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.
I. S. Anureev

(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 \texttt{int}, \texttt{nat}, and \texttt{bool} denote the set of integers, the set of natural numbers, and the set \{\texttt{true}, \texttt{false}\}, respectively.

Let \(\texttt{Set}^\ast\) denote the set of all finite sequences consisting of the elements of a set \(\texttt{Set}\), \(\texttt{Set}^+\) denote the set of all finite nonempty sequences consisting of the elements of \(\texttt{Set}\), and \(\texttt{pset}(\texttt{Set})\) denote the set of all subsets of \(\texttt{Set}\). Let \(\texttt{empseq}\) denote the empty sequence, and \(\texttt{El}_1; : : : ; \texttt{El}_N\) denote the sequence consisting of the elements \(\texttt{El}_1; : : : ; \texttt{El}_N\). Let \(\texttt{len}(\texttt{Seq})\), \(\texttt{Seq}::\texttt{I}\), \(\texttt{first}(\texttt{Seq})\), and \(\texttt{last}(\texttt{Seq})\) denote the length of a sequence \(\texttt{Seq}\) and its \(\texttt{I}\)-th, first, and last elements, respectively.

Let \(\texttt{Set} \rightarrow \texttt{Set}'\) denote the set of all functions from \(\texttt{Set}\) to \(\texttt{Set}'\), and \(\texttt{Set} \rightarrow_t \texttt{Set}'\) denote the set of all total functions from \(\texttt{Set}\) to \(\texttt{Set}'\). Let \(\text{dom}(\texttt{Fun})\) denote the domain of a function \(\texttt{Fun}\), and \(\text{und}\) denote the indeterminate value. We assume that \(\texttt{Fun}(\texttt{Arg}) = \text{und}\), if \(\texttt{Arg} \notin \text{dom}(\texttt{Fun})\).

Let \(\text{dom}(\texttt{Fun}) \cap \text{dom}(\texttt{Fun}') = \emptyset\). The union \(\texttt{Fun} \cup \texttt{Fun}'\) of the functions \(\texttt{Fun}\) and \(\texttt{Fun}'\) is defined as the function \(\texttt{Fun}''\) such that \(\text{dom}(\texttt{Fun}'') = \text{dom}(\texttt{Fun}) \cup \text{dom}(\texttt{Fun}')\), \(\texttt{Fun}''(\texttt{Arg}) = \texttt{Fun}(\texttt{Arg})\) for \(\texttt{Arg} \in \text{dom}(\texttt{Fun})\), and \(\texttt{Fun}''(\texttt{Arg}) = \texttt{Fun}'(\texttt{Arg})\) for \(\texttt{Arg} \in \text{dom}(\texttt{Fun}')\). Let \(\text{range}(\texttt{Fun})\) denote the range of \(\texttt{Fun}\), i.e. the set \(\{\texttt{Fun}(\texttt{Arg}) \mid \texttt{Arg} \in \text{dom}(\texttt{Fun})\}\). Let \(\text{graph}(\texttt{Fun})\) denote the graph of \(\texttt{Fun}\), i.e. the set \(\{(\texttt{Arg}, \texttt{Fun}(\texttt{Arg})) \mid \texttt{Arg} \in \text{dom}(\texttt{Fun})\}\).

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

We say that \(\texttt{Set}\) is defined by the functions \(\texttt{Fun}_1, \ldots, \texttt{Fun}_N\), if \(\text{dom}(\texttt{Fun}_I) = \texttt{Set}\) for each \(1 \leq I \leq N\), and information about \(\texttt{Set}\) 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
3. The main concepts of the theory of domain-specific transition systems

The set $\text{dsts}$ of domain-specific transition systems is defined by the functions $\text{el}$, $\text{par}$, $\text{elgen}$, $\text{frm}$, $\text{match}$, and $\text{atom}$.

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

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

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

The function $\text{catched}$ specifies the propagation of the indeterminate value $\text{false}$ in the definition of the function $\text{val}$ (see below). The element $\text{false}$ 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 $\text{Par}$ be associated with $\text{ElSeq}$. If $\text{vkind}(\text{Par}) = \text{eval}$, $\text{Par}$ is called an evaluated parameter, and $\text{val}(\text{Elseq}, \text{St})$ is the value of $\text{Par}$. If $\text{vkind}(\text{Par}) = \text{quote}$, $\text{Par}$ is called a quoted parameter, and $\text{Elseq}$ is the value of $\text{Par}$. The function $\text{vkind}$ is called a parameter value specifier.

