УДК 004.8

Conceptual transition systems and their application to development of conceptual models of programming languages[*]

Anureev I.S. (Institute of Informatics Systems),

Promsky A.V. (Institute of Informatics Systems)

In the paper the notion of the conceptual model of a programming language is proposed. This formalism represents types of the programming language, values, exceptions, states and executable constructs of the abstract machine of the language, and the constraints for these entities at the conceptual level. The new definition of conceptual transition systems oriented to specification of conceptual models of programming languages is presented, the language of redefined conceptual transition systems CTSL is described, and the technique of the use of CTSL as a domain-specific language of specification of conceptual models of programming languages is proposed. The conceptual models for the family of sample programming languages illustrate this technique.

Keywords: *operational semantics, conceptual transition system, programming language, conceptual model, domain-specific language*

1. Introduction

This paper relates to the development of operational semantics of programming languages. Following [1], we distinguish two parts of the operational semantics of a programming language. The structural part defines how the elements of the language relate to runtime elements that an abstract machine of the programming language can use at runtime. The structural part is called instantiation semantics or structure-only semantics [2]. The dynamic part describes the actual state changes that take place at runtime.

In traditional operational semantics approaches [3–6], the main focus is on state changes, while the structural part is defined ad-hoc. The modern programming languages becomes more complex. Therefore, development of formalisms, languages and frameworks to describe the instantiation semantics is very important problem.

[*] Partially supported by RFBR under grants 15-01-05974 and 17-01-00789 and SB RAS interdisciplinary integration project No.15/10.

The meta-model-based object-oriented approach [1] to description of the instantiation semantics uses MOF (EMF) [7]. The algebraic approach [8] is based on abstract state machines. Abstract state machines are the special kind of transition systems in which states are algebraic systems. The structural part of the operational semantics is flexibly modelled by the appropriate choice of the symbols of the signature of an algebraic system. Rewrite-based approach is implemented in the frameworks K [9] and Maude [10].

These approaches do not take into account the natural conceptual nature of instantiation semantics which is easier to describe in the ontological terms of concepts, their instances and attributes.

In this paper, we introduce the notion of the conceptual model of a programming language. This formalism describes the instantiation semantics at the conceptual level. The conceptual model is specified in terms of conceptual transition systems (CTSs) [11] in the language of conceptual transition systems CTSL [12]. Thus, CTSL acts as a domain-specific language oriented to specification of conceptual models of programming languages.

The paper has the following structure. The preliminary concepts and notation are given in section 2. The new definition of CTSs is presented in section 3. The basic definitions of the theory of CTSs are given in sections 4 and 5. The language CTSL for redefined CTSs is described in section 6. The definition of the conceptual model of a programming language is introduced, and the technique of development of conceptual models of programming languages is illustrated by the sample programming language examples in section 7.

2. Preliminaries

The preliminary concepts and notation are given in this section.

2.1. Sets and sequences

Let $\$w$, $\$w1$, $\$w2$, … denote elements of the sort w, where w is a word, and $\$\$w$ denote the set of all elements of the sort w. For example, if n is a sort of natural numbers, then $\$n$, $\$n1$, … are natural numbers, and $$n is the set of all natural numbers.

Let $\$\$o$ and $\$\$set$ be sets of objects and sets considered in this paper. Let $\$\$i$, $\$\$n$, and $\$\$bo$ be sets of integers, natural numbers (with zero), and boolean values $true$ and $false$.

Let $\$\$se$ denote the set of finite sequences of the form $\$o1$ ... $\$o\$n$. Let $\$\$w^*$ denote the set of finite sequences of the form $\$w1$ … $\$w\$n$, and $\$w^*$, $\$w^*1$, $\$w^*2$, and so on denote the elements of the set $\$\$w^*$. Let $[es]$ denote the empty sequence. Let $\$\$w^+$ denote the set of finite nonempty sequences of the form $\$w1$ … $\$w\$n$, and $\$w^+$, $\$w^+1$, $\$w^+2$, and so on denote the elements of the set $\$\$w^+$.

Let $[repeat\ \$o\ \$n]$ denote the sequence consisting of $\$n$-th occurrences of the object $\$o$.

Let $[\$o \in \$se]$ and $[\$se1 \sqsubseteq \$se2]$ denote $\$o \in \{\$se\}$ and $\{\$se1\} \sqsubseteq \{\$se2\}$. Let $[len\ \$se]$ denote the length of $se. Let und denote the undefined value. Let $[\$se\ ..\ \$n]$ denote the $\$n$-th element of $\$se$. If $[len\ \$se] < \$n$, then $[\$se\ ..\ \$n] = und$. Let $[\$se\ ..\ \$n := \$o]$ denote the result $\$se1$ of replacement of $\$n$-th element in $\$se$ by $\$o$. If $\$n > [len\ \$se]$, then $\$se1 = \$se\ [repeat\ und\ [[len\ \$se] - \$n - 1]]\ \$o$.

Let $[\$o \in \$se]$ and $[\$se1 \sqsubseteq \$se2]$ denote $\$o \in \{\$se\}$ and $\{\$se1\} \sqsubseteq \{\$se2\}$. Let $[len\ \$se]$ denote the length of $se. Let und denote the undefined value. Let $[\$se\ ..\ \$n]$ denote the $\$n$-th element of $\$se$. If $[len\ \$se] < \$n$, then $[\$se\ ..\ \$n] = und$. Let $[\$se\ ..\ \$n := \$o]$ denote the result $\$se1$ of replacement of $\$n$-th element in $\$se$ by $\$o$. If $\$n = [len\ \$se] + 1$, then $\$se1 = \$se\ \$o$. If $\$n > [len\ \$se] + 1$, then $\$se1 = und$.

Let $[\$o1 \prec_{[\![\$se]\!]} \$o2]$ denote the fact that there exist $\$o^*1$, $\$o^*2$ and $\$o^*3$ such that $\$se = \$o^*1\ \$o1\ \$o^*2\ \$o2\ \$o^*3$.

