Bull. Nov. Comp. Center, Comp. Science, 37 (2014), 23–39

Domain-specific transition systems and their application to a formal definition of a model programming language*

I. S. Anureev

Abstract. The paper presents a new object model of domain-specific transition systems, a formalism designed for the specification and validation of formal methods for assuring software reliability. A formal definition of a model programming language is given on the basis of this model.

Keywords: state transition systems, domain-specific transition systems, operational semantics.

1. Introduction

Assuring software reliability is an urgent problem of the theory and practice of programming. Formal methods play an important role in solving this problem. Currently, there are quite a lot of reliable software development tools based on formal methods. They cover many aspects, from design and prototyping of software systems to their formal specification and verification.

However, while in the Semantic Web there is a tendency to integrate heterogeneous data and services, in the reliable software development we are still dealing with a set of separate tools, each of which covers only certain specific aspects of the development and, as a rule, is designed for use only with a small number of computer languages. The gap between the great potential of formal methods and, with a rare exception, toy examples of their application is also noticeable [11]. Among the obstacles that prevent a widespread introduction of formal methods to software development, we note the difficulties to master them, the high price of their introduction, and the fact that the software engineers and programmers are skeptical about them. Insufficient attention is also focused on the technological aspects of the development of formal semantics of computer languages, which plays an important role in assuring the software reliability.

A unified approach to assuring the software reliability which covers the stages of software development such as prototyping, design, specification, and verification of software systems was proposed in [10, 6, 2]. This approach was also used to develop a formal operational semantics and safety logic

*Partially supported by the RAS project N 15/10 and SB RAS interdisciplinary integration project N 3.

(a variant of axiomatic semantics) of computer languages [5]. It is based on a special kind of transition systems, domain-specific transition systems (DSTSs).

DSTSs can also be considered as "technological" abstract state machines [8], in which the rules for defining the states and the transition relation are explicitly specified. In this case, DSTSs provide a higher level of abstraction in specifying software systems in comparison with the implementation languages of abstract state machines ASML [7] and XASM [9].

The Atoment language for specification of DSTSs and the sublanguages for specification of particular kinds of DSTSs focused on solving the tasks of reliable software development were presented in [6, 4, 3]. The Atoment-oriented object model of DSTSs was presented in [1]. In this paper, we describe the language-independent object model of DSTS and apply it to define formally a model programming language.

2. Preliminaries

Let $\mathtt{int}$, $\mathtt{nat}$, and $\mathtt{bool}$ denote the set of integers, the set of natural numbers, and the set $\{\mathtt{true}, \mathtt{false}\}$, respectively.

Let Set^* denote the set of all finite sequences consisting of the elements of a set Set, Set^+ denote the set of all finite nonempty sequences consisting of the elements of Set, and $\mathtt{pset}(Set)$ denote the set of all subsets of Set. Let $\mathtt{empseq}$ denote the empty sequence, and $El_1 \ \dots \ El_N$ denote the sequence consisting of the elements $El_1, \dots, El_N$. Let $\mathtt{len}(Seq)$, $Seq.I$, $\mathtt{first}(Seq)$, and $\mathtt{last}(Seq)$ denote the length of a sequence Seq and its I-th, first, and last elements, respectively.

Let $Set \to Set'$ denote the set of all functions from Set to Set', and $Set \to_t Set'$ denote the set of all total functions from Set to Set'. Let $\mathtt{dom}(Fun)$ denote the domain of a function Fun, and $\mathtt{und}$ denote the indeterminate value. We assume that $Fun(Arg) = \mathtt{und}$, if $Arg \notin \mathtt{dom}(Fun)$. Let $\mathtt{dom}(Fun) \cap \mathtt{dom}(Fun') = \emptyset$. The union $Fun \cup Fun'$ of the functions Fun and Fun' is defined as the function Fun'' such that $\mathtt{dom}(Fun'') = \mathtt{dom}(Fun) \cup \mathtt{dom}(Fun')$, $Fun''(Arg) = Fun(Arg)$ for $Arg \in \mathtt{dom}(Fun)$, and $Fun''(Arg) = Fun'(Arg)$ for $Arg \in \mathtt{dom}(Fun')$. Let $\mathtt{range}(Fun)$ denote the range of Fun, i.e. the set $\{Fun(Arg) \mid Arg \in \mathtt{dom}(Fun)\}$. Let $\mathtt{graph}(Fun)$ denote the graph of Fun, i.e. the set $\{(Arg, Fun(Arg)) \mid Arg \in \mathtt{dom}(Fun)\}$.

The boolean function $\mathtt{odif}$ is defined as follows: $\mathtt{odif}(Fun, Fun', Set) = \mathtt{true}$ if and only if $Fun(Arg) = Fun'(Arg)$ for $Arg \notin Set$. Thus, the values of the functions Fun and Fun' may differ only at the elements of Set.

We say that Set is defined by the functions $Fun_1, \dots, Fun_N$, if $\mathtt{dom}(Fun_I) = Set$ for each $1 \le I \le N$, and information about Set is specified only by these functions. For simplicity, we will omit the only argument of these functions where it will not cause collisions. For example, we can write

Fun_1 instead of $Fun_1(El)$ for some implicit argument $El \in Set$.

3. The main concepts of the theory of domain-specific transition systems

The set `dsts` of domain-specific transition systems is defined by the functions `el`, `par`, `elgen`, `frm`, `match`, and `atom`.

The set $\mathtt{el}(Dsts)$ of elements includes integers and boolean values, i.e. $\mathtt{int} \cup \mathtt{bool} \subset \mathtt{el}$. The element sequences (in particular, elements as one-element sequences) have static and dynamic semantics. The static semantics of $Elseq$ defines the value $\mathtt{val}(Elseq, St) \in \mathtt{el}$ returned by $Elseq$ in the current state St of the system $Dsts$. The function `val` and the set `st` of states are defined below. In this case, $Elseq$ can be considered as a query to $Dsts$ to get information about the current state St of the system $Dsts$. The dynamic semantics of $Elseq$ defines how $Elseq$ change the current state of $Dsts$, i.e. it defines the set of states to which $Dsts$ can go from the current state by $Elseq$. In this case, $Elseq$ can be considered as an instruction controlling the state of $Dsts$.

The set $\mathtt{par}(Dsts)$ of parameters, where $\mathtt{par}(Dsts) \subseteq \mathtt{el}$, is defined by the functions `vkind`, `skind`, and `catched` such that $\mathtt{skind}(Par) \in \{\mathtt{elem}, \mathtt{seq}\}$, $\mathtt{vkind}(Par) \in \{\mathtt{eval}, \mathtt{quote}\}$, and $\mathtt{catched}(Par) \in \mathtt{bool}$.

The parameters are used as the pattern parameters in the pattern matching. If $\mathtt{skind}(Par) = \mathtt{elem}$, the pattern matching associates Par with an element. If $\mathtt{skind}(Par) = \mathtt{seq}$, the pattern matching associates Par with (possibly empty) an element sequence. The function `skind` is called a parameter structure specifier.

The function `catched` specifies the propagation of the indeterminate value `false` in the definition of the function `val` (see below). The element `false` plays the role of both the boolean and the indeterminate value.

The element sequences associated with parameters are converted to the parameter values and used as arguments of the functions defining the static semantics of these element sequences. Let Par be associated with $ElSeq$. If $\mathtt{vkind}(Par) = \mathtt{eval}$, Par is called an evaluated parameter, and $\mathtt{val}(Elseq, St)$ is the value of Par. If $\mathtt{vkind}(Par) = \mathtt{quote}$, Par is called a quoted parameter, and $Elseq$ is the value of Par. The function `vkind` is called a parameter value specifier.