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

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

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

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

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

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

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

The sequence $Arg'$ is defined as follows:

- if $\text{embedded}(Elgen)(I) = \text{true}$, then $Arg'.I = \text{subst}(Arg, Sub)$;
- if $\text{embedded}(Elgen)(I) \neq \text{true}$, 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 $\text{frm}$ of forms is defined by the functions $\text{pat}$, $\text{pars}$, $\text{pcond}$, $\text{rvcond}$, and $\text{kind}$.

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 $\text{pat}$, $\text{pars}$, and $\text{pcond}$ to define whether $Elseq$ is an instance of $Frm$.

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

The sequence $\text{pars}(Frm) \in par^*$ such that the elements of $\text{pars}(Frm)$ are pairwise distinct defines the parameters of the pattern $\text{pat}(Frm)$. Let $1 \leq I \leq \text{len}(\text{pars}(Frm))$. The element $\text{par}(Frm).I$ is called a parameter of $Frm$. The number $\text{len}(\text{pars}(Frm))$ is called the arity of $Frm$ denoted by $\text{arity}(Frm)$. 
A form defines the static semantics of its instances by its value. The value of \( \mathsf{Frm} \) is defined as a function of the values of its parameters.

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

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

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

The forms of the fourth kind define the state of \( \mathsf{Dsts} \). The function \( \mathsf{St} \in \{\mathsf{Frm} \mid \mathsf{kind}(\mathsf{Frm}) = \mathsf{und}\} \to \cup_{n \in \mathbb{N}_0}(\mathsf{el}^n \to \mathsf{el}) \) such that \( \mathsf{St}(\mathsf{Frm}) \in \mathsf{el}^{\mathsf{arity}(\mathsf{Frm})} \to \mathsf{el} \) for all \( \mathsf{Frm} \) is called the state of \( \mathsf{Dsts} \). Let \( \mathsf{st} \) be the set of all states of \( \mathsf{Dsts} \). The state \( \mathsf{St} \) is called empty, if \( \mathsf{dom}(\mathsf{St}) = \emptyset \). The element \( \mathsf{St}(\mathsf{Frm}) \) is called the value of \( \mathsf{Frm} \) in \( \mathsf{St} \).

The form \( \mathsf{Frm} \) of the kind \( \mathsf{statedependent} \) is called a state-dependent predefined form. It is additionally defined by the function \( \mathsf{frmsem} \) such that \( \mathsf{frmsem}(\mathsf{Frm}) \in \mathsf{st} \to \cup_{n \in \mathbb{N}_0}(\mathsf{el}^n \to \mathsf{el}) \), and \( \mathsf{frmsem}(\mathsf{Frm})(\mathsf{St}) \in \mathsf{el}^{\mathsf{arity}(\mathsf{Frm})} \to \mathsf{el} \). The function \( \mathsf{frmsem} \) is called a form semantics. The element \( \mathsf{frmsem}(\mathsf{Frm})(\mathsf{St}) \) is called the value of \( \mathsf{Frm} \) in \( \mathsf{St} \).

The form \( \mathsf{Frm} \) of the kind \( \mathsf{statefree} \) is called a state-free predefined form. It is additionally defined by the function \( \mathsf{frmsem} \) such that \( \mathsf{frmsem}(\mathsf{Frm}) \in \cup_{n \in \mathbb{N}_0}(\mathsf{el}^n \to \mathsf{el}) \), and \( \mathsf{frmsem}(\mathsf{Frm})(\mathsf{St}) \in \mathsf{el}^{\mathsf{arity}(\mathsf{Frm})} \to \mathsf{el} \). The function \( \mathsf{frmsem} \) is called a form semantics. The element \( \mathsf{frmsem}(\mathsf{Frm})(\mathsf{St}) \) is called the value of \( \mathsf{Frm} \) in \( \mathsf{St} \).

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

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

- \( \mathsf{Elseq} = \mathsf{subst}(\mathsf{Frm}, \mathsf{Sub}) \);
- \( \mathsf{dom}(\mathsf{Sub}) \) is the set of parameters of \( \mathsf{Frm} \);
- if \( \mathsf{skind}(\mathsf{Par}) = \mathsf{elem} \), then \( \mathsf{Sub}(\mathsf{Par}) \in \mathsf{el} \);
\begin{itemize}
  \item if \text{skind}(Par) = \text{seq}, then \text{Sub}(Par) \in \mathbb{E}_1^*;
  \item \text{arity}(Frm) = \text{arity}(Frm') \text{, and } \text{Sub}(\text{pars}(Frm).I) = \text{Sub}'(\text{pars}(Frm').I) \text{ for all } (Frm', \text{Sub}') \in \text{match}(Elseq) \text{, and } 1 \leq I \leq \text{arity}(Frm).
\end{itemize}