Let $[\$o\ \$o1 \hookleftarrow \$o2]$ denote the result of replacement of all occurrences of $\$o1$ in $\$o$ by $\$o2$. Let $[\$se\ \$o \hookleftarrow\!* \ \$o1]$ denote the result of replacement of each element $\$o2$ in $\$se$ by $[\$o1\ \$o \hookleftarrow \$o2]$. For example, $[a\ b\ x \hookleftarrow\!* \ (f\ x)]$ denotes $(f\ a)\ (f\ b)$.

Let $\$o1, \$o2 \in \$\$se \cup \$\$set$. Then $[\$o1 =_{set} \$o2]$ denote that the sets of elements of $\$o1$ and $\$o2$ coincide, and $[\$o1 =_{mul} \$o2]$ denote that the multisets of elements of $\$o1$ and $\$o2$ coincide.

The above defined operations on the set $\$\$se$ are also applied to the set $\{(\$se) \mid \$se \in \$\$se\}$. The results of $[(\$se)\ ..\ \$n]$, $[\$o \in (\$se)]$, $[(\$se1) \sqsubseteq (\$se2)]$, $[\$o1 \prec_{[\![(\$se)]\!]} \$o2]$, $[(\$se)\ \$o \hookleftarrow^* \$o1]$, $[len\ (\$se)]$, $[(\$se)\ ..\ \$n \coloneqq \$o]$ and $[and\ (\$se)]$ are $[\$se\ ..\ \$n]$, $[\$o \in \$se]$, $[\$se1 \sqsubseteq \$se2]$, $[\$o1 \prec_{[\![\$se]\!]} \$o2]$, $[\$se\ \$o \hookleftarrow^* \$o1]$, $[len\ \$se]$, $[\$se\ ..\ \$n \coloneqq \$o]$ and $[and\ \$se]$.

Let $[(o^*) + (\$o^*1)]$, $[\$o\ .{+}(o^*)]$ and $[(o^*)\ {+}.\ \$o]$ denote $(\$o^*\ \$o^*1)$, $(\$o\ \$o^*)$ and $(\$o^*\ \$o)$.

2.2. Contexts

The terms used in the paper can be context-dependent. A context has the form $[\![\$o^*]\!]$. The elements of $\$o^*$ are called embedded contexts. The context in which some embedded contexts are omitted is called a partial context. All omitted embedded contexts are considered bound by the existential quantifier, unless otherwise specified.

Let o[\![\$o^*]\!]$ denote the object $o in the context $[\![\$o^*]\!]$. The expression 'in $[\![\$o1, \$o^*]\!]$' can be rewritten as 'in $[\![\$o1]\!]$ in $[\![\$o^*]\!]$', if this does not lead to ambiguity.

2.3. Functions

Let $\$\$f$ be a set of functions. Let $\$\$a$ and $\$\$v$ be sets of objects called arguments and values. Let $[\$f\ a^*]$ denote the result of application of $\$f$ to $\$a^*$. Let $[support\ \$f]$ denote the support in $[\![\$f]\!]$, i. e. $[support\ \$f] = \{\$a \mid [\$f\ \$a] \neq und\}$. Let $[image\ \$f\ \$set]$ denote the image in $[\![\$f,\ \$set]\!]$, i. e. $[image\ \$f\ \$set] = \{[\$f\ \$a] : \$a \in \$set\}$. Let $[image\ \$f]$ denote the image in $[\![\$f,\ [support\ \$f]]\!]$. Let $[narrow\ \$f\ \$set]$ denote the function $\$f1$ such that $[support\ \$f1] = [support\ \$f] \cap \$set$, and $[\$f1\ \$a] = [\$f\ \$a]$ for each $\$a \in [support\ \$f1]$. The function $\$f1$ is called a narrowing of $\$f$ to $\$set$. Let $[support\ \$f1] \cap [support\ \$f2] = \emptyset$. Let $\$f1 \cup \$f2$ denote the union $\$f$ of $\$f1$ and $\$f2$ such that $[\$f\ \$a] = [\$f1\ \$a]$ for each $\$a \in [support\ \$f1]$, and $[\$f\ \$a] = [\$f2\ \$a]$ for each $\$a \in [support\ \$f2]$. Let $\$f1 \subseteq \$f2$ denote the fact that $[support\ \$f1] \subseteq [support\ \$f2]$, and $[\$f1\ \$a] = [\$f2\ \$a]$ for each $\$a \in [support\ \$f1]$.

An object $\$u$ of the form $\$a := \$v$ is called an update. The objects $\$a$ and $\$v$ are called an argument and values in $[\![\$u]\!]$. Let $\$\$u$ be a set of updates.

Let $[\$f\ \$u]$ denote the function $\$f1$ such that $[\$f1\ \$a] = [\$f\ \$a]$ if $\$a \neq \$a[\![\$u]\!]$, and $[\$f1\ \$a[\![\$u]\!]] = \$v[\![\$u]\!]$. Let $[\$f\ \$u\ \$u^*]$ be a shortcut for $[[\$f\ \$u]\ \$u^*]$. Let $[\$f\ \$a.\$a1.\ \dots\ .\$a\$n \coloneqq \$v]$ be a shortcut for $[\$f\ \$a \coloneqq [[\$f\ \$a]\ \$a1.\ \dots\ .\$a\$n \coloneqq \$v]]$. Let $[\$u^*]$ be a shortcut for $[\$f\ \$u^*]$, where $[support\ \$f] = \emptyset$.

Let $[if\ \$con\ then\ \$o1\ else\ \$o2]$ denote the object $\$o$ such that $\$o = \$o1$ for $\$con = true$, and $\$o = \$o2$ for $\$con = false$.

3. Conceptual transition systems

The notion of conceptual transition systems (CTSs) is based on the notion of conceptual structures.

Let $\$\$ato$ be a set of objects called atoms.

The set $\$\$cs$ of conceptual structures in $[\![\$\$ato]\!]$ is defined as follows:

- $\$ato \in \$\$cs$;
- $(\$cs^*) \in \$\$cs$;
- if the elements of $\$cs^+$ are pairwise distinct, and $\$cs \neq und$, then $\$cs\colon (\$cs^+) \in \$\$s$;
- if the elements of $\$cs^+$ are pairwise distinct, and $\$cs \neq und$, then $(\$cs^+)\colon\colon \$cs \in \$\$cs$.