The set $\mathtt{elgen}(Dsts)$ of element generators is defined by the functions `sem` and `embedded` such that $\mathtt{sem}(Elgen)$ is a function, $\mathtt{range}(\mathtt{sem}(Elgen)) \subseteq \mathtt{el}$, $\mathtt{embedded}(Elgen) \in \mathtt{nat} \to \mathtt{bool}$, $\mathtt{dom}(\mathtt{embedded}(Elgen)) \subseteq \{I \in \mathtt{nat} \mid 1 \leq I \leq \mathtt{arity}(\mathtt{sem}(Elgen))\}$, and if $Arg \in \mathtt{sem}(Elgen)$, $1 \leq I \leq \mathtt{arity}(\mathtt{sem}(Elgen))$, and $\mathtt{embedded}(Elgen)(Arg.I) = \mathtt{true}$, then $Arg.I \in \mathtt{el}^*$.

The element generators are used to generate new kinds of elements. The element El is generated by $Elgen$, if $El = \texttt{sem}(Elgen)(Arg)$ for some $Arg \in \texttt{dom}(\texttt{sem}(Elgen))$.

The element generators also define the embedded structure of element sequences. The element El' appears in El, if $El' = El$, or $El = \texttt{sem}(Elgen)(Arg)$, and El' appears in $Arg.I$ for some $1 \leq I \leq \texttt{len}(Arg)$ such that $\texttt{embedded}(Elgen)(I) = \texttt{true}$.

The set `elgen` includes the object `seqcomp` such that $\texttt{sem}(\texttt{seqcomp}) \in \texttt{el}^* \to_t \texttt{el}$ is a bijection, and $\texttt{embedded}(\texttt{seqcomp})(1) = \texttt{true}$. The object `seqcomp` is called a sequential element composition.

For simplicity, below we write $Elgen$ instead of $\texttt{sem}(Elgen)$. For example, we write `seqcomp` instead of `sem(seqcomp)`.

The elements of the set $\texttt{sub} = \texttt{el} \to \texttt{el}^*$ are called substitutions. If $\texttt{dom}(Sub) = \{X_1, ..., X_n\}$, Sub can be written as $\{(X_1 \leftarrow Sub(X_1)),\ ...,\ (X_n \leftarrow Sub(X_n))\}$. The substitution function $\texttt{subst} \in \texttt{el}^* \times \texttt{sub} \to \texttt{el}^*$ is defined as follows (the first proper rule is applied):

- $\texttt{subst}(\texttt{empseq}, Sub) = \texttt{empseq}$;
- if $El \in \texttt{dom}(Sub)$, then $\texttt{subst}(El, Sub) = Sub(El)$;
- $\texttt{subst}(\texttt{sem}(Elgen)(Arg), Sub) = \texttt{sem}(Elgen)(Arg')$;
- $\texttt{subst}(El, Sub) = El$;
- $\texttt{subst}(El\ Elseq, Sub) = \texttt{subst}(El, Sub)\ \texttt{subst}(Elseq, Sub)$.

The sequence Arg' is defined as follows:

- if $\texttt{embedded}(Elgen)(I) = \texttt{true}$, then $Arg'.I = \texttt{subst}(Arg, Sub)$;
- if $\texttt{embedded}(Elgen)(I) \neq \texttt{true}$, then $Arg'.I = Arg$.

Substitutions are used to associate parameters with the element sequences as a result of the pattern matching, and to associate parameters with their values.

The set `frm` of forms is defined by the functions `pat`, `pars`, `pcond`, `rvcond`, and `kind`.

A form Frm defines the static and dynamic semantics for the set of element sequences called the instances of the form. The pattern matching uses the functions `pat`, `pars`, and `pcond` to define whether $Elseq$ is an instance of Frm.

The sequence $\texttt{pat}(Frm) \in \texttt{el}^+$ is called a pattern of Frm.

The sequence $\texttt{pars}(Frm) \in \texttt{par}^*$ such that the elements of $\texttt{pars}(Frm)$ are pairwise distinct defines the parameters of the pattern $\texttt{pat}(Frm)$. Let $1 \leq I \leq \texttt{len}(\texttt{pars}(Frm))$. The element $\texttt{par}(Frm).I$ is called a parameter of Frm. The number $\texttt{len}(\texttt{pars}(Frm))$ is called the arity of Frm denoted by $\texttt{arity}(Frm)$.

A form defines the static semantics of its instances by its value. The value of Frm is defined as a function of the values of its parameters.

The element $\mathtt{pcond}(Frm) \in \mathtt{el}$ is called a parameter condition of Frm. It defines a restriction on the values of the parameters of Frm. The sequence $Elseq$ is an instance of Frm only if this restriction is satisfied.

The element $\mathtt{rvcond}(Frm) \in \mathtt{el}$ is called a return-value condition of Frm. It defines a restriction on the value of Frm. The condition $\mathtt{rvcond}(Frm)$ can include the parameters of Frm and the element $\mathtt{retval} \in \mathtt{el}$, where $\mathtt{retval} \in \mathtt{elgen}$, which refers to the value of Frm.

The partial function $\mathtt{kind}(Frm) \in \{\mathtt{statedependent}, \mathtt{statefree}, \mathtt{defined}\}$ defines the kind of Frm. Thus, forms are divided into four kinds (the fourth kind corresponds to $\mathtt{kind}(Frm) = \mathtt{und}$), and each kind has its own semantics.

The forms of the fourth kind define the state of $Dsts$. The function $St \in \{Frm \mid \mathtt{kind}(Frm) = \mathtt{und}\} \to \cup_{n \in \mathtt{nat0}}(\mathtt{el}^n \to_t \mathtt{el})$ such that $St(Frm) \in \mathtt{el}^{\mathtt{arity}(Frm)} \to_t \mathtt{el}$ for all Frm is called the state of $Dsts$. Let $\mathtt{st}$ be the set of all states of $Dsts$. The state St is called empty, if $\mathtt{dom}(St) = \emptyset$. The element $St(Frm)$ is called the value of Frm in St.

The form Frm of the kind $\mathtt{statedependent}$ is called a state-dependent predefined form. It is additionally defined by the function $\mathtt{frmsem}$ such that $\mathtt{frmsem}(Frm) \in \mathtt{st} \to \cup_{N \in \mathtt{nat0}}(\mathtt{el}^N \to_t \mathtt{el})$, and $\mathtt{frmsem}(Frm)(St) \in \mathtt{el}^{\mathtt{arity}(Frm)} \to_t \mathtt{el}$. The function $\mathtt{frmsem}$ is called a form semantics. The element $\mathtt{frmsem}(Frm)(St)$ is called the value of Frm in St.

The form Frm of the kind $\mathtt{statefree}$ is called a state-free predefined form. It is additionally defined by the function $\mathtt{frmsem}$ such that $\mathtt{frmsem}(Frm) \in \cup_{N \in \mathtt{nat0}}(\mathtt{el}^N \to_t \mathtt{el})$, and $\mathtt{frmsem}(Frm) \in \mathtt{el}^{arity(Frm)} \to_t \mathtt{el}$. The function $\mathtt{frmsem}$ is called a form semantics. The element $\mathtt{frmsem}(Frm)$ is called the value of Frm in St.

The form Frm of the kind $\mathtt{defined}$ is called a defined form. It is additionally defined by the function $\mathtt{body}$ such that $\mathtt{body}(Frm) \in \mathtt{el}^+$, which specifies the value of Frm. The elements of $\mathtt{body}(Frm)$ can include the parameters of Frm. Let Sub' map the parameters of Frm onto their values. The value of Frm in St is a function which maps the values of parameters of Frm, represented by Sub', onto $\mathtt{val}(\mathtt{subst}(\mathtt{body}(Frm), Sub'), St)$. The function $\mathtt{val}$ is defined below.