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

The function \text{match}(Dsts) \in \mathbb{E}_1^* \times \text{st} \rightarrow \mathbb{Frm} \times \text{sub} \times \text{sub} \text{ is called a form matching with parameter meaning if } \text{match}(Elseq, St) = (Frm, Sub, Sub') \text{ if and only if the following properties are satisfied:}

\begin{itemize}
  \item \((Frm, Sub) \in \text{match}(Dsts)(Elseq)\);
  \item \text{Sub}' = \text{parval}(\text{pars}(Frm), Sub, St);
  \item \text{val}(\text{subst}(\text{pcond}(Frm), Sub'), St) = \text{true}.
\end{itemize}

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

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

The set \text{elgen}(Dsts) includes the functions \text{quote} \in \mathbb{E}_1^* \rightarrow \mathbb{E}_1 \text{, and } \text{eval} \in \mathbb{E}_1^* \rightarrow \mathbb{E}_1 \text{ such that } \text{embedded}(\text{quote}, 1) = \text{embedded}(\text{eval}, 1) = \text{true}. \text{ They specify the value of } Par \text{ in the case when } \text{Sub}(Par) \text{ has the form } \text{eval}(\text{Elseq}) \text{ or } \text{quote}(\text{Elseq}). \text{ If } \text{Sub}(Par) = \text{eval}(\text{Elseq}), \text{ then } \text{Sub}'(Par) = \text{val}(\text{Elseq}, St). \text{ If } \text{Sub}(Par) = \text{quote}(\text{Elseq}), \text{ then } \text{Sub}'(Par) = \text{Elseq}.

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

\begin{itemize}
  \item if \text{sub}(Par) = \text{eval}(\text{Elseq}), then \text{parval}(Par Parseq, Sub, St) = \{(Par \leftarrow \text{val}(\text{Elseq}, St))\} \cup \text{parval}(\text{Parseq}, Sub, St);
  \item if \text{sub}(Par) = \text{quote}(\text{Elseq}), then \text{parval}(Par Parseq, Sub, St) = \{(Par \leftarrow \text{Elseq})\} \cup \text{parval}(\text{Parseq}, Sub, St);
  \item if \text{vkind}(Par) = \text{eval}, \text{ and } \text{skind}(Par) = \text{elem}, then \text{parval}(Par Parseq, Sub, St) = \{(Par \leftarrow \text{val}(\text{Sub}(Par), St))\} \cup \text{parval}(\text{Parseq}, Sub, St);
  \item if \text{vkind}(Par) = \text{eval}, \text{ skind}(Par) = \text{seq}, \text{ and } \text{Sub}(Par) = \text{Elorpar}_1 \ldots \text{Elorpar}_N, \text{ then } \text{parval}(Par Parseq, Sub, St) = \{(Par \leftarrow \text{ifval}(\text{Elorpar}_1, \text{eval}, St) \ldots \text{ifval}(\text{Elorpar}_N, \text{eval}, St))\} \cup \text{parval}(\text{Parseq}, Sub, St);
\end{itemize}
• if $\text{vkind}(\text{Par}) = \text{quote}$, and $\text{skind}(\text{Par}) = \text{elem}$, then $\text{parval}(\text{Par} \ \text{Parseq}, \text{Sub}, \text{St}) = \{(\text{Par} \leftarrow \text{Sub}(\text{Par}))\} \cup \text{parval}(\text{Parseq}, \text{Sub}, \text{St})$;
• if $\text{vkind}(\text{Par}) = \text{quote}$, $\text{skind}(\text{Par}) = \text{seq}$, and $\text{Sub}(\text{Par}) = E_1 \ldots E_N$, then $\text{parval}(\text{Par} \ \text{Parseq}, \text{Sub}, \text{St}) = \{(\text{Par} \leftarrow \text{ifval}(E_1, \text{quote}, \text{St}) \ldots \text{ifval}(E_N, \text{quote}, \text{St}))\} \cup \text{parval}(\text{Parseq}, \text{Sub}, \text{St})$.

The function $\text{ifval} \in \text{el}^+ \times \text{st} \times \{\text{eval}, \text{quote}\} \rightarrow \text{el}$ is defined as follows:
• $\text{ifval evaluator}(\text{Elseq}, \text{St}, V \text{parorqpar}) = \text{val}(\text{Elseq}, \text{St})$;
• $\text{ifval quote evaluator}(\text{Elseq}, \text{St}, V \text{parorqpar}) = \text{Elseq}$;
• $\text{ifval El, St, eval} = \text{val}(\text{El}, \text{St})$;
• $\text{ifval El, St, quote} = \text{El}$.