A structure $\$cs$ is atomic if $\$cs \in \$\$ato$.

A structure $\$ccs$ is a compound structure if $\$ccs$ has the form $(\$cs^*)$. The operation $(\dots)$ is called a sequential composition. A structure $\$cs$ is an element in $[\![\$ccs]\!]$ if $\$cs^* = \$cs^*1\ \$cs\ \$cs^*2$ for some $\$cs^*1$ and $\$cs^*2$. The structure $(\ )$ is called an empty structure. Let $\$\$ccs$ be a set of compound structures.

Let $\$\$t$ and $\$\$v$ be sets of objects called types and values. An object $\$mt$ is a multi-type if $\$mt = (t^{+})$. Let $\$\$mt$ be a set of multi-types. An object $\$mv$ is a multi-value if $\$mv = (v^{+})$. Let $\$\$mv$ be a set of multi-values.

A structure $\$cs$ is an absolutely typed structure if $\$cs = \$v{:}{:}\$mt$. The operation $\ldots{:}{:}(\ldots)$ is called an absolute typification operation. Let $\$\$atcs$ be a set of absolutely typed structures.

A structure $\$t$ is an absolute type in $[\![\$atcs]\!]$ if $\$atcs = \$v{:}{:}\$mt$, and $\$t \in \$mt$ for some $\$v$ and $\$mt$. A structure $\$atcs$ has an absolute type $\$t$ if $\$t$ is an absolute type in $[\![\$atcs]\!]$. A structure $\$v$ is a value in $[\![atcs]\!]$ if $\$atcs = \$v{:}{:}\$mt$ for some $\$mt$.

A structure $\$mt$ is an absolute multi-type in $[\![\$atcs]\!]$ if $\$atcs = \$v{:}{:}\$mt1$, and $\$mt \subseteq \$mt1$ for some $\$v$ and $\$mt1$. A structure $\$atcs$ has an absolute multi-type $\$mt$ if $\$mt$ is an absolute multi-type in $[\![\$atcs]\!]$.

The absolute typification operation categorizes structures, using absolute types as category names and absolute multi-types as category unions. It also models instance constructors for these categories. For example, the structure $"division\ by\ zero"{:}{:}(exception)$ specifies the value (instance) of the type (category) $exception$ as the result of application of the instance constructor $::\ (exception)$ to the argument $"division\ by\ zero"$.

A structure $\$cs$ is a relatively typed structure if $\$cs = \$v{:}\,\$mt$. The operation $\ldots{:}\,(\ldots)$ is called a relative typification operation. Let $\$\$rtcs$ be a set of relatively typed structures.

A structure $\$t$ is a relative type in $[\![\$v,\ (\$cs^{*})]\!]$ if $\$cs^{*} = \$cs^{*}1\ \$v{:}\,\$mt\ \$cs^{*}2$, and $\$t \in \$mt$ for some $\$cs^{*}1$, $\$cs^{*}2$ and $\$mt$. A structure $\$t$ is a relative type in $[\![\$ccs]\!]$ if $\$t$ is a relative type in $[\![\$v,\ \$ccs]\!]$ for some $\$v$. A structure $\$v$ has a relative type $\$t$ in $[\![\$ccs]\!]$ if $\$t$ is a relative type in $[\![\$v,\ \$ccs]\!]$. A structure $\$ccs$ has a relative type $\$t$ if $\$t$ is a relative type in $[\![\$ccs]\!]$. A structure $\$v$ is a value in $[\![\$rtcs]\!]$ if $\$rtcs = \$v{:}\,\$mt$ for some $\$mt$. A structure $\$v$ is a value in $[\![\$t, \$rtcs]\!]$ if $\$cs^{*} = \$cs^{*}1\ \$v{:}\,\$mt\ \$cs^{*}2$, and $\$t \in \$mt$ for some $\$cs^{*}1$, $\$cs^{*}2$ and $\$mt$.

A structure $\$mt$ is a relative multi-type in $[\![\$v,\ (\$cs^{*})]\!]$ if $\$cs^{*} = \$cs^{*}1\ \$v{:}\,\$mt1\ \$cs^{*}2$, and $\$mt \subseteq \$mt1$ for some $\$cs^{*}1$, $\$cs^{*}2$ and $\$mt$. A structure $\$mt$ is a relative multi-type in $[\![\$ccs]\!]$ if $\$mt$ is a relative multi-type in $[\![\$v,\ \$ccs]\!]$ for some $\$v$. A structure $\$v$ has a relative multi-type $\$mt$ in $[\![\$ccs]\!]$ if $\$mt$ is a relative multi-type in $[\![\$v,\ \$ccs]\!]$. A structure $\$ccs$ has a relative multi-type $\$mt$ if $\$mt$ is a relative multi-type in $[\![\$ccs]\!]$. A structure $\$v$ is a value in $[\![\$mt, \$rtcs]\!]$ if $\$cs^{*} = \$cs^{*}1\ \$v{:}\,\$mt1\ \$cs^{*}2$, and $\$mt \subseteq \$mt1$ for some $\$cs^{*}1$, $\$cs^{*}2$ and $\$mt$.

The relative typification operation categorizes elements of compound structures using relative types as category names and relative multi-types as category unions.

A structure $\$cs$ is typed if $\$cs$ is relatively typed, or $\$cs$ is absolutely typed. Let $\$\$tcs$ be a set of typed structures.

Conceptual transition systems are transition systems that have elements, and in which elements and states are conceptual structures.

Let $\$\$s$ be a set of objects called states. A subset of the set $\$\$s \times \$\$s$ is called a transition relation. Let $\$\$tr$ be a set of transition relations. A pair $(\$\$s, \hookrightarrow)$ is a transition system if $\hookrightarrow \in \$\$tr$.