The function $\mathtt{match}(Dsts) \in \mathtt{el}^+ \to \mathtt{pset}(\mathtt{frm} \times \mathtt{sub})$ is called a form matching if for all $(Frm, Sub) \in \mathtt{match}(Elseq)$ the following properties are satisfied:

- $Elseq = \mathtt{subst}(Frm, Sub)$;
- $\mathtt{dom}(Sub)$ is the set of parameters of Frm;
- if $\mathtt{skind}(Par) = \mathtt{elem}$, then $Sub(Par) \in \mathtt{el}$;

- if $\mathtt{skind}(Par) = \mathtt{seq}$, then $Sub(Par) \in \mathtt{el}^*$;
- $\mathtt{arity}(Frm) = \mathtt{arity}(Frm')$, and $Sub(\mathtt{pars}(Frm).I) = Sub'(\mathtt{pars}(Frm').I)$ for all $(Frm', Sub') \in \mathtt{match}(Elseq)$, and $1 \leq I \leq \mathtt{arity}(Frm)$.

The sequence $Elseq$ is called an instance of Frm w.r.t. Sub, if $(Frm, Sub) \in \mathtt{match}(Dsts)(Elseq)$ for some Sub. The sequence $Elseq$ is called an instance of Frm, if $Elseq$ is an instance of Frm w.r.t. some Sub.

The function $\mathtt{match}(Dsts) \in \mathtt{el}^+ \times \mathtt{st} \to \mathtt{frm} \times \mathtt{sub} \times \mathtt{sub}$ is called a form matching with parameter meaning if $\mathtt{match}(Elseq, St) = (Frm, Sub, Sub')$ if and only if the following properties are satisfied:

- $(Frm, Sub) \in \mathtt{match}(Dsts)(Elseq)$;
- $Sub' = \mathtt{parval}(\mathtt{pars}(Frm), Sub, St)$;
- $\mathtt{val}(\mathtt{subst}(\mathtt{pcond}(Frm), Sub'), St) = \mathtt{true}$.

It matches the form and the element sequence and sets the values of the parameters of this form. The function `parval` that sets the values of the parameters of the matched form is defined below.

The sequence $Elseq$ is called an instance of Frm in St w.r.t. the matching substitution Sub and the parameter meaning Sub', if $\mathtt{match}(Elseq, St) = (Frm, Sub, Sub')$. The sequence $Elseq$ is called an instance of Frm in St, if $Elseq$ is an instance of Frm in St w.r.t. some Sub and Sub'.

The set $\mathtt{elgen}(Dsts)$ includes the functions $\mathtt{quote} \in \mathtt{el}^+ \to \mathtt{el}$, and $\mathtt{eval} \in \mathtt{el}^+ \to \mathtt{el}$ such that $\mathtt{embedded}(\mathtt{quote}, 1) = \mathtt{embedded}(\mathtt{eval}, 1) = true$. They specify the value of Par in the case when $Sub(Par)$ has the form $\mathtt{eval}(Elseq)$ or $\mathtt{quote}(Elseq)$. If $Sub(Par) = \mathtt{eval}(Elseq)$, then $Sub'(Par) = \mathtt{val}(Elseq, St)$. If $Sub(Par) = \mathtt{quote}(Elseq)$, then $Sub'(Par) = Elseq$.

The function $\mathtt{parval} \in \mathtt{par}^* \times \mathtt{sub} \times \mathtt{st} \to \mathtt{sub}$ sets the values of parameters in accordance with the element sequences which match these parameters:

- if $\mathtt{sub}(Par) = \mathtt{eval}(Elseq)$, then $\mathtt{parval}(Par\ Parseq, Sub, St) = \{(Par \leftarrow \mathtt{val}(Elseq, St))\} \cup \mathtt{parval}(Parseq, Sub, St)$;
- if $\mathtt{sub}(Par) = \mathtt{quote}(Elseq)$, then $\mathtt{parval}(Par\ Parseq, Sub, St) = \{(Par \leftarrow Elseq)\} \cup \mathtt{parval}(Parseq, Sub, St)$;
- if $\mathtt{vkind}(Par) = \mathtt{eval}$, and $\mathtt{skind}(Par) = \mathtt{elem}$, then $\mathtt{parval}(Par\ Parseq, Sub, St) = \{(Par \leftarrow \mathtt{val}(Sub(Par), St))\} \cup \mathtt{parval}(Parseq, Sub, St)$;
- if $\mathtt{vkind}(Par) = \mathtt{eval}$, $\mathtt{skind}(Par) = \mathtt{seq}$, and $Sub(Par) = Elorpar_1\ \ldots\ Elorpar_N$, then $\mathtt{parval}(Par\ Parseq, Sub, St) = \{(Par \leftarrow \mathtt{ifval}(Elorpar_1, \mathtt{eval}, St)\ \ldots\ \mathtt{ifval}(Elorpar_N, \mathtt{eval}, St))\} \cup \mathtt{parval}(Parseq, Sub, St)$;

- if $\mathtt{vkind}(Par) = \mathtt{quote}$, and $\mathtt{skind}(Par) = \mathtt{elem}$, then $\mathtt{parval}(Par\ Parseq, Sub, St) = \{(Par \leftarrow Sub(Par))\} \cup \mathtt{parval}(Parseq, Sub, St)$;
- if $\mathtt{vkind}(Par) = \mathtt{quote}$, $\mathtt{skind}(Par) = \mathtt{seq}$, and $Sub(Par) = El_1\ \ldots\ E_N$, then $\mathtt{parval}(Par\ Parseq, Sub, St) = \{(Par \leftarrow \mathtt{ifval}(El_1, \mathtt{quote}, St)\ \ldots\ \mathtt{ifval}(El_N, \mathtt{quote}, St))\} \cup \mathtt{parval}(Parseq, Sub, St)$.

The function $\mathtt{ifval} \in \mathtt{el}^+ \times \mathtt{st} \times \{\mathtt{eval}, \mathtt{quote}\} \to \mathtt{el}$ is defined as follows:

- $\mathtt{ifval}(\mathtt{eval}(Elseq), St, Vparorqpar) = \mathtt{val}(Elseq, St)$;
- $\mathtt{ifval}(\mathtt{quote}(Elseq), St, Vparorqpar) = Elseq$;
- $\mathtt{ifval}(El, St, \mathtt{eval}) = \mathtt{val}(El, St)$;
- $\mathtt{ifval}(El, St, \mathtt{quote}) = El$.

The function $\mathtt{val} \in \mathtt{el}^+ \times \mathtt{st} \to \mathtt{el}$ called an element sequence meaning is defined as follows (the first proper rule is applied):

- $val(\mathtt{true}, St) = \mathtt{true}$;
- if $\mathtt{match}(Elseq) = (Frm, Sub, Sub')$, $\mathtt{pars}(Frm) = Par_1\ \ldots\ Par_N$, $Arg = Sub'(Par_1), \ldots, Sub'(Par_N)$, and $\mathtt{Retvalcond}(U)$ denotes

  $$\mathtt{val}(\mathtt{subst}(\mathtt{rvcond}(Frm), Sub' \cup \{(\mathtt{retval}(Dsts) \leftarrow U)\}), St) = \mathtt{true},$$

  then

  - if $\mathtt{catched}(Par_I) \neq \mathtt{true}$, and $Sub'(Par_I) = \mathtt{false}$ for some $1 \leq I \leq \mathtt{arity}(Frm)$, then $\mathtt{val}(Elseq, St) = \mathtt{false}$;
  - if $\mathtt{kind}(Frm) = \mathtt{und}$, $St(Frm) \neq \mathtt{und}$, and $\mathtt{Retvalcond}(St(Frm)(Arg))$, then $\mathtt{val}(Elseq, St) = St(Frm)(Arg)$;
  - if $\mathtt{kind}(Frm) = \mathtt{statedependent}$, and $\mathtt{Retvalcond}(\mathtt{frmsem}(Frm)(St)(Arg))$, then $\mathtt{val}(Elseq, St) = \mathtt{frmsem}(Frm)(St)(Arg)$;
  - if $\mathtt{kind}(Frm) = \mathtt{statefree}$, and $\mathtt{Retvalcond}(frmsem(Frm)(Arg))$, then $\mathtt{val}(Elseq, St) = \mathtt{frmsem}(Frm)(Arg)$;
  - if $\mathtt{kind}(Frm) = \mathtt{defined}$, and $\mathtt{Retvalcond}(\mathtt{val}(\mathtt{subst}(\mathtt{body}(Frm), Sub'), St))$, then $\mathtt{val}(Elseq, St) = \mathtt{val}(\mathtt{subst}(\mathtt{body}(Frm), Sub'), St)$;
- if $\mathtt{atom}(Dsts)(Elseq) = \mathtt{true}$, then $\mathtt{val}(Elseq, St) = Elseq$;
- $\mathtt{val}(Elseq, St) = \mathtt{false}$.