The function $\text{val} \in \text{el}^+ \times \text{st} \rightarrow \text{el}$ called an element sequence meaning is defined as follows (the first proper rule is applied):
• $\text{val}(\text{true}, \text{St}) = \text{true}$;
• if $\text{match}(\text{Elseq}) = (\text{Frm}, \text{Sub}, \text{Sub}')$, $\text{pars}(\text{Frm}) = \text{Par}_1 \ldots \text{Par}_N$, $\text{Arg} = \text{Sub}'(\text{Par}_1), \ldots, \text{Sub}'(\text{Par}_N)$, and $\text{Retvalcond}(U)$ denotes

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

– if $\text{catched}(\text{Par}_1) \neq \text{true}$, and $\text{Sub}'(\text{Par}_1) = \text{false}$ for some $1 \leq I \leq \text{arity}(\text{Frm})$, then $\text{val}(\text{Elseq}, \text{St}) = \text{false}$;
– if $\text{kind}(\text{Frm}) = \text{und}$, $\text{St}(\text{Frm}) \neq \text{und}$, and $\text{Retvalcond}(\text{St}(\text{Frm})(\text{Arg}))$, then $\text{val}(\text{Elseq}, \text{St}) = \text{St}(\text{Frm})(\text{Arg})$;
– if $\text{kind}(\text{Frm}) = \text{statedependent}$, and $\text{Retvalcond}(\text{frmsem}(\text{Frm})(\text{St})(\text{Arg}))$, then $\text{val}(\text{Elseq}, \text{St}) = \text{frmsem}(\text{Frm})(\text{St})(\text{Arg})$;
– if $\text{kind}(\text{Frm}) = \text{statefree}$, and $\text{Retvalcond}(\text{frmsem}(\text{Frm})(\text{Arg}))$, then $\text{val}(\text{Elseq}, \text{St}) = \text{frmsem}(\text{Frm})(\text{Arg})$;
– if $\text{kind}(\text{Frm}) = \text{defined}$, and $\text{Retvalcond}(\text{val}(\text{subst}(\text{body}(\text{Frm}), \text{Sub}', \text{St})))$, then $\text{val}(\text{Elseq}, \text{St}) = \text{val}(\text{subst}(\text{body}(\text{Frm}), \text{Sub}', \text{St}))$;
• if $\text{atom}(\text{Dsts})(\text{Elseq}) = \text{true}$, then $\text{val}(\text{Elseq}, \text{St}) = \text{Elseq}$;
• $\text{val}(\text{Elseq}, \text{St}) = \text{false}$.

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

The function $\text{atom}(\text{Dsts}) \in \text{el}^+ \rightarrow_1 \text{bool}$ 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 $\text{tr} \in \text{conf} \times \text{conf} \rightarrow \text{bool}$ called a transition relation. The set $\text{conf}$ of configurations and the function $\text{tr}$ are defined below. The system $\text{Dsts}$ can go from $\text{Conf}$ to $\text{Conf}'$ if and only if $\text{tr}(\text{Conf}, \text{Conf}') = \text{true}$.

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

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

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

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

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

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

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

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

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

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

The function \( tr \) with an additional argument \( Rul \) is defined as follows:

\[
tr(Conf, Conf'; Rul) = true \quad \text{if and only if } \quad seq(Conf) = Elseq Elseq', \text{ match}(Elseq, st(Conf)) = (Rul, Sub, Sub'), \text{ and one of two conditions is satisfied: } \quad rkind(Rul) = \text{ predefined, and } \quad \text{rulsem}(Rul)(Conf, Conf', Sub') = true, \text{ or } \quad rkind(Rul) = \text{ defined, seq}(Conf') = subst(body(Rul), Sub') Elseq', \text{ and } \quad st(Conf') = st(Conf).
\]

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 \( tr(Conf, Conf', Rul) = true \).