An object $\$cts$ of the form $(\$\$ato,\ \$\$is, \hookrightarrow)$ is a conceptual transition system if $(\$\$cs[\![\$ato]\!], \hookrightarrow)$ is a transition system, and $\$\$is \subseteq \$\$cs$. The elements of $\$\$ato$ and $\$\$is$ are called atoms and initial states in $[\![\$cts]\!]$. The relation $\hookrightarrow$ is called a transition relation in $[\![\$cts]\!]$. Let $\$\$cts$ be a set of CTSs. The sets $\$\$s$ and $\$\$e$ of states and elements in $[\![\$\$cts]\!]$ are defined as follows: $\$\$s = \$\$e = \$\$cs$.

Let $[.\ \$mt]$, $[.\ \$mt \coloneqq \$v]$ and $[.\ \$mt \coloneqq]$ denote $[\$s\ .\ \$mt]$, $[\$s\ .\ \$mt \coloneqq \$v]$ and $[\$s\ .\ \$mt \coloneqq]$ for the current state $\$s$.

4. The CTSL language

Let $\$\$sa$ be a set of syntactic constructs called special atoms.

The CTSL language is a basic language of CTSs. It only defines the syntax of conceptual structures and does not concretize the set $\$\$sa$ and the transition relation $\hookrightarrow$. The extensions of CTSL for the special kinds of CTSs use the CTSL syntax and concretize $\$\$s$a and $\hookrightarrow$.

The set $\$\$ato$ of atoms in CTSL is defined as follows:

- if $\$o$ is a sequence of Unicode symbols except for the whitespace symbols and the symbols ", (,), {, }, ;, ,, and :, then $\$o \in \$\$ato$;
- $\$\$sa \subseteq \$\$ato$;
- if $\$o$ has the form "$\$o1$", $\$o1$ is a sequence of Unicode symbols, each occurrence of the symbol " in $\$o1$ is preceded by the symbol \, and each occurrence of the symbol \ in $\$o1$ is doubled, then $\$o \in \$\$ato$. In this case, the atom $\$o$ is called a string.

The set $\$\$cs$ of conceptual structures in CTSL is defined as follows:

- $\$ato \in \$\$cs$;
- $(\$cs^*) \in \$\$cs$;
- if the elements of $\$cs^+$ are pairwise distinct, and \$cs $\neq und$, then $\$cs\text{: }(\$cs^+) \in \$\$s$;
- if the elements of $\$cs^+$ are pairwise distinct, and \$cs $\neq und$, then $(\$cs^+)\text{::}\,\$cs \in \$\$cs$.

The whitespace symbols, comma (,) and the semicolon (;) are interchangeable in compound structures in CTSL. For example, $(\$cs1,\ \$cs2)$, $(\$cs1; \$cs2)$ and $(\$cs1\ \$cs2)$ represent the same conceptual structure.

The bracket pairs (,) and {, } are interchangeable in compound structures in CTSL. For example, $(\$cs^*)$ and $\{\$cs^*\}$ represent the same compound conceptual structure.

5. The basic operations on conceptual structures

The conceptual structure access operation $[\$cs\,.\,\$mt]$ makes selection of elements of a compound structure in accordance with their relative types. It is defined as follows:

- if $\$cs = \$v{:}\,\$mt1$, and $\$mt \subseteq \$mt1$, then $[\$cs\,.\,\$mt] = \$v$;
- if $\$cs \in \$\$ccs$, and there exists only one element $\$cs1$ of $\$cs$ such that $\$cs1 = \$v{:}\,\$mt1$, and $\$mt \subseteq \$mt1$, then $[\$cs\,.\,\$mt] = \$v$;
- if $\$cs \in \$\$ccs$, $\$n > 1$, $\$cs1, \ldots, \$cs\$n$ are (ordered from left to right) elements of $\$cs$ such that $\$cs\$n1 = \$v\$n1{:}\,\$mt\$n1$, and $\$mt \subseteq \$mt\$n1$ for each $1 \le \$n1 \le \$n$, then $[\$cs\,.\,\$mt] = (\$v1 \ldots \$v\$n){::}\,(multivalue)$;
- otherwise, $[\$cs\,.\,\$mt] = und$.

An element $\$v$ is a value in $[\![\$mt, \$cs]\!]$ if $\$v = [\$cs\,.\,\$mt]$. The value of the form $\$mv{::}\,(multivalue)$ is called a multi-value. Let $[support\ \$cs]$ denote $\{\$mt \mid [\$cs\,.\,\$mt] \ne und\}$.

A structure $\$t$ is an (single-valued) attribute in $[\![\$cs]\!]$ if $[\$cs\,.\,(\$t)]$ does not have the absolute type $multivalue$. A structure $\$t$ is a multi-valued attribute in $[\![\$cs]\!]$ if $[\$cs\,.\,(\$t)]$ has the absolute type $multivalue$. Let $\$\$att$ and $\$\$mvatt$ be sets of attributes and multi-valued attributes.

A structure $\$cs$ is an (single-valued) attribute structure if $\$t$ is an attribute in $[\![\$cs]\!]$ for each $\$t \in \$\$cs$. A structure $\$ccs$ is a multi-valued attribute structure if $\$t$ is a multi-valued attribute in $[\![\$ccs]\!]$ for some $\$t \in \$\$cs$. Let $\$\$as$ and $\$\$mvas$ be sets of attribute structures and multi-valued attribute structures.

For example, the conceptual structure

$(x{:}\,\{variable\}\ int{:}\,\{(type\ x)\}\ 3{:}\,\{(value\ x)\}\ y{:}\,\{variable\}\ bool{:}\,\{(type\ y)\}\ true{:}\,\{(value\ y)\})$

defines the variables x and y by the multi-attribute $variable$, the types int and $bool$ of these variables by the parametric attribute $(type\ \$va)$, where the values of the parameter $\$va$ are variables, and the values 3 and $true$ of these variables by the parametric attribute $(value\ \$va)$.

A structure $\$mt$ is a (single-valued) multi-attribute in $[\![\$cs]\!]$ if $[\$cs\,.\,\$mt]$ does not have the absolute type $multivalue$. A structure $\$mt$ is a multi-valued multi-attribute in $[\![\$cs]\!]$ if $[\$cs\,.\,\$mt]$ has the absolute type $multivalue$. Let $\$\$matt$ and $\$\$mvmatt$ be sets of multi-attributes and multi-valued multi-attributes.