The element $\mathtt{val}(Elseq, St)$ is called the value of $Elseq$ in St.

The function $\mathtt{atom}(Dsts) \in \mathtt{el}^+ \to_t \mathtt{bool}$ defines the element sequences which coincide with their values. Such sequences are called atoms.

The dynamic semantics of the element sequences is defined by the function $\mathtt{tr} \in \mathtt{conf} \times \mathtt{conf} \to \mathtt{bool}$ called a transition relation. The set conf of configurations and the function tr are defined below. The system $Dsts$ can go from $Conf$ to $Conf'$ if and only if $\mathtt{tr}(Conf, Conf') = \mathtt{true}$.

The set conf of configurations is defined by the functions seq and st such that $\mathtt{seq}(Conf) \in \mathtt{el}^*$, and $\mathtt{st}(Conf) \in \mathtt{st}$. The sequence $\mathtt{seq}(Conf)$ is called a control sequence of $Conf$. It defines the states to which $Dsts$ can go from the current state and the control sequences executed in these states.

The configuration $Conf$ is called a final one, if there is no configuration $Conf'$ such that $\mathtt{tr}(Conf, Conf') = \mathtt{true}$. The sequence $Confseq \in \mathtt{conf}^+$ is called a run, if $\mathtt{last}(Confseq)$ is a final configuration.

A final configuration $Conf$ is called unsafe, if $\mathtt{seq}(Conf) \neq \mathtt{empseq}$. It specifies incorrect termination of $Dsts$. A final configuration $Conf$ is called safe, if $Conf$ is not unsafe. A run $Confseq$ such that $\mathtt{last}(Confseq)$ is unsafe, is called unsafe. A run $Confseq$ is called safe, if $Confseq$ is not unsafe. A configuration $Conf$ is called unsafe, if there is an unsafe run $Confseq$ such that $\mathtt{first}(Confseq) = Conf$. A configuration $Conf$ is called safe, if $Conf$ is not unsafe.

A sequence $Elseq$ is correct in St, if $Conf$ is safe, where $\mathtt{seq}(Conf) = Elseq$, and $\mathtt{st}(Conf) = St$. A sequence $Elseq$ is incorrect in St, if $Elseq$ is not corrrect in St.

The set elgen includes the function $\mathtt{fail} \in \mathtt{el}$ such that if $\mathtt{first}(\mathtt{seq}(Conf)) = \mathtt{fail}$, then $Conf$ is final. The configuration $Conf$ is also unsafe, since $\mathtt{seq}(Conf) \neq \mathtt{empseq}$. Therefore the element fail is called an unsafe termination.

Let $\mathtt{tr}(Conf, Conf') = \mathtt{true}$. Then $\mathtt{seq}(Conf)$ and $\mathtt{seq}(Conf')$ are called the input and output control sequences of the transition, and $\mathtt{st}(Conf)$ and $\mathtt{st}(Conf')$ are called the input and output states of the transition, respectively.

The function tr is defined by a special kind of forms, or transition rules. A form Frm is called a transition rule, if it is additionally defined by the function rkind such that $\mathtt{rkind}(Frm) \in \{\mathtt{defined}, \mathtt{predefined}\}$. The function kind defines the kind of the rule. Thus, if $\mathtt{rkind}(Frm) = \mathtt{und}$, then Frm is not a transition rule, and transition rules are divided into two kinds, and each kind has its own semantics. Let $\mathtt{rul}(Dsts)$ be a set of all rules of $Dsts$, and $Rul \in \mathtt{rul}(Dsts)$.

A rule Rul of the kind defined is called defined. It is additionally defined by the function body such that $\mathtt{body}(Rul) \in \mathtt{el}^*$. The sequence $\mathtt{body}(Rul)$ is called the body of Rul and it defines the execution of Rul.

A rule Rul of the kind predefined is called predefined. It is additionally defined by the function rulsem such that $\mathtt{rulsem}(Rul) \in \mathtt{conf} \times \mathtt{conf} \times \mathtt{sub} \to_t \mathtt{bool}$. This function is called a rule semantics and it defines the execution of Rul. The third argument of the function stores the values of the parameters of Rul.

The function tr is defined as follows: $\mathtt{tr}(Conf, Conf') = \mathtt{true}$ if and only if there is a rule Rul such that $\mathtt{tr}(Conf, Conf', Rul) = \mathtt{true}$.

The function $\mathtt{tr}$ with an additional argument Rul is defined as follows: $\mathtt{tr}(Conf, Conf', Rul) = \mathtt{true}$ if and only if $\mathtt{seq}(Conf) = Elseq\ Elseq'$, $\mathtt{match}(Elseq, \mathtt{st}(Conf)) = (Rul, Sub, Sub')$, and one of two conditions is satisfied: $\mathtt{rkind}(Rul) = \mathtt{predefined}$, and $\mathtt{rulsem}(Rul)(Conf, Conf', Sub') = \mathtt{true}$, or $\mathtt{rkind}(Rul) = \mathtt{defined}$, $\mathtt{seq}(Conf') = \mathtt{subst}(\mathtt{body}(Rul), Sub')\ Elseq'$, and $\mathtt{st}(Conf') = \mathtt{st}(Conf)$.

Thus, when a defined rule Rul is applied, the state of $Dsts$ does not change, and the control sequence changes only its prefix matched with Rul.

The configuration $Conf$ is called final w.r.t. Rul, if there is no configuration $Conf'$ such that $\mathtt{tr}(Conf, Conf', Rul) = \mathtt{true}$.

4. Domain-specific transition systems with backtracking

The use of backtracking in DSTSs expands their expressive power.

A DSTS $Dsts$ is called a DSTS with backtracking if the following properties are satisfied:

- $\mathtt{conf}$ is additionally defined by the function $\mathtt{rulset}$ such that $\mathtt{rulset}(Conf) \subseteq \mathtt{rul}(Dsts)$. This function specifies which transition rules have been applied in the transitions from the configuration $Conf$.
- $Dsts$ is additionally defined by the function $\mathtt{backfrm}$ such that $\mathtt{backfrm}(Dsts) \subseteq Frm$. The set $\mathtt{backfrm}(Dsts)$ specifies the forms whose values are preserved when $Dsts$ backtracks to the previous backtracking point.

The function $\mathtt{ifst}(St, St')$ returns a state; it is defined as follows:

- if $Frm \in \mathtt{backfrm}$, then $\mathtt{ifst}(St, St')(Frm) = St'(Frm)$;
- if $Frm \notin \mathtt{backfrm}$, then $\mathtt{ifst}(St, St')(Frm) = St(Frm)$.

DSTS with controlled backtracking. A DSTS $Dsts$ with backtracking is called a DSTS with controlled backtracking, if

- $\mathtt{elgen}(Dsts)$ includes the functions $\mathtt{backtrack}$ and $\mathtt{branch}$ such that $\mathtt{backtrack} \in \mathtt{el}$, $\mathtt{branch} \in \mathtt{el}^{**} \to_t \mathtt{el}$, and $\mathtt{embedded}(\mathtt{branch})(1) = \mathtt{true}$. The element $\mathtt{backtrack}$ called a backtracking condition initiates backtracking to the previous backtracking point. The element $\mathtt{branch}(ElSeqSeq)$ called a branch element is used to define possible variants in the backtracking point given by the elements of $ElSeqSeq$.
- $\mathtt{tr} \in \mathtt{conf}^* \times \mathtt{conf}^* \to \mathtt{bool}$ is a controlled backtracking.