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:

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

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

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

**DSTS with controlled backtracking.** A DSTS \( Dsts \) with backtracking is called a DSTS with controlled backtracking, if

- \( \text{elgen}(Dsts) \) includes the functions \( \text{backtrack} \) and \( \text{branch} \) such that \( \text{backtrack} \in \text{el, branch} \in \text{el} \rightarrow \text{el} \rightarrow_{1} \text{el} \), and \( \text{embedded}(\text{branch})(1) = true \). The element \( \text{backtrack} \) called a backtracking condition initiates backtracking to the previous backtracking point. The element \( \text{branch}(ElSeqSeq) \) called a branch element is used to define possible variants in the backtracking point given by the elements of \( ElSeqSeq \).
- \( tr \in \text{conf}^* \times \text{conf}^* \rightarrow \text{bool} \) is a controlled backtracking.
Let ba(St) and fi(St) be configurations such that seq(ba(St)) = backtrack, st(ba(St)) = St, rulset(ba(St)) = $\emptyset$, seq(fi(St)) = empseq, st(fi(St)) = St, and rulset(ba(St)) = $\emptyset$.

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

- $Conf seq = Conf seq'' Conf, seq(Conf) = backtrack Elseq$, where $Elseq \neq empseq$, or rulset(Conf) $\neq \emptyset$, and $Conf seq' = Conf seq'' ba(st(Conf))$;
- $Conf seq = Conf seq'' Conf ba(St)$, where seq(Conf) = branch(Elseq' Elseqseq) Elseq, Conf seq' = Conf seq'' Conf' Conf'', seq(Conf') = branch(Elseqseq Elseq, st(Conf') = st(Conf), rulset(Conf') = rulset(Conf), seq(Conf'') = Elseq' Elseq, st(Conf'') = ifst(st(Conf), St), and rulset(Conf'') = $\emptyset$;
- $Conf seq = Conf seq'' Conf ba(St)$, where seq(Conf) = branch(empseq) Elseq, and Conf seq' = Conf seq'' Conf' ba(St);
- $Conf seq = Conf seq'' Conf ba(St)$, where seq(Conf) = branch(empseq) Elseq, and Conf seq' = Conf seq'' branch(st(Conf), St);
- $Conf seq = Conf seq'' Conf ba(St)$, where seq(Conf) = branch(Elseq' Elseqseq) Elseq, Conf seq' = Conf seq'' Conf' Conf'', seq(Conf') = branch(Elseqseq Elseq, st(Conf') = st(Conf), rulset(Conf') = rulset(Conf), seq(Conf'') = Elseq' Elseq, st(Conf'') = st(Conf), and rulset(Conf'') = $\emptyset$;
- $Conf seq = Conf seq'' Conf, seq(Conf) = branch(empseq) Elseq, and Conf seq' = Conf seq'' branch(st(Conf))$;
- $Conf seq = Conf seq'' Conf ba(St)$, where seq(Conf) = Elseq' Elseq, Elseq' $\notin$ predel(Dsts), Rul $\in$ rul(Dsts) \ rulset(Conf), tr(Conf', Conf', Rul) = true, seq(Conf') = Elseq' Elseq, st(Conf') = ifst(st(Conf), St), seq(Conf'') = Elseq' Elseq seq(Conf'') = Elseq' Conf seq' Conf seq' b = Elseq' Conf seq, Rul = $\emptyset$, seq(Conf') = Elseq' Elseq, st(Conf') = St, Rul = $\emptyset$, seq(Conf') = Elseq' Elseq seq(Conf') = Elseq' Elseq seq(Conf') = $\emptyset$;
- $Conf seq = Conf seq'' Conf ba(St)$, and Conf seq' = Conf seq'' Conf' ba(St);
- $Conf seq = Conf seq'' Conf, seq(Conf) = Elseq' Elseq, Elseq' $\notin$ predel, Rul $\in$ rul(Dsts) \ rulset(Conf), tr(Conf, Conf' Rul) = true, Conf seq' = Conf seq'' Conf'' seq(Conf'') = $\emptyset$. 


\[\text{seq}(\text{Conf}), \text{st}(\text{Conf}^\prime) = \text{st}(\text{Conf}), \text{rulset}(\text{Conf}^\prime) = \text{rulset}(\text{Conf}) \cup \{\text{Rul}\}, \text{seq}(\text{Conf}^\prime) = \text{seq}(\text{Conf}^\prime), \text{st}(\text{Conf}^\prime) = \text{st}(\text{Conf}^\prime), \text{an rulset}(\text{Conf}^\prime) = \emptyset;\]

- \text{Confseq} = \text{Confseq}^\prime \text{Conf}, where \text{seq}(\text{Conf}) = \text{Elseq}^0 \text{Elseq}, \text{Elseq}^0 \in \text{predel}(\text{Dsts}), \text{rulsem}(\text{Dsts})(\text{Conf}, \text{Conf}^\prime) = \text{true}, \text{Confseq} = \text{Confseq}^\prime \text{Conf}^\prime \text{Conf}^\prime, \text{seq}(\text{Conf}^\prime) = \text{Elseq}, \text{and only if} \text{seq}(\text{Conf}^\prime) = \text{empseq}, \text{and st}(\text{Conf}^\prime) = \text{st}(\text{Conf}^\prime). \text{The element stop is called a stop element.}\]

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

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

- if \(\text{Sub}^\prime(\text{Par}) = \text{true}\), then \(\text{seq}(\text{Conf}^\prime) = \text{Elseq}\);
- \(\text{seq}(\text{Conf}^\prime) = \text{backtrack} \text{Elseq}\).