A structure $\$cs$ is an (single-valued) multi-attribute structure if $\$mt$ is a multi-attribute in $[\![\$cs]\!]$ for each $\$mt \in \{(t^+) | t^+ \in \$\$cs^+\}$. A structure $\$ccs$ is a multi-valued multi-attribute structure if

$\$mt$ is a multi-valued multi-attribute in $[\![\$ccs]\!]$ for some $\$mt \in \{(t^+)|t^+ \in \$\$cs^+\}$. Let $\$\$mas$ and $\$\$mvmas$ be sets of mult-attribute structures and multi-valued multi-attribute structures.

The conceptual structure update operation $[\$cs\,.\,\$mt \coloneqq \$v]$ replaces all values in $[\![\$mt,\ \$cs]\!]$ in $\$cs$ by $\$v$ from left to right and deletes these values in case when $\$v = und$. It is defined as follows (the first proper rule is applied):

- if $mt is not a relative multi-type in $[\![(\$cs^*)]\!]$, and $\$v \neq und$, then
  $[(\$cs^*)\,.\,\$mt \coloneqq \$v] = (\$cs^*\ \$v{:}\,\$mt)$;
- if $mt is not a relative multi-type in $[\![\$cs]\!]$, and $\$v \neq und$, then
  $[\$cs\,.\,\$mt \coloneqq \$v] = (\$cs\ \$v{:}\,\$mt)$;
- if $\$mt \sqsubseteq \$mt1$, and $\$v \neq und$, then $[\$v1{:}\,\$mt1\,.\,\$mt \coloneqq \$v] = \$v{:}\,\$mt1$;
- if $\$mt \sqsubseteq \$mt1$, then $[\$v1{:}\,\$mt1\,.\,\$mt \coloneqq und] = und$;
- $[(\$cs^*)\ .\ \$mt \coloneqq \$v] = ([\$cs^*\ .{:}{:}\{seq\}\ \$mt \coloneqq \$v])$;
- $[\$cs\,.\,\$mt \coloneqq \$v] = \$cs$.

The conceptual structure update operation $[\$cs^*\ .{:}{:}\{seq\}\ \$mt \coloneqq \$v]$ is defined as follows (the first proper rule is applied):

- if $\$mt \sqsubseteq \$mt1$, and $\$v \neq und$, then
  $[\$v1{:}\,\$mt1\ \$cs^*\ .{:}{:}\{seq\}\ \$mt \coloneqq \$v] = \$v{:}\,\$mt1\ [\$cs^*\ .{:}{:}\{seq\}\ \$mt \coloneqq \$v]$;
- if $\$mt \sqsubseteq \$mt1$, then
  $[\$v1{:}\,\$mt1\ \$cs^*\ .{:}{:}\{seq\}\ \$mt \coloneqq und] = [\$cs^*\ .\ {:}{:}\{seq\}\ \$mt \coloneqq und]$;
- $[\$cs\ \$cs^*\ .{:}{:}\{seq\}\ \$mt \coloneqq \$v] = \$cs\ [\$cs^*\ .{:}{:}\{seq\}\ \$mt \coloneqq \$v]$;
- $[[es]\ .{:}{:}\{seq\}\ \$mt \coloneqq \$v] = [es]$.

The conceptual structure update operation $[\$cs\,.\,\$mt1 \coloneqq \$v1\ \ldots\ \$mt\$n \coloneqq \$v\$n]$ is defined as follows:

- $[\$cs\,.\,\$mt \coloneqq \$v\ \$se] = [[\$cs\,.\$mt \coloneqq \$v]\ \$se]$;
- $[\$cs\,.\,[es]] = \$cs$.

The conceptual structure update operation $[\$cs\,.\,\$mt \coloneqq]$ is a shortcut for $[\$cs\,.\,\$mt \coloneqq und]$.

6. The properties of conceptual transition systems

An element $\$tra$ of the form $(\$s1,\ \$s2)$ is called a transition. The states $\$s1$ and $\$s2$ are called input and output states in $[\![tra]\!]$. Let $\$\$tra$ be a set of transitions.

A state $\$s1$ is final if there is no $\$s2$ such that $\$s1 \hookrightarrow \$s2$. Let $\$\$fs$ be a set of final states in $[\![\$cts]\!]$. A system $\$cts$ stops in $[\![\$s]\!]$ if $\$s$ is final.

A state $\$s$ is reachable if there exist $\$n > 0$, $\$s1$, …, $\$s\$n$ such that $\$s1 \in \$is$, $\$s\$n1 \hookrightarrow \$s[\$n1 + 1]$ for each $1 \leq \$n1 < \$n$, and $\$s = \$s\$n$. Let $\$\$rs$ be a set of reachable states in $[\![\$cts]\!]$.

An element \$t is an attribute in $[\![\$cts]\!]$ if \$t is an attribute in $[\![\$s]\!]$ for each $\$s \in \$\$rs[\![\$cts]\!]$. An element \$t is a multi-valued attribute in $[\![\$cts]\!]$ if there exists $\$s \in \$\$rs[\![\$cts]\!]$ such that \$t is a multi-valued attribute in $[\![\$s]\!]$. An element \$mt is a multi-attribute in $[\![\$cts]\!]$ if \$mt is a multi-attribute in $[\![\$s]\!]$ for each $\$s \in \$\$rs[\![\$cts]\!]$. An element \$mt is a multi-valued multi-attribute in $[\![\$cts]\!]$ if there exists $\$s \in \$\$rs[\![\$cts]\!]$ such that \$mt is a multi-valued multi-attribute in $[\![\$s]\!]$.

A system $\$cts$ is a CTS with return values if $value$ is an attribute in $[\![\$cts]\!]$. An element $\$v$ is a value in $[\![\$s]\!]$ if $\$v = [\$s\,.\,\{value\}]$. An element $\$v$ is a value in $[\![tra]\!]$ if $[\$tra\,..\,1] \hookrightarrow [\$tra\,..\,2]$, and $\$v$ is a value in $[\![[\$tra\,..\,2]]\!]$. A transition $\$tra$ returns a value $\$v$ if $\$v$ is a value in $[\![\$tra]\!]$. An element $\$v$ is undefined if $\$v = und$. The set $\$\$v$ of (possible) values is defined as follows: $\$\$v = \$\$e$.