Let $\mathtt{ba}(St)$ and $\mathtt{fi}(St)$ be configurations such that $\mathtt{seq}(\mathtt{ba}(St)) = \mathtt{backtrack}$, $\mathtt{st}(\mathtt{ba}(St)) = St$, $\mathtt{rulset}(\mathtt{ba}(St)) = \emptyset$, $\mathtt{seq}(\mathtt{fi}(St)) = \mathtt{empseq}$, $\mathtt{st}(\mathtt{fi}(St)) = St$, and $\mathtt{rulset}(\mathtt{ba}(St)) = \emptyset$.

A transition relation $\mathtt{tr} \in \mathtt{conf}^* \times \mathtt{conf}^* \to \mathtt{bool}$ is called a controlled backtracking, if $\mathtt{tr}(Confseq, Confseq') = \mathtt{true}$ if and only if the first proper property is satisfied:

- $Confseq = Confseq''\ Conf$, $\mathtt{seq}(Conf) = \mathtt{backtrack}\ Elseq$, where $Elseq \neq \mathtt{empseq}$, or $\mathtt{rulset}(Conf) \neq \emptyset$, and $Confseq' = Confseq''\ \mathtt{ba}(\mathtt{st}(Conf))$;
- $Confseq = Confseq''\ Conf\ \mathtt{ba}(St)$, where $\mathtt{seq}(Conf) = \mathtt{branch}(Elseq'\ Elseqseq)\ Elseq$, $Confseq' = Confseq''\ Conf'\ Conf''$, $\mathtt{seq}(Conf') = \mathtt{branch}(Elseqseq)\ Elseq$, $\mathtt{st}(Conf') = \mathtt{st}(Conf)$, $\mathtt{rulset}(Conf') = \mathtt{rulset}(Conf)$, $\mathtt{seq}(Conf'') = Elseq'\ Elseq$, $\mathtt{st}(Conf'') = \mathtt{ifst}(\mathtt{st}(Conf), St)$, and $\mathtt{rulset}(Conf'') = \emptyset$;
- $Confseq = Confseq''\ Conf'\ Conf\ \mathtt{ba}(St)$, where $\mathtt{seq}(Conf) = \mathtt{branch}(\mathtt{empseq})\ Elseq$, and $Confseq' = Confseq''\ Conf'\ \mathtt{ba}(St)$;
- $Confseq = Conf\ \mathtt{ba}(St)$, where $\mathtt{seq}(Conf') = \mathtt{branch}(\mathtt{empseq})\ Elseq$, and $Confseq' = \mathtt{ba}(\mathtt{ifst}(\mathtt{st}(Conf), St))$;
- $Confseq = Confseq''\ Conf$, where $\mathtt{seq}(Conf) = \mathtt{branch}(Elseq'\ Elseqseq)\ Elseq$, $Confseq' = Confseq''\ Conf'\ Conf''$, $\mathtt{seq}(Conf') = \mathtt{branch}(Elseqseq)\ Elseq$, $\mathtt{st}(Conf') = \mathtt{st}(Conf)$, $\mathtt{rulset}(Conf') = \mathtt{rulset}(Conf)$, $\mathtt{seq}(Conf'') = Elseq'\ Elseq$, $\mathtt{st}(Conf'') = \mathtt{st}(Conf)$, and $\mathtt{rulset}(Conf'') = \emptyset$;
- $Confseq = Confseq''\ Conf$, where $\mathtt{seq}(Conf) = \mathtt{branch}(\mathtt{empseq})\ Elseq$, and $Confseq' = Confseq''\ \mathtt{fi}(\mathtt{st}(Conf))$;
- $Confseq = Confseq''\ Conf\ \mathtt{ba}(St)$, where $\mathtt{seq}(Conf) = Elseq^0\ Elseq$, $Elseq^0 \notin \mathtt{predel}(Dsts)$, $Rul \in \mathtt{rul}(Dsts) \setminus \mathtt{rulset}(Conf)$, $\mathtt{tr}(Conf^2, Conf^3, Rul) = \mathtt{true}$, $\mathtt{seq}(Conf^2) = Elseq^0\ Elseq$, $\mathtt{st}(Conf^2) = \mathtt{ifst}(\mathtt{st}(Conf), St)$, $\mathtt{seq}(Conf^3) = Elseq'$, $Confseq' = Confseq''\ Conf^4\ Conf^5$, $\mathtt{seq}(Conf^4) = Elseq^0\ Elseq$, $\mathtt{st}(Conf^4) = \mathtt{st}(Conf)$, $\mathtt{rulset}(Conf^4) = \mathtt{rulset}(Conf) \cup \{Rul\}$, $\mathtt{seq}(Conf^5) = Elseq'$, $\mathtt{st}(Conf^5) = \mathtt{st}(Conf^3)$, and $\mathtt{rulset}(Conf^5) = \emptyset$;
- $Confseq = Confseq''\ Conf'\ Conf\ \mathtt{ba}(St)$, and $Confseq' = Confseq''\ Conf'\ \mathtt{ba}(St)$;
- $Confseq = Conf\ \mathtt{ba}(St)$, and $Confseq' = \mathtt{ba}(\mathtt{ifst}(\mathtt{st}(Conf), St))$;
- $Confseq = \mathtt{ba}(St)$, and $Confseq' = \mathtt{fi}(St)$;
- $Confseq = Confseq''\ Conf$, where $\mathtt{seq}(Conf) = Elseq^0\ Elseq$, $Elseq^0 \notin \mathtt{predel}$, $Rul \in \mathtt{rul}(Dsts) \setminus \mathtt{rulset}(Conf)$, $\mathtt{tr}(Conf, Conf'$, $Rul) = \mathtt{true}$, $Confseq' = Confseq''\ Conf''\ Conf'''$, $\mathtt{seq}(Conf'') =$

$\mathtt{seq}(Conf)$, $\mathtt{st}(Conf'') = \mathtt{st}(Conf)$, $\mathtt{rulset}(Conf'') = \mathtt{rulset}(Conf) \cup \{Rul\}$, $\mathtt{seq}(Conf''') = \mathtt{seq}(Conf')$, $\mathtt{st}(Conf''') = \mathtt{st}(Conf')$, an $\mathtt{rulset}(Conf''') = \emptyset$;

- $Confseq = Confseq''\ Conf$, where $\mathtt{seq}(Conf) = Elseq^0\ Elseq$, $Elseq^0 \in \mathtt{predel}(Dsts)$, $\mathtt{rulsem}(Dsts)(Conf, Conf') = \mathtt{true}$, $Confseq' = Confseq''\ Conf''\ Conf'''$, $\mathtt{seq}(Conf'') = Elseq$, $\mathtt{st}(Conf'') = \mathtt{st}(Conf)$, $\mathtt{rulset}(Conf'') = \mathtt{rulset}(Conf)$, $\mathtt{seq}(Conf''') = \mathtt{seq}(Conf')$, $\mathtt{st}(Conf''') = \mathtt{st}(Conf')$, and $\mathtt{rulset}(Conf''') = \emptyset$;
- $\mathtt{false}$.

5. Examples of predefined transition rules

Let us consider examples of predefined transition rules which are often used to define operational semantics of computer languages.

