-sts` for the program `p` w.r.t. `pre` is defined as the set of values of the symbol `(value*)` in all states `s` such that `((io-sem (reset with precondition pre) p (stop)) undef-state s)`, where `undef-state` is a state such that `((undef-state f) = undef)` for all `(f` $\in$ `(symbols of undef-state))`.

6. The definition of ontological transition systems in terms of operational semantics specific transition systems

The operational ontological approach to formal specification of a programming language, based on ontology of this language, has been proposed in [14]. A new kind of program semantics, operational ontological semantics, was introduced in the framework of the approach. In contrast to usual operational semantics, which does not impose any restrictions on states, in the operational ontological semantics states are defined as ontological models (sets of instances of concepts of the programming language ontology).

A formalism for description of operational ontological semantics, ontological transition systems (OTSs) [2, 3, 12, 13], has also been proposed.

In this section we define OTSs and operational ontological semantics using OS-STSs. This allows us to combine the advantages of the ontological approach with the expressiveness of the formalism of OS-STSs.

An OTS `ots` is an OS-STS such that the set `modifiable-symbols` includes two subsets `ontological-symbols` and `instantiation-symbols`.

The elements of `ontological-symbols` are called ontological symbols. They specify the elements (concepts, attributes, relations and so on) of a programming language ontology.

Let us consider an example of the set `ontological-symbols`. We use the letters `a`, `b`, `c` instead of argument specifiers to determine the informal meaning of symbols of this set:

- `(a is concept)` means that `a` is a concept

- `(a is attribute of b)` means that `a` is an attribute of the concept `b`;
- `(a is attribute of b of type c)` means that `a` is an attribute of the concept `b` of type `c`. The type `c` is a concept. For example, `(a is attribute of b of integer)` means that `a` is an integer attribute of the concept `b`.

The elements of `instantiation-symbols` are called instantiation symbols. They specify relations of the ontology elements with their instances.

Let us consider an example of the set `instantiation-symbols`:

- `(a is b)` means that `a` is an instance of the concept `b`;
- `(a of b)` is the value of the attribute `a` of the instance `b` of a concept. For example, if `b` is an assignment statement of the form `(u := v)`, then the expression `(left-side of b)` is the left side `u` of this statement;
- `(a of b of c)` is the value of the attribute `a` of the instance `b` of the concept `c`. This symbol is used to resolve the conflict when `b` is an instance of several concepts that have the attribute `a`.

Now we can formally define the operational ontological semantics of a programming language. Let `L` be a programming language. The operational ontological semantics of `L` is defined as a pair `(p ots)`, where `p` $\in$ `programs` and `ots` is an OTS.

The program `p` is called a specification of an ontology of `L`. The result of execution of `p` is an ontology of `L`. The OTS `ots` defines semantics of executable concepts of `L`. An executable concept is a concept that describes a set of executable entities of `L`.

Let us note that the ontology of a modern programming language includes not only a description of its constructions and their constituent elements, but also the description of fundamental concepts such as exception propagation, overload resolution, communications of applications with operational environments, finding the dynamic type of an object in object-oriented programming languages, and etc.

The OTS `ots` should preserve the ontology of `L`. An OTS preserves ontology, if transitions of this OTS do not change interpretations of ontological symbols.

7. Example

We illustrate the application of P-STSs by an example of a toy programming language `L`.

Let `s` be a current state. The language `L` includes the following kinds of expressions:

- `(block p)` sequentially executes expressions from `(p ∈ programs)`;
- `x` returns the value of the variable `(x ∈ atoms)`. This expression is called a variable access. If there is no variable named `x`, `(fail)` is generated. For simplicity, we assume that all variables are of integer type. Integers are specified by the predefined symbol `(_ is integer)`;
- `(x := e)` assigns the value of the expression `e` to the variable `x`. If `(x := e)` is the first assignment to the atom `x`, this assignment serves as the declaration of the variable `x` with the initializer `e`. For simplicity, we assume that `e` is either a variable access or an integer;
- `c` returns `c`, if `((value of (c is integer) in s) = true)`.
- `(if x then y else z)` is a conditional statement with the condition `(x ∈ expressions)`, the `then`-branch `(y ∈ programs)` and the `else`-branch `(z ∈ programs)`;
- `(while x do y)` is a loop with the condition `(x ∈ expressions)` and the body `(y ∈ programs)`;
- `(random)` returns an integer.

First, we consider the application of OS-STSs to the development of the operational semantics of `L` and describe the steps of the definition of the OS-STS `os-sts` which specifies operational semantics of `L`.

In the first step, the set `modifiable-symbols` of `os-sts` is defined. It includes the symbols `(_ is variable)` and `(value of _)` in addition to the basic symbols like `(value)`. The symbol `(_ is variable)` defines the set of variables of a program in `L`. The symbol `(value of _)` stores the values of variables of this program.