A system $\$cts$ with return values can return exceptions. A value $\$v$ is an exception (an exceptional value) if $\$v$ has the absolute type $exception$. Thus, exceptions are specified by the absolute type $exception$. Let exc be a shortcut for $exception$. Let $\$\$ex$ be a set of exceptions. An element $\$t$ is called a type in $[\![\$ex]\!]$ if $\$t = [\$v\,.\,\{type\}]$, where $\$v$ is a value in $[\![\$ex]\!]$.

A value $\$v$ is abnormal if $\$v$ is undefined, or $\$v$ is an exception. Let $\$\$av$ be a set of abnormal values. A value $\$v$ is normal if $\$v \notin \$\$av$. Let $\$\$nv$ be a set of normal values. A transition $\$tra$ returns (generates) an exception $\$ex$ if $\$ex$ is a value in $[\![\$tra]\!]$. A transition $\$tra$ is normally executed if $\$tra$ does not return exceptions.

A system $\$cts$ is a CTS with programs if $program$ is an attribute in $[\![\$cts]\!]$, and the value of this attribute is a compound structure. A compound structure $\$p$ is a program in $[\![\$s]\!]$ if $\$p = [\$s\,.\,\{program\}]$. Let $\$\$p$ be a set of programs. A program in $[\![\$s]\!]$ is empty if $[\$s\,.\,\{program\}] = ()$. A program in $[\![\$s]\!]$ initiates transitions from $\$s$.

The elements that initiate transitions are called executable elements. Let $\$\$ee$ be a set of executable elements. A program \$p is an executable element, and the elements of $[\![\$p]\!]$ are executable elements.

A system $\$cts$ is a CTS with direct stop if $stop$ is an attribute in $[\![\$cts]\!]$, and $\$s$ is final for each $\$s$ such that $[\$s\,.\,\{stop\}] \neq und$. A state \$s is a stop state if $[\$s\,.\,\{stop\}] \neq und$. The value of the attribute $stop$ specifies why the system $\$cts$ stopped.

An attribute $\$bi$ in $[\![\$s]\!]$ is a backtracking invariant in $[\![\$s]\!]$ if $[\$s\,.\,\{((backtracking\ invariant)\ \$bi)\}] \neq und$. An attribute $\$bi$ is a backtracking invariant in

$[\![\$cts]\!]$ if $\$bi$ is a backtracking invariant in $[\![\$s]\!]$ for some $\$s \in \$\$rs[\![\hookrightarrow [\![\$cts]\!]]\!]$. Backtracking invariants preserves their values after backtracking. Let $\$\$bi$ be a set of backtracking invariants.

Let $[(propagate\ backtracking\ invariants)\ \$s1\ \$s2]$ denote the state $\$s$ such that $[\$s\,.\,\{\$att\}] = [\$s1\,.\,\{\$att\}]$ for each $\$att \in \$\$bi[\![\$s2]\!]$, and $[\$s\,.\,\{\$att\}] = [\$s2\,.\,\{\$att\}]$ for each $\$att \notin \$\$bi[\![\$s2]\!]$.

Let $\$e^*\ \#\ \$v\ \#\ \$s$ and $\$e^*\ \#\ \$s$ denote $[\$s\,.\,\{program\} := (\$e^*),\ \{value\} := \$v]$ and $[\$s\,.\,\{program\} := (\$e^*)]$.

A system $\$cts$ is a CTS with backtracking in $[\![\$\$bi]\!]$ if $\$cts$ is a CTS with return values with programs, $((backtracking\ invariant)\ \$bi)$ is a parametric attribute in $[\![\$cts]\!]$, $\$bi$ is a backtracking invariant in $[\![\$cts]\!]$ for each $\$bi \in \$\$bi$, and $\hookrightarrow [\![\$cts]\!]$ satisfies the following properties:

- if $\$v[\![\$s]\!] \neq und$, then $(backtracking\ \$s1\ \$e^*1)\ \$e^*\ \#\ \$s\ \hookrightarrow \$e^*\ \#\ \$s$;
- if $\$v[\![\$s]\!] = und$, then
  $(backtracking\ \$s1\ \$e^*1)\ \$e^*\ \#\ \$s\ \hookrightarrow$
  $\$e^*1\ \$e^*\ \#\ [(propagate\ backtracking\ invariants)\ \$s1\ \$s]$.

The element $\$e$ of the form $(backtracking\ \$s\ \$e^*)$ is called a backtracking point. The objects $\$s$ and $\$e^*$ are called a state and a program prefix in $[\![\$e]\!]$.

7. Examples of conceptual models of programming languages

Let $\$\$l$ be a set of programming languages. Let $[am\ \$l]$ denotes an abstract machine executing the constructs of $\$l$. A tuple $(\$\$t,\ \$\$v,\ \$\$ex, \$\$s,\ \$\$c, \$\$ax)$ is a conceptual model of $\$l$ in CTSL if $\$\$t[\![\$l]\!]$ is a set of elements in CTSL representing the types of $\$l$, $\$\$v[\![\$l]\!]$ is a set of elements in CTSL representing the values in $[am\ \$l]$ (in particular, the values of the types of $\$l$), $\$\$\mathrm{ex}[\![\$l]\!]$ is a set of exceptions in CTSL representing the exceptions in $[am\ \$l]$, $\$\$\mathrm{ex}[\![\$l]\!] \sqsubseteq \$\$v[\![\$l]\!]$, $\$\$s[\![\$l]\!]$ is a set of states in CTSL representing the states of $[am\ \$l]$, $\$\$c[\![\$l]\!]$ is a set of executable elements in CTSL representing the executable constructs of $[am\ \$l]$ (in particular, the elements of programs in $\$l$), and $\$\$ax$ is a set of axioms representing the constraints for the conceptual model of $\$l$ (the other elements of the tuple).

Let $[content\ \$t]$ denote the set of values in $[\![\$t]\!]$. The set $[content\ \$t]$ is called the content in $[\![\$t]\!]$. The fact that $\$\$t$ and $\$t$ depend on $\$s$ is denoted by $\$\$t[\![\$s]\!]$ and $\$t[\![\$s]\!]$.