Let $\mathtt{elgen}(Dsts)$ include the function $\mathtt{stop} \in el$. A form Rul is called a stop rule, if $\mathtt{pat}(Rul) = \mathtt{stop}$, $\mathtt{arity}(Rul) = 0$, $\mathtt{cond}(Rul) = \mathtt{true}$, $\mathtt{rkind}(Rul) = \mathtt{predefined}$, and $\mathtt{rulsem}(Rul)(Conf, Conf') = \mathtt{true}$, where $\mathtt{seq}(Conf) = \mathtt{stop}\ Elseq$, if and only if $\mathtt{seq}(Conf') = \mathtt{empseq}$, and $\mathtt{st}(Conf') = \mathtt{st}(Conf)$. The element $\mathtt{stop}$ is called a stop element.

Let $\mathtt{elgen}(Dsts)$ include the function $\mathtt{assume} \in \mathtt{el} \to \mathtt{el}$. A form Rul is called a continuation rule, if $\mathtt{pat}(Rul) = \mathtt{assume}(Par)$, $\mathtt{pars}(Rul) = Par$, $\mathtt{vkind}(Par) = \mathtt{eval}$, $\mathtt{skind}(Par) = \mathtt{seq}$, $\mathtt{cond}(Rul) = \mathtt{true}$, $\mathtt{rkind}(Rul) = \mathtt{predefined}$, and $\mathtt{rulsem}(Rul)(Conf, Conf') = \mathtt{true}$, where $\mathtt{seq}(Conf) = El\ Elseq$, if and only if $\mathtt{match}(El, \mathtt{st}(Conf)) = (Rul, Sub, Sub')$, $\mathtt{st}(Conf') = \mathtt{st}(Conf)$, and the first proper property is satisfied:

- if $Sub'(Par) = \mathtt{true}$, then $\mathtt{seq}(Conf') = Elseq$;
- $\mathtt{seq}(Conf') = \mathtt{backtrack}\ Elseq$.

The element El is called a continuation condition. This condition is based on the element $\mathtt{backtrack}$ and used in DSTS with controlled backtracking.

Let $\mathtt{elgen}(Dsts)$ include the function $\mathtt{frmupd} \in \mathtt{el}^+ \times \mathtt{el}^+ \to \mathtt{el}$. A form Rul is called a form update rule, if $\mathtt{pat}(Rul) = \mathtt{frmupd}(Par, Par')$, $\mathtt{pars}(Rul) = Par\ Par'$, $\mathtt{vkind}(Par) = \mathtt{quote}$, $\mathtt{vkind}(Par') = \mathtt{eval}$, $\mathtt{skind}(Par) = \mathtt{skind}(Par') = \mathtt{seq}$, $\mathtt{cond}(Rul) = \mathtt{true}$, $\mathtt{rkind}(Rul) = \mathtt{predefined}$, and $\mathtt{rulsem}(Rul)(Conf, Conf') = \mathtt{true}$, where $\mathtt{seq}(Conf) = El\ Elseq$, if and only if $\mathtt{match}(El, \mathtt{st}(Conf)) = (Rul, Sub, Sub')$, and the first proper property is satisfied:

- if $\mathtt{match}(Sub'(Par), \mathtt{st}(Conf)) = (Frm, Sub_1, Sub'_1)$, $\mathtt{kind}(Frm) = \mathtt{und}$, $\mathtt{arity}(Frm) = N$, $Arg = Sub'_1(\mathtt{pars}(Frm).1), \ldots, Sub'_1(\mathtt{pars}(Frm).N)$, then $\mathtt{seq}(Conf') = Elseq$, $\mathtt{odif}(\mathtt{st}(Conf'),$

$\mathtt{st}(Conf), \{Frm\}) = \mathtt{true}$, $\mathtt{odif}(\mathtt{st}(Conf')(Frm), \mathtt{st}(Conf)(Frm),$ $\{(Arg)\}) = \mathtt{true}$, and $\mathtt{st}(Conf')(Frm)(Arg) = Sub'(Par')$;

- $\mathtt{seq}(Conf') = \mathtt{fail}\ El\ Elseq$, and $\mathtt{st}(Conf') = \mathtt{st}(Conf)$.

The element El is called a form update.

Let $\mathtt{elgen}(Dsts)$ include the function $\mathtt{assert} \in \mathtt{el}^+ \to \mathtt{el}$. A form Rul is called a safety rule, if $\mathtt{pat}(Rul) = \mathtt{assert}(Par)$, $\mathtt{pars}(Rul) = Par$, $\mathtt{vkind}(Par) = \mathtt{eval}$, $\mathtt{skind}(Par) = \mathtt{seq}$, $\mathtt{cond}(Rul) = \mathtt{true}$, $\mathtt{rkind}(Rul) = \mathtt{predefined}$, $\mathtt{rulsem}(Rul)(Conf, Conf') = \mathtt{true}$, where $\mathtt{seq}(Conf) = El\ Elseq$, if and only if $\mathtt{match}(El, \mathtt{st}(Conf)) = (Rul, Sub, Sub')$, $\mathtt{st}(Conf') = \mathtt{st}(Conf)$, and the first proper property is satisfied:

- if $Sub'(Par) = \mathtt{true}$, then $\mathtt{seq}(Conf') = Elseq$;
- $\mathtt{seq}(Conf') = \mathtt{fail}\ El\ Elseq$.

The element El is called a safety condition.

Let $\mathtt{elgen}(Dsts)$ include the function $\mathtt{cases} \in \mathtt{par} \times \mathtt{par} \cup \mathtt{par} \times \mathtt{par} \times \mathtt{par} \cup \mathtt{el}^* \times \mathtt{el}^{**} \cup \mathtt{el}^* \times \mathtt{el}^{**} \times \mathtt{el}^* \to \mathtt{el}$, such that $(Elseq, Elseqseq[, Elseq']) \in \mathtt{dom}(\mathtt{cases})$ if and only if $\mathtt{len}(Elseq) = \mathtt{len}(Elseqseq)$.

A form Rul is called a conditional branching rule, if $\mathtt{pat}(Rul) = \mathtt{cases}(Par, Par'[, Par''])$, $\mathtt{pars}(Rul) = Par\ Par'\ Par''$, $\mathtt{vkind}(Par) = \mathtt{quote}$, and $\mathtt{skind}(Par) = \mathtt{seq}$ for each $Par \in \mathtt{pars}(Rul)$, $\mathtt{cond}(Rul) = \mathtt{true}$, $\mathtt{rkind}(Rul) = \mathtt{predefined}$, $\mathtt{rulsem}(Rul)(Conf, Conf') = \mathtt{true}$, where $\mathtt{seq}(Conf) = El\ Elseq$, if and only if $\mathtt{match}(El, \mathtt{st}(Conf)) = (Rul, Sub, Sub')$, $\mathtt{st}(Conf') = \mathtt{st}(Conf)$, and $\mathtt{seq}(Conf') = \mathtt{branch}(Arg.1, \ldots, Arg.N\ [, Sub'(Par'')])$, where $Arg.I = \mathtt{assume}(Sub'(Par).1)\ Sub'(Par').1$ for $1 \le I \le \mathtt{len}(Sub'(Par)) = N$. The element El is called a conditional branching.

6. Formal definition of the model programming language

Let us define a simple model programming language MPL by DSTS.

The MPL language includes the set id of identifiers (sequences of letters from $\{\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots, \mathtt{Z}\}$, digits from $\{0, \ldots, 9\}$, and the underscore character _, starting with a letter), the finite set $\mathtt{btype} \subset \mathtt{id}$ of basic types such that $\mathtt{lit}(Btype)$ is a set of literals of the type $Btype \in \mathtt{btype}$, $\mathtt{lit}(Btype) \cap \mathtt{id} = \emptyset$ for each $Btype$, $\mathtt{int} \in \mathtt{btype}$, $\mathtt{lit}(\mathtt{int}) = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, $\mathtt{bool} \in \mathtt{btype}$, and $\mathtt{lit}(\mathtt{bool}) = \{\mathtt{true}, \mathtt{false}\}$, the operations $=$, and $! =$ on these types, the arithmetic operations $+$, $-$, $*$, div, mod, and the arithmetic relations $<$, $>$, $<=$, $>=$ on integers, the boolean operations and, or, not, implies, variable declaration, assignment statement, if statement, and while statement.