In the second step, the transition rules of `os-sts` that specify the semantics of `L` expressions are defined:

```
(if (block a) var (seq a) then a)

(if a var a
 then (assert (a is variable)) ((value) ::= (value of a)))

(if a var a then (assume (a is integer)) ((value) ::= a))

(if (a := b) var a b
 then b ((a is variable) ::= true) ((value of a) ::= (value)))

(if (if a then b else c) var a (seq b) (seq c)
 then a (assume ((value) = true)) b)

(if (if a then b else c) var a (seq b) (seq c)
 then a (assume ((value) = false)) c)
```

```
(if (while a do b) var a (seq b)
 then (assume a) b (while a do b))

(if (while a do b) var a (seq b) then (assume (not a)))

(if (random) then (modify ((: value) is integer)))
```

In the third step, the set `predefined-symbols` of `os-sts` is defined. It consists of the predefined symbols used in transition rules of `os-sts`. A description of these symbols and their interpretations completes the definition of a state of `os-sts`. In our case, `predefined-symbols` includes the symbols `(_ = _)`, `(not _)` and `(_ is integer)` with their usual interpretation (equality, negation and «to be an integer»).

Let us note that if a programming language has a syntax that is different from the expression syntax, a preliminary step is required. In this step, an equivalence relation between the constructs of the language and expressions is determined, and its constructs are translated into the equivalent expressions. Since this translation can be considered as a denotational semantics of some kind, the formal semantics of the programming language is defined as a combined denotational-operational semantics.

We now consider the application of SL-STSs to the development of a safety logic for `L`. SL-STSs are defined so that their rules for many programming language constructs are syntactically identical to the rules of the corresponding OS-STSs. In our case, the safety rules for `L` coincide with the corresponding rules of the operational semantics of `L` except for the loop rules.

The loop `(while a do b)` is replaced by the annotated loop `(while a invariant i do b)` with the invariant `i`. The safety logic for this loop is as follows:

```
(if (while a invariant i do b) var a i (seq b)
 then (assert i) (stop))

(if (while a invariant i do b) var a i (seq b)
 then (precondition ::=* i) (assume a) b (assert i) (stop))

(if (while a invariant i do b) var a i (seq b)
 then (precondition ::=* i) (assume (not a)))
```

Finally, we consider the application of OTSs for the development of operational ontological semantics.

We first consider the program `p` which is building the ontology of `L`:

```
((block is concept) ::= true)
```

```
((sequence is attribute of block) ::= true)

((identifier is concept) ::= true)

((variable is concept) ::= true)
((value is attribute of variable)) ::= true)

((constant is concept) ::= true)

((assignment is concept) ::= true)
((left-side is attribute of assignment)) ::= true)
((right-side is attribute of assignment)) ::= true)

((if-statement is concept) ::= true)
((condition is attribute of if-statement)) ::= true)
((then is attribute of if-statement)) ::= true)
((else is attribute of if-statement)) ::= true)

((while-statement is concept) ::= true)
((condition is attribute of while-statement)) ::= true)
((body is attribute of while-statement)) ::= true)

((random is concept) ::= true)
```

We now define the OTS which specifies the operational ontological semantics with the following set of rules:

```
(if a var a then (assume (a is block)) (* sequence of a))

(if a var a then (assume (a is identifier))
  (assert (a is variable)) ((value) ::= (value of a)))

(if a var a then (assume (a is integer)) ((value) ::= a))

(if a var a then (assume (a is assignment))
 (* right-side of a) ((a is integer) ::= true)
 ((value of (* left-side of a)) ::= (value)))

(if a var a then (assume (a is if-statement))
  (* condition of a) (assume ((value) = true)) (* then of a))

(if a var a then (assume (a is if-statement))
  (* condition of a) (assume ((value) = false)) (* else of a))

(if a var a then (assume (a is while-statement))
  (assume (* condition of a)) (* body of a) a)
```

```
(if a var a then (assume (a is while-statement))
 (assume (not (* condition of a))))

(if a then (assume (a is random)) (modify ((: (value)) is integer)))
```

8. Conclusion

In this paper a new kind of labeled transition systems, program specific transition systems, is proposed and three classes of such systems — operational semantics specific transition systems, safety logic specific transition systems and ontological transition systems — are considered.

We plan to unify the previously developed various kinds of operational [24, 27] and axiomatic [6, 18, 19, 20, 23, 25, 28] semantics on the basis of operational semantics specific transition systems and safety logic specific transition systems, respectively, and to integrate them into the multilingual system of program analysis and verification Spectrum [9, 21]. Program specific transition systems will be used in a new definition of the domain-specific language Atoment [10, 16] on which the system Spectrum is based, and in the knowledge portal on computer languages [1, 17] for formal specification of these languages.

We also plan to develop new specialized labeled transition systems to specify the previously developed simplification methods for verification conditions [4, 5, 8, 11, 15], context machines [7] and program translation algorithms in the multi-level methods of verification of C-light [6, 26] and C# [22, 23] programs.

References

[1] Andreeva T.A., Anureev I.S., Bodin E.V., Gorodnyaya L.V., Marchuk A.G., Murzin F.A., Shilov N.V. Educational significance of classification of computer languages // Prikladnaya informatika. – 2009. – No.6 (24). – P. 18–28 (In Russian).