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

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

- if \(\text{match}(\text{Sub}^\prime(\text{Par}), \text{st}(\text{Conf})) = (\text{Frm}, \text{Sub}_1, \text{Sub}_1^\prime)\), \(\text{kind}(\text{Frm}) = \text{und}\), \(\text{arity}(\text{Frm}) = N\), \(\text{Arg} = \text{Sub}_1^\prime(\text{pars}(\text{Frm}).1)\), \ldots, \(\text{Sub}_1^\prime(\text{pars}(\text{Frm}).N)\), then \(\text{seq}(\text{Conf}^\prime) = \text{Elseq}, \text{odif}(\text{st}(\text{Conf}^\prime), \text{}}\)
\[ st(Conf), \{ Frm \} = true, ori(st(Conf')(Frm), st(Conf)(Frm), \{(Arg)\}) = true, \text{ and } st(Conf')(Frm)(Arg) = Sub'(Par'); \]

- seq(Conf') = fail El Elseq, and st(Conf') = st(Conf).

The element El is called a form update.

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

- if Sub'(Par) = true, then seq(Conf') = Elseq;

- seq(Conf') = fail El Elseq.

The element El is called a safety condition.

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

A form Rul is called a conditional branching rule, if pat(Rul) = cases(Par, Par', Par''), pars(Rul) = Par Par' Par'', vkind(Par) = quote, and skind(Par) = seq for each Par \( \in \text{pars(Rul)} \), cond(Rul) = true, rkind(Rul) = predefined, rulsem(Rul)(Conf, Conf') = true, where seq(Conf) = El Elseq, if and only if match(El, st(Conf)) = (Rul, Sub, Sub'), st(Conf') = st(Conf), and seq(Conf') = branch(Par.1, ..., Arg.N [Sub'(Par'')]), where Arg.I = assume(Sub'(Par).1 Sub'(Par').1 for 1 \( \leq I \leq \text{len(Sub'(Par))} = N \). 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 \{a, ..., z, A, ..., Z\}, digits from \{0, ..., 9\}, and the underscore character _, starting with a letter), the finite set btype \( \subseteq \text{id} \) of basic types such that lit(Btype) is a set of literals of the type Btype \( \in \text{btype} \), lit(Btype) \( \cap \text{id} = \emptyset \) for each Btype, int \( \in \text{btype} \), lit(int) = \{..., -2, -1, 0, 1, 2, ...\}, bool \( \in \text{btype} \), and lit(bool) = \{true, 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:
• \( el \stackrel{def}{=} id \cup \bigcup_{Elgen \in elgen(Dsts)} \text{range}(Elgen) \);

• \( \text{par}(Dsts)(El) = \text{true} \) if and only if \( El \in id \setminus \text{dtype} \);

• \( \text{elgen}(Dsts) \stackrel{def}{=} \{ \text{delcomp}, \text{eval}, \text{quote}, \text{fail}, \text{backtrack}, \text{branch}, \text{stop}, \text{assume}, \text{frmupd}, \text{assert}, \text{cases} \} \);

• if \( \text{len}(Elseq) = N \), then \( \text{delcomp}(Elseq) \stackrel{def}{=} (Elseq.1 \ldots Elseq.N) \);

• \( \text{eval}(Elseq) \stackrel{def}{=} (\text{eval} Elseq) \);

• \( \text{quote}(Elseq) \stackrel{def}{=} (\text{quote} Elseq) \);

• \( \text{retval}(Dsts) \stackrel{def}{=} \text{retval} \);

• \( \text{fail} \stackrel{def}{=} \text{fail} \);

• \( \text{backtrack} \stackrel{def}{=} \text{backtrack} \);

• \( \text{branch}(Elseqseq) \stackrel{def}{=} (\text{branch} Elseqseq) \);

• \( \text{stop} \stackrel{def}{=} \text{stop} \);