Let **Axiom:** $\$ax$ denote that $\$ax$ is an axiom of the conceptual model of $\$l$.

The family of model programming languages (MPLs) is described and their conceptual models are defined in this section.

7.1. MPL1: types, typed variables and basic statement

The MPL1 language is an extension of CTSL that adds types, typed variables, the variable access operation, and the basic statements such as variable declarations, variable assignments, if statements, while statements and block statements.

7.1.1. Types, values, states

For MPL1, $\$\$t[\![\text{MPL1}]\!] = \{int,\ nat\}$, $\$\$v[\![\text{MPL1}]\!] = \$\$i \cup \$\$n$, and $\$\$ex[\![\text{MPL1}]\!] = \emptyset$, where $[content\ int] = \$\$i$, and $[content\ nat] = \$\$n$.

An element $\$e$ is a name if $\$e$ is normal. Let $\$\$na$ be a set of names.

The attribute $(variable\ \$na)$ specifies variables in MPL1. A name $\$na$ is a variable in $[\![\$s]\!]$ if $[\$s\ .\ \{(variable\ \$na)\}] \neq und$. Let $\$\$va$ be a set of variables.

The attribute $(type\ \$va)$ specifies the type of the variable $\$va$. A type $\$t$ is a type in $[\![\$va,\ \$s]\!]$ if $[\$s\ .\ \{(type\ \$va)\}] = \$t$.

Axiom: If $[\$s\ .\ \{(variable\ \$va)\}] \neq und$, then $[\$s\ .\ \{(type\ \$va)\}] \in \$\$t[\![\$s]\!]$.

The attribute $(value\ \$va)$ specifies the value of the variable $\$va$. A value $\$v$ is a value in $[\![\$va,\ \$s]\!]$ if $[\$s\ .\ \{(value\ \$va)\}] = \$v$.

Axiom: If $[\$s\ .\ \{(value\ \$va)\}] \neq und$, then $[\$s\ .\ \{(type\ \$va)\}] = \$t$, and $[\$s\ .\ \{(value\ \$va)\}] \in [content\ \$t]$ for some $\$t \in \$\$t[\![\$s]\!]$.

7.1.2. Constructs

The MPL1 program is represented by the element $(program\ \$na\ \$c^*)$. It specifies a program with the name $\$na$ and the body $\$c^*$.

The variable declaration is represented by the element $(var\ \$va\ \$t)$. It declares the variable $\$va$ of the type $\$t$.

Axiom: Variable declarations are elements of the program body.

The variable access operation is represented by $\$va$. It returns the value of the variable $\$va$.

The variable assignment is represented by the element $(\$va \backslash{:=}\ \$c)$. If $\$v$ is a value of $\$c$, then it assigns $\$v$ to the variable $\$va$.

The block statement is represented by the element $(block\ \$c^*)$. It specifies the block statement with the body $\$c^*$.

The if statement is represented by the element $(\backslash if\ \$c\ then\ \$c^*1\ else\ \$c^*2)$. It specifies the if statement with the condition $\$c$, the then-branch $\$c^*1$ and the else-branch $\$c^*2$. The element $(\backslash if\ \$c\ then\ \$c^*1)$ is a shortcut for $(\backslash if\ \$c\ then\ \$c^*1\ else)$.

The while statement is represented by the element $(\backslash while\ \$c\ do\ \$c^*)$. It specifies the while statement with the condition $\$c$ and the body $\$c^*$.

7.2. MPL2: variable scopes

The MPL2 language is an extension of MPL1 that adds the variable scopes feature.

The relative scope of the variable $\$va$ occuring in the element $\$c$ is the number of blocks surrounding this occurrence of $\$va$ in $\$c$. The value and type of $\$va$ depend on its scope. The variable $\$va$ can be global (with the scope 0) and local. The following example illustrates variable scopes:

```
(program scopes // x = und, y = und, scope = 0
 (var x int) // x = und, y = und, scope = 0
 (x := 0) // x = 0, y = und, scope = 0
 (var y bool) // x = 0, y = und, scope = 0
 (y := true) // x = 0, y = true, scope = 0
 (block // x = 0, y = true, scope = 1
   (var x bool) // x = und, y = true, scope = 1
   (x := false) // x = false, y = true, scope = 1
   (block // x = false, y = true, scope = 2
    (var x int) // x = und, y = true, scope = 2
    (x := 2) // x = 2, y = true, scope = 2
   ) // x = false, y = true, scope = 1
   (var y int) // x = false, y = und, scope = 1
   (y := 1) // x = false, y = 1, scope = 1
 ) // x = 0, y = true, scope = 0
   ).
```

7.2.1. Types, values, states

For MPL2, $\$\$t[\![\text{MPL2}]\!] = \$\$t[\![\text{MPL1}]\!]$, $\$\$v[\![\text{MPL2}]\!] = \$\$v[\![\text{MPL1}]\!]$, and $\$\$ex[\![\text{MPL2}]\!] = \emptyset$.

Let $\$\$sc$ be a set of (relative) variable scopes.

The attribute $(variable\ \$na\ \$sc)$ specifies variables in $[\![\$sc]\!]$. A name $\$na$ is a variable in $[\![\$s,\ \$sc]\!]$ if $[\$s\ .\ \{(variable\ \$na\ \$sc)\}] \neq und$.

A variable $\$va[\![\$s,\ \$sc]\!]$ is global if $\$sc = 0$. A variable $\$va[\![\$s,\ \$sc]\!]$ is local if $\$sc > 0$.

The attribute $(current\ scope)$ specifies the scope of the current block. A scope $\$sc$ is a current scope in $[\![\$s]\!]$ if $[\$s\ .\ \{(current\ scope)\}] = \$sc$. A name $\$na$ is a variable in $[\![\$s]\!]$ if $\$na$ is a variable in $[\![\$s, \$sc]\!]$ for some $0 \leq \$sc \leq [\$s\ .\ \{(current\ scope)\}]$. A scope $\$sc$ is a scope in $[\![\$va, \$s]\!]$ if

$\$va$ is a variable in $[\![\$s, \$sc]\!]$, $0 \leq \$sc \leq [\$s\,.\,\{(current\ scope)\}]$, and $\$va$ is not a variable in $[\![\$s, \$sc1]\!]$ for each $\$sc < \$sc1 \leq [\$s\,.\,\{(current\ scope)\}]$.