Let us consider $Dsts$ which specifies MPL. The functions el, par, elgen, frm, match, and atom are defined as follows:

- $\mathtt{el} \stackrel{def}{=} \mathtt{id} \cup \cup_{Elgen \in \mathtt{elgen}(Dsts)} \mathtt{range}(Elgen)$;
- $\mathtt{par}(Dsts)(El) = \mathtt{true}$ if and only if $El \in \mathtt{id} \setminus \mathtt{dtype}$;
- $\mathtt{elgen}(Dsts) \stackrel{def}{=} \{\mathtt{delcomp}, \mathtt{eval}, \mathtt{quote}, \mathtt{fail}, \mathtt{backtrack}, \mathtt{branch}, \mathtt{stop}, \mathtt{assume}, \mathtt{frmupd}, \mathtt{assert}, \mathtt{cases}\}$;
- if $\mathtt{len}(Elseq) = N$, then $\mathtt{delcomp}(Elseq) \stackrel{def}{=} (Elseq.1 \ ... \ Elseq.N)$;
- $\mathtt{eval}(Elseq) \stackrel{def}{=} (\mathtt{eval}\ Elseq)$;
- $\mathtt{quote}(Elseq) \stackrel{def}{=} (\mathtt{quote}\ Elseq)$;
- $\mathtt{retval}(Dsts) \stackrel{def}{=} \mathtt{retval}$;
- $\mathtt{fail} \stackrel{def}{=} \mathtt{fail}$;
- $\mathtt{backtrack} \stackrel{def}{=} \mathtt{backtrack}$;
- $\mathtt{branch}(Elseqseq) \stackrel{def}{=} (\mathtt{branch}\ Elseqseq)$;
- $\mathtt{stop} \stackrel{def}{=} \mathtt{stop}$;
- $\mathtt{assume}(Elseq) \stackrel{def}{=} (\mathtt{assume}\ Elseq)$;
- $\mathtt{frmupd}(Elseq, Elseq') \stackrel{def}{=} (Elseq \ ::= \ Elseq')$;
- $\mathtt{assert}(Elseq) \stackrel{def}{=} (\mathtt{assert}\ Elseq)$;
- if $\mathtt{len}(Elseq) = N$, then $\mathtt{cases}(Elseq, Elseqseq, Elseq') \stackrel{def}{=} (\mathtt{cases}\ (\mathtt{if}\ Elseq.1\ \mathtt{then}\ Elseqseq.1) \ \ldots (\mathtt{if}\ Elseq.N\ \mathtt{then}\ Elseqseq.N)(\mathtt{else}\ Elseq'))$;
- $\mathtt{frm}(Dsts) \stackrel{def}{=} \{Frm\_Id \mid Id \in \mathtt{id}\} \cup \{Rul\_Id \mid Id \in \mathtt{id}\}$;
- the algorithm $\mathtt{match}(Dsts)$ choose the first proper element sequence from left to right. For example, if $\mathtt{pat}(Frm) = (\mathtt{if}\ X\ \mathtt{then}\ Y\ \mathtt{else}\ Z)$, $\mathtt{pars}(Frm) = X\ Y\ Z$, $\mathtt{skind}(X) = \mathtt{elem}$, $\mathtt{skind}(Y) = \mathtt{seq}$, and $\mathtt{skind}(Z) = \mathtt{seq}$, then $(Frm, (X \leftarrow A, Y \leftarrow B, Z \leftarrow C\ \mathtt{else}\ D)) \in \mathtt{match}(Dsts)((\mathtt{if}\ A\ \mathtt{then}\ B\ \mathtt{else}\ C\ \mathtt{else}\ D))$, and $(Frm, (X \leftarrow A, Y \leftarrow B\ else\ C, Z \leftarrow D)) \notin \mathtt{match}(Dsts)((\mathtt{if}\ A\ \mathtt{then}\ B\ \mathtt{else}\ C\ \mathtt{else}\ D))$;
- $\mathtt{atom}(Dsts)(El) = \mathtt{true}$ if and only if $El \in \mathtt{id} \cup \bigcup_{Btype \in \mathtt{btype}} \mathtt{lit}(Btype)$.

The form `Frm_bool` is associated with the type `bool`, and it is defined as follows: $\mathtt{pat}(\mathtt{Frm\_bool}) = (X\ \mathtt{isof}\ \mathtt{bool})$, $\mathtt{pars}(\mathtt{Frm\_bool}) = X$, $\mathtt{vkind}(X) = \mathtt{eval}$, $\mathtt{skind}(X) = \mathtt{elem}$, $\mathtt{pcond}(\mathtt{Frm\_bool}) = \mathtt{true}$, $\mathtt{rvcond}(\mathtt{Frm\_bool}) = \mathtt{true}$, $\mathtt{kind}(\mathtt{Frm\_bool}) = \mathtt{statefree}$, and $\mathtt{frmsem}(\mathtt{Frm\_bool})(El) = \mathtt{true}$ if and only if $El \in \mathtt{lit}(\mathtt{bool})$.

The form Frm_sort checks whether an element sequence belongs to a particular sort, and it is defined as follows: pat(Frm_sort) = (X is Y), pars(Frm_sort) = X, vkind(X) = quote, vkind(y) = eval, skind(X) = skind(Y) = seq, pcond(Frm_sort) = true, rvcond(Frm_sort) = ((retval) isof bool), kind(Frm_sort) = statefree, and frmsem(Frm_sort)(El) = true if and only if (X isof Y). The forms with the pattern (X isof $Elseq$) for particular sorts $Elseq$ are defined below.

The form Frm_id specifies the characteristic function for id, and it is defined as follows: pat(Frm_id) = (X isof identifier), pars(Frm_id) = X, vkind(X) = quote, skind(X) = elem, pcond(Frm_id) = true, rvcond(Frm_bool) = ((retval) is bool), kind(Frm_id) = statefree, and frmsem(Frm_id)(El) = true if and only if $El \in$ id.

The form Frm_btype specifies the characteristic function for btype, and it is defined as follows: pat(Frm_btype) = (X isof btype), pars(Frm_btype) = X, vkind(X) = quote, skind(X) = elem, pcond(Frm_btype) = (X is identifier), rvcond(Frm_btype) = ((retval) is bool), kind(Frm_btype) = statefree, and frmsem(Frm_btype)(El) = true if and only if $El \in$ btype.

The form Frm_Btype specifies the characteristic function for $Btype \neq$ bool, and it is defined as follows: pat(Frm_Btype) = (X isof $Btype$), pars(Frm_Btype) = X, vkind(X) = eval, skind(X) = elem, pcond(Frm_Btype) = true, rvcond(Frm_Btype) = ((retval) is bool), kind(Frm_Btype) = statefree, and frmsem(Frm_Btype)(El) = true if and only if $El \in$ lit($Btype$).