• \( \text{assume}(Elseq) \stackrel{def}{=} (\text{assume} Elseq) \);

• \( \text{frmupd}(Elseq, Elseq') \stackrel{def}{=} (Elseq ::= Elseq') \);

• \( \text{assert}(Elseq) \stackrel{def}{=} (\text{assert} Elseq) \);

• if \( \text{len}(Elseq) = N \), then \( \text{cases}(Elseq, Elseqseq, Elseq') \stackrel{def}{=} (\text{cases} (\text{if} Elseq.1 \text{then} Elseqseq.1) \ldots (\text{if} Elseq.N \text{then} Elseqseq.N)(\text{else} Elseq')) \);

• \( \text{frm}(Dsts) \stackrel{def}{=} \{ \text{Frm}_\text{Id} \mid Id \in id \} \cup \{ \text{Rul}_\text{Id} \mid Id \in id \} \);

• the algorithm \( \text{match}(Dsts) \) choose the first proper element sequence from left to right. For example, if \( \text{pat}(Fr \text{m}) = (\text{if} X \text{then} Y \text{else} Z) \), \( \text{pars}(Fr \text{m}) = X Y Z \), \( \text{skind}(X) = \text{elem} \), \( \text{skind}(Y) = \text{seq} \), and \( \text{skind}(Z) = \text{seq} \), then \( (Fr \text{m}, (X \leftarrow A, Y \leftarrow B, Z \leftarrow C \text{ else} D)) \in \text{match}(Dsts)((\text{if} A \text{ then} B \text{ else} C \text{ else} D)) \), and \( (Fr \text{m}, (X \leftarrow A, Y \leftarrow B \text{ else} C, Z \leftarrow D)) \notin \text{match}(Dsts)((\text{if} A \text{ then} B \text{ else} C \text{ else} D)) \);

• \( \text{atom}(Dsts)(El) = \text{true} \) if and only if \( El \in id \cup \bigcup_{B \text{type} \in b \text{type}} \text{lit}(B \text{type}) \).

The form \( \text{Frm}_\text{bool} \) is associated with the type \( \text{bool} \), and it is defined as follows: \( \text{pat}(\text{Frm}_\text{bool}) = (X \text{ isof} \text{ bool}) \), \( \text{pars}(\text{Frm}_\text{bool}) = X \), \( \text{vkind}(X) = \text{eval} \), \( \text{skind}(X) = \text{elem} \), \( \text{pcond}(\text{Frm}_\text{bool}) = \text{true} \), \( \text{rvcond}(\text{Frm}_\text{bool}) = \text{true} \), \( \text{kind}(\text{Frm}_\text{bool}) = \text{statefree} \), and \( \text{frmsem}(\text{Frm}_\text{bool})(El) = \text{true} \) if and only if \( El \in \text{lit}(\text{bool}) \).
The form \texttt{Frm\_sort} checks whether an element sequence belongs to a particular sort, and it is defined as follows: \( \text{pat}(\text{Frm\_sort}) = (X \text{ is } Y) \), \( \text{pars}(\text{Frm\_sort}) = X \), \( \text{vkind}(X) = \text{quote} \), \( \text{vkind}(y) = \text{eval} \), \( \text{skind}(X) = \text{skind}(Y) = \text{seq} \), \( \text{pcond}(\text{Frm\_sort}) = \text{true} \), \( \text{rvcond}(\text{Frm\_sort}) = ((\text{retval}) \text{ isof bool}) \), \( \text{kind}(\text{Frm\_sort}) = \text{statefree} \), and \( \text{frmsem}(\text{Frm\_sort})(El) = \text{true} \) if and only if \((X \text{ isof } Y)\). The forms with the pattern \((X \text{ isof } Elseq)\) for particular sorts \(Elseq\) are defined below.

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

The form \texttt{Frm\_btype} specifies the characteristic function for \texttt{Btype \neq bool}, and it is defined as follows: \( \text{pat}(\text{Frm\_btype}) = (X \text{ isof } Btype) \), \( \text{pars}(\text{Frm\_btype}) = X \), \( \text{vkind}(X) = \text{quote} \), \( \text{skind}(X) = \text{elem} \), \( \text{pcond}(\text{Frm\_btype}) = \text{true} \), \( \text{rvcond}(\text{Frm\_btype}) = ((\text{retval}) \text{ is bool}) \), \( \text{kind}(\text{Frm\_btype}) = \text{statefree} \), and \( \text{frmsem}(\text{Frm\_btype})(El) = \text{true} \) if and only if \(El \in \text{btype}\).