[2] Anureev I.S. A language of actions in ontological transition systems // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2007. – IIS Special Iss. 26. – P. 19–38.

[3] Anureev I.S. A Language of description of ontological transition systems OTSL as a tool for formal specification of program systems // Vestnik NGU. Ser. Information Technologies. – 2008. – Vol. 6, No.3. – P. 24–34 (In Russian).

[4] Anureev I.S. A method for simplification procedures design based on formula rewriting systems // Joint NCC&IIS Bulletin. Ser. Computer Science. – Novosibirsk, 1998. – Iss. 8. – P. 1–18.

[5] Anureev I.S. A method of elimination of data structures based on formula rewriting systems // Programming and Computer Software. – 1999. – Vol. 25, No.4. – P. 184–192.

[6] Anureev I.S. A three-stage method of C program verification // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2008. – IIS Special Iss. 28. – P. 1–29.

[7] Anureev I.S. Context machines as formalism for specification of dynamic systems // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2009. – IIS Special Iss. 29. – P. 1–16.

[8] Anureev I.S. Formula rewriting systems and their application to automated program verification // Joint NCC&IIS Bulletin. Ser. Computer Science. – Novosibirsk, 1999. – Iss. 10. – P. 1–5.

[9] Anureev I.S. Integrated approach to analysis and verification of imperative programs // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2011. – IIS Special Iss. 32. – P. 1–18.

[10] Anureev I.S. Introduction to the Atoment language // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2010. – IIS Special Iss. 31. – P. 1–16.

[11] Anureev I.S. Multi-branch narrowing: satisfiability and termination // Joint NCC&IIS Bulletin. Ser. Computer Science. – Novosibirsk, 2000. – Iss. 13. – P. 1–11.

[12] Anureev I.S. Ontological models in OTSL // Problems in Programming. – 2008. – No.2-3. – P. 41–49.

[13] Anureev I.S. Ontological transition systems // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2007. – IIS Special Iss. 26. – P. 1–18.

[14] Anureev I.S. Operational ontological approach to formal programming language specification // Programming and Computer Software. – 2009. – Vol. 35, No.1. – P. 35–42.

[15] Anureev I.S. Program verification based on specification language Simple // Joint NCC&IIS Bulletin. Ser. Computer Science. – Novosibirsk, 2001. – Iss. 15. – P. 1–16.

[16] Anureev I.S. Typical examples of using the Atoment language // Automatic Control and Computer Sciences. – 2012. – Vol. 46, No.7. – P. 299–307.

[17] Anureev I.S., Bodin E.V., Gorodnyaya L.V., Marchuk A.G., Murzin F.A., Shilov N.V. On the problem of computer language classification // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2008. – IIS Special Iss. 28. – P. 31–42.

[18] Anureev I.S., Bodin E.V., Shilov N.V. Effective generation of verification conditions for non-deterministic unstructured programs // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2007. – IIS Special Iss. 26. – P. 39–64.

[19] Anureev I.S., Maryasov I.V., Nepomniaschy V.A. C-Programs Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. – 2011. – Vol. 45, No.7. – P. 485–500.

[20] Atuchin M.M., Anureev I.S. Attribute annotations and their use in C program deductive verification // Automatic Control and Computer Sciences. – 2012. – Vol. 46, No.7. – P. 308–316.

[21] Nepomniaschy V.A., Anureev I.S., Atuchin M.M., Maryasov I.V., Petrov A.A., Promsky A.V. C Program Verification in SPECTRUM Multilanguage System // Automatic Control and Computer Sciences. – 2011. – Vol. 45, No.7. – P. 413–420.

[22] Nepomniashy 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. – Novosibirsk, 2004. – Iss. 20. – P. 61–85.

[23] Nepomniaschy V.A., Anureev I.S., Dubranovskii I.V., Promsky A.V. Towards verification of C# programs: a three-level approach // Programming and Computer Software. – 2006. – Vol. 32, No.4. – P. 190–202.

[24] Nepomniaschy V.A., Anureev I.S., Mikhailov I.N., Promskii A.V. Towards Verification of C Programs. C-light language and its formal semantics // Programming and Computer Software. – 2002. – Vol. 28, No.6. – P. 314–323.

[25] Nepomniaschy V.A., Anureev I.S., Promskii A.V. Towards verification of C programs: axiomatic semantics of the C-kernel language // Programming and Computer Software. – 2003. – Vol. 29, No.6. – P. 338–350.

[26] Nepomniaschy V.A., Anureev I.S., Promsky A.V. Towards verification of C programs. Language ïS-light and its trasnformational semantics // Problems in Programming. – 2006. – No.2–3. – P. 359–368 (In Russian).

[27] Nepomniaschy V.A., Anureev I.S., Promsky A.V. Verification-oriented language C-light and its structural operational semantics // PSI-2003. Proc. of Conf. – Lect. Notes Comput. Sci. – 2003. – Vol. 2890. – P. 103–111.

[28] Shilov N.V., Anureev I.S., Bodin E.V. Generation of correctness conditions for imperative programs // Programming and Computer Software. – 2008. – Vol. 34, No.6. – P. 307–321.