The operations =, and != on basic types, the arithmetic operations $+$, $-$, $*$, div, mod, and arithmetic relations $<$, $>$, $<=$, $>=$ on integers, the boolean operations and, or, not, $\Longrightarrow$ are defined by the corresponding state-free predefined forms Frm_eq, Frm_neq, Frm_add, Frm_sub, Frm_mul, Frm_div, Frm_mod, Frm_less, Frm_more, Frm_lesseq, Frm_moreeq, Frm_and, Frm_or, Frm_not, and Frm_implies:

- pat(Frm_eq) = (X = Y), pars(Frm_eq) = X Y, vkind(X) = vkind(Y) = eval, skind(X) = skind(Y) = elem, pcond(Frm_eq) = true, rvcond(Frm_eq) = ((retval) is bool), kind(Frm_eq) = statefree, and frmsem(Frm_eq)(El, El') = true if and only if $El \in Btype$, $El' \in Btype$ for some $Btype$, and $El = El'$;
- pat(Frm_add) = (X + Y), pars(Frm_add) = X Y, vkind(X) = vkind(Y) = eval, skind(X) = skind(Y) = elem, pcond(Frm_add) = ((X is int) and (Y is int)), rvcond(Frm_add) = ((retval) is int), kind(Frm_add) = statefree, and frmsem(Frm_add)(El, El') = El'' if and only if $El'' = El + El'$;
- pat(Frm_less) = (X < Y), pars(Frm_less) = X Y, vkind(X) = vkind(Y) = eval, skind(X) = skind(Y) = elem,

pcond(Frm_less) = ((X is int) and (Y is int)), rvcond(Frm_less) = ((retval) is bool), kind(Frm_less) = statefree, and frmsem(Frm_less)(El, El') = true if and only if $El < El'$;

- *pat*(Frm_and) = (X and Y), pars(Frm_and) = X Y, vkind(X) = vkind(Y) = eval, skind(X) = skind(Y) = elem, pcond(Frm_and) = ((X is bool) and (Y is bool)), rvcond(Frm_and) = ((retval) is bool), kind(Frm_and) = statefree, and frmsem(Frm_and)(El, El') = true if and only if El = true, or El' = true.

The other forms are defined in a similar way.

The state St of $Dsts$ is defined by the forms Frm_isvar, Frm_vartype, and Frm_varval.

The form Frm_isvar specifies which identifiers are variables in St and is defined as follows: pat(Frm_isvar) = (X is variable), pars(Frm_isvar) = X, vkind(X) = quote, skind(X) = elem, pcond(Frm_isvar) = (X is identifier), rvcond(Frm_isvar) = ((retval) is bool), and kind(Frm_isvar) = und.

The form Frm_vartype specifies the types of variables in St and is defined as follows: pat(Frm_vartype) = (type of X), pars(Frm_vartype) = X, vkind(X) = quote, skind(X) = elem, pcond(Frm_vartype) = (X is variable), rvcond(Frm_vartype) = ((retval) is btype), and kind(Frm_vartype) = und.

The form Frm_varval specifies the values of variables in St and is defined as follows: pat(Frm_varval) = X, pars(Frm_varval) = X, vkind(X) = quote, skind(X) = elem, pcond(Frm_varval) = (X is variable), rvcond(Frm_varval) = ((retval) is (type of X)), and kind(Frm_varval) = und.

The variable declaration is defined by the rule Rul_vardec such that pat(Rul_vardec) = (var X Y), pars(Rul_vardec) = X Y, vkind(X) = vkind(Y) = quote, skind(X) = skind(Y) = elem, pcond(Rul_vardec) = ((X is identifier) and (not (X is btype)) and (not (X is variable)) and (Y is btype)), rvcond(Rul_vardec) = true, rkind(Rul_vardec) = defined, and body(Rul_vardec) = ((X is variable) ::= true) ((type of X) ::= Y).

The assignment statement is defined by the rule Rul_assign such that pat(Rul_assign) = (X := Y), pars(Rul_assign) = X Y, vkind(X) = quote, vkind(Y) = eval, skind(X) = skind(Y) = elem, pcond(Rul_assign) = ((X is variable) and (Y is (type of X))), rvcond(Rul_assign) = true, rkind(Rul_assign) = defined, and body(Rul_assign) = (X ::= Y).

The if statement is defined by the rule Rul_if such that pat(Rul_if) =

(if X then Y else Z), pars(Rul_if) $= X\ Y\ Z$, vkind(X) = vkind(Y) = vkind(Z) = quote, skind(X) = elem, skind(Y) = skind(Z) = seq, pcond(Rul_if) = (X is bool), rvcond(Rul_if) = true, rkind(Rul_if) = defined, and body(Rul_if) = (cases (if X then Y) (else Z)).

The while statement is defined by the rule Rul_while such that pat(Rul_while) = (while X do Y), pars(Rul_while) = $X\ Y$, vkind(X) = vkind(Y) = quote, skind(X) = elem, skind(Y) = seq, pcond(Rul_while) = true, rvcond(Rul_while) = true, rkind(Rul_while) = defined, and body(Rul_while) = (cases (if X then Y (while X do Y)) (else)).

An element sequence is called a program in the MPL language. A program *Elseq* is called correct in *St*, if *Elseq* is a correct sequence in *St*. A program *Elseq* is called incorrect in *St* if *Elseq* is not correct in *St*.

The program (var X int) (X := 5) (if (X = 5) then (X := 0) else) is correct in the empty state. Its execution returns the state St' such that dom(St') = {Frm_isvar, Frm_vartype, Frm_varval}, graph(St'(Frm_isvar)) = {(X, true)}, graph(St'(Frm_vartype)) = {(X, int)}, and graph(St'(Frm_varval)) = {(X, 0)}.

The program (X := 5) is incorrect in the empty state, since in accordance with the definition of the rule Rul_assign the identifier X must be a variable in this state.

The program (assume ((X is variable) and ((type of X) is int))) (X := 5) is correct in the empty statement, since in accordance with the definition of assume the assignment (X := 5) will not be executed. In accordance with the definition of the controlled backtracking, execution of this program terminates in the empty statement.

The program (var X int) (X := 5) (var X int) is incorrect in the empty statement, since in accordance with the definition of the rule Rul_vardec a variable can not be declared twice.

7. Conclusion

DSTSs are a special type of transition systems for determining domain-specific languages used to solve the problems of the development of computer language semantics and of the design, specification, prototyping, and verification of software systems. DSTSs form the basis of a comprehensive approach to solving these problems.

In this paper, the new object model of DSTSs has been described. It introduces new entities and concepts into the theory of DSTSs such as forms, element generators, and propagation of the indeterminate value with its handling. It also extends the concepts of substitution and pattern matching, determines the classification of forms and transition rules, adds constraints on the parameters and the return values of forms, improves the algorithm for finding the element values, considers the transition rules as a special kind of forms, improves the definitions of backtracking, safe configurations and runs, and correct control element sequences. The formal definition of the model programming language with the extensible set of basic types, based on this model, has been also given.

References

[1] Anureev I.S. Domain-specific transition systems: object model and language // System Informatics. – 2013. – No. 1. – P. 1–34 (In Russian).

[2] Anureev I.S. Integrated approach to analysis and verification of imperative programs // Bulletin NCC. Series: Computer Science. – Novosibirsk, 2011. – IIS Special Iss. 32. – P. 1–18.

[3] Anureev I.S. Introduction to the Atoment language // Bulletin NCC. Series: Computer Science. – Novosibirsk, 2010. – IIS Special Iss. 31. – P. 1–16.

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

[5] Anureev I.S., Maryasov I.V., Nepomniaschy V.A. Two-level mixed verification method of C-light programs in terms of safety logic // Bulletin NCC. Series: Computer Science. – Novosibirsk, 2012. – IIS Special Iss. 34. – P. 23–42.

[6] Anureev I.S. Program specific transition systems // Bulletin NCC. Series: Computer Science. – Novosibirsk, 2012. – IIS Special Iss. 34. – P. 1–21.

[7] AsmL: The Abstract state Machine Language. Reference Manual, 2002. – http://research.microsoft.com/en-us/projects/asml/

[8] Gurevich Y. Abstract state Machines: An Overview of the Project // Foundations of Information and Knowledge Systems (FoIKS). Proc. Third Internat. Symp. – Lect. Notes Comput. Sci. – 2004. – Vol. 2942. – P. 6–13.

[9] Anlauff M. XasM – An Extensible, Component-Based Abstract State Machines Language. – http://xasm.sourceforge.net/XasmAnl00/XasmAnl00.html

[10] 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.

[11] Parnas D.L. Really rethinking formal methods // Computer. IEEE Computer Society. – 2010. – Vol. 43 (1). – P. 28–34.