The operations \(=\), and \(!=\) on basic types, the arithmetic operations \(+, -, *, \text{div}, \text{mod}\), and arithmetic relations \(<, >, \leq, \geq\) on integers, the boolean operations \(\text{and}, \text{or}, \text{not}\), \(\Rightarrow\) are defined by the corresponding state-free predefined forms \texttt{Frm\_eq}, \texttt{Frm\_neq}, \texttt{Frm\_add}, \texttt{Frm\_sub}, \texttt{Frm\_mul}, \texttt{Frm\_div}, \texttt{Frm\_mod}, \texttt{Frm\_less}, \texttt{Frm\_more}, \texttt{Frm\_lesseq}, \texttt{Frm\_moreeq}, \texttt{Frm\_and}, \texttt{Frm\_or}, \texttt{Frm\_not}, and \texttt{Frm\_implies}:

- \(\text{pat}(\text{Frm\_eq}) = (X = Y)\), \(\text{pars}(\text{Frm\_eq}) = X\ Y\), \(\text{vkind}(X) = \text{vkind}(Y) = \text{eval} \), \(\text{skind}(X) = \text{skind}(Y) = \text{elem} \), \(\text{pcond}(\text{Frm\_eq}) = \text{true} \), \(\text{rvcond}(\text{Frm\_eq}) = ((\text{retval}) \text{ is bool}) \), \(\text{kind}(\text{Frm\_eq}) = \text{statefree} \), and \(\text{frmsem}(\text{Frm\_eq})(El, El') = \text{true} \) if and only if \(El \in \text{Btype}, El' \in \text{Btype}\) for some \texttt{Btype}, and \(El = El'\);

- \(\text{pat}(\text{Frm\_add}) = (X \text{ + } Y)\), \(\text{pars}(\text{Frm\_add}) = X\ Y\), \(\text{vkind}(X) = \text{vkind}(Y) = \text{eval} \), \(\text{skind}(X) = \text{skind}(Y) = \text{elem} \), \(\text{pcond}(\text{Frm\_add}) = ((X \text{ is int}) \text{ and } (Y \text{ is int})) \), \(\text{rvcond}(\text{Frm\_add}) = ((\text{retval}) \text{ is int}) \), \(\text{kind}(\text{Frm\_add}) = \text{statefree} \), and \(\text{frmsem}(\text{Frm\_add})(El, El') = El'' \) if and only if \(El'' = El + El'\);

- \(\text{pat}(\text{Frm\_less}) = (X < Y)\), \(\text{pars}(\text{Frm\_less}) = X\ Y\), \(\text{vkind}(X) = \text{vkind}(Y) = \text{eval} \), \(\text{skind}(X) = \text{skind}(Y) = \text{elem} \),
pcond(Frm\_less) = ((X is int) and (Y is int)), rvcond(Frm\_less) = ((retval is bool), kind(Frm\_less) = statefree, andfrmsem(Frm\_less)(El, El') = true if and only if El < El';

• \(\text{pat(Frm\_and)} = (X \text{ and } Y), \text{pars}(Frm\_and) = X Y, \text{vkind}(X) = vkind(Y) = \text{eval}, \text{skind}(X) = \text{skind}(Y) = \text{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: \(\text{pat(Frm\_isvar)} = (X \text{ is variable}), \text{pars}(Frm\_isvar) = X, \text{vkind}(X) = \text{quote}, \text{skind}(X) = \text{elem}, pcond(Frm\_isvar) = (X \text{ 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: \(\text{pat(Frm\_vartype)} = (\text{type of } X), \text{pars}(Frm\_vartype) = X, \text{vkind}(X) = \text{quote}, \text{skind}(X) = \text{elem}, pcond(Frm\_vartype) = (X \text{ 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: \(\text{pat(Frm\_varval)} = X, \text{pars}(Frm\_varval) = X, \text{vkind}(X) = \text{quote}, \text{skind}(X) = \text{elem}, pcond(Frm\_varval) = (X \text{ 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 \(\text{pat(Rul\_vardec)} = (\text{var } X Y), \text{pars}(Rul\_vardec) = X Y, \text{vkind}(X) = vkind(Y) = \text{quote}, \text{skind}(X) = \text{skind}(Y) = \text{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 \text{body}(Rul\_vardec) = ((X is variable) ::= true)((type of } X) ::= Y).

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

The if statement is defined by the rule Rul\_if such that \(\text{pat(Rul\_if)} = \ldots \)
(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 state, 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 state.

The program (var X int) (X := 5) (var X int) is incorrect in the empty state, 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


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