The attribute $(type\ \$va\ \$sc)$ specifies the type of the variable $va in $[\![\$sc]\!]$. A type $\$t$ is a type in $[\![\$va, \$s, \$sc]\!]$ if $[\$s\,.\,\{(type\ \$va\ \$sc)\}] = \$t$. A type $\$t$ is a type in $[\![\$va, \$s]\!]$ if $\$t$ is a type in $[\![\$va, \$s, \$sc]\!]$, where $\$sc$ is a scope in $[\![\$va, \$s]\!]$.

Axiom: If $[\$s\,.\,\{(variable\ \$va\ \$sc)\}] \neq und$, then $[\$s\,.\,\{(type\ \$va\ \$sc)\}] \in \$\$t[\![\$s]\!]$.

The attribute $(value\ \$va\ \$sc)$ specifies the value of the variable $\$va$ in $[\![\$sc]\!]$. A value $\$v$ is a value in $[\![\$va, \$s, \$sc]\!]$ if $[\$s\,.\,\{(value\ \$va\ \$sc)\}] = \$v$. A value $\$v$ is a value in $[\![\$va, \$s]\!]$ if $\$v$ is a value in $[\![\$va, \$s, \$sc]\!]$, where $\$sc$ is a scope in $[\![\$va, \$s]\!]$.

Axiom: If $[\$s\,.\,\{(value\ \$va\ \$sc)\}] \neq und$, then $[\$s\,.\,\{(type\ \$va\ \$sc)\}] = \$t$, and $[\$s\,.\,\{(value\ \$va\ \$sc)\}] \in [content\ \$t]$ for some $\$t \in \$\$t[\![\$s]\!]$.

7.2.2. Constructs

For MPL2, $\$\$c[\![\text{MPL2}]\!] = \$\$c[\![\text{MPL1}]\!]$.

Axiom: Variable declarations are elements of the program body or of block bodies.

7.3. MPL3: functions

The MPL3 language is an extension of MPL2 that adds the functions feature: declarations and calls of functions, and the return statement.

Axiom: Function overloading is prohibited.

7.3.1. Types, values, states

The exception $(return{:}\{type\},\ v{:}\{value\}){::}\{exc\}$ specifies the execution of the return statement with the return value v. Let $\$\$ex1$ be a set of such exceptions.

For MPL2, $\$\$t[\![\text{MPL3}]\!] = \$\$t[\![\text{MPL2}]\!]$, $\$\$v[\![\text{MPL3}]\!] = \$\$v[\![\text{MPL2}]\!] \cup \$\$ex[\![\text{MPL3}]\!]$, and $\$\$ex[\![MPL3]\!] = \$\$ex1$.

The attribute $(function\ \$na)$ specifies functions. A name $\$na$ is a function in $[\![\$s]\!]$ if $[\$s\,.\,\{(function\ \$na)\}] \neq und$. Let $\$\$f$ be a set of functions.

The attribute $(arity\ \$f)$ specifies the arity of the function $\$f$. A number $\$n$ is an arity in $[\![\$f, \$s]\!]$ if $[\$s\,.\,\{(arity\ \$f)\}] = \$n$.

The attribute $(argument\ \$f\ \$n)$ specifies the $\$n$-th argument of the function $\$f$. A name $\$na$ is an argument in $[\![\$f, \$n]\!]$ if $[\$s\,.\,\{(argument\ \$f\ \$n)\}] = \$na$, and $1 \leq \$n \leq [\$s\,.\,\{(arity\ \$f)\}]$. Let $\$\$a$ be a set of arguments.

The attribute $((argument\ type)\ \$f\ \$n)$ specifies the type of the $\$n$-th argument of the function $\$f$. A type $\$t$ is a type in $[\![\$f, \$n]\!]$ if $[\$s\ .\ \{((argument\ type)\}\ \$f\ \$n)] \neq und$, and $1 \leq \$n \leq [\$s\ .\ \{(arity\ \$f)\}]$.

The attribute $((return\ type)\ \$f)$ specifies the return type of the function $\$f$. A type $\$t$ is a return type in $[\![\$f]\!]$ if $\left[\$s\ .\ \left\{\left((return\ type)\$f\right)\right\}\right] = \$t$.

The attribute $(body\ \$f)$ specifies the body of the function $\$f$. A sequence $\$c^*$ is a body in $[\![\$f]\!]$ if $[\$s\ .\ \{(body\ \$f)\}] = (\$c^*)$.

A call level is a number of embedded function calls. Let $\$\$cl$ be a set of call levels. The attribute $(current\ call\ level)$ specifies the current call level. A level $\$cl$ is a current call level in $[\![\$s]\!]$ if $\$cl = [\$s\ .\ \{(current\ call\ level)\}]$.

The $(current\ return\ type)$ specifies the return type in the current function call. A type $\$t$ is a current return type in $[\![\$s]\!]$ if $[\$s\ .\ \{(current\ return\ type)\}] = \$t$.

The attribute $(variable\ \$na\ \$sc\ \$cl)$ specifies variables in $[\![\$sc, \$cl]\!]$. A name $\$na$ is a variable in $[\![\$s, \$sc, \$cl]\!]$ if $[\$s\ .\ \{(variable\ \$na\ \$sc\ \$cl)\}] \neq und$, and $\$sc = 0$ implies $\$cl = 0$.

A variable $\$va[\![\$s, \$sc, \$cl]\!]$ is global if $\$sc = 0$, and $\$cl = 0$. A variable $\$va[\![\$s, \$sc, \$cl]\!]$ is local if $\$sc > 0$, and $\$cl > 0$.

A name $\$na$ is a variable in $[\![\$s, \$cl]\!]$ if $\$na$ is a variable in $[\![\$s, \$sc,\ \$cl]\!]$ for some $0 \leq \$sc \leq [\$s\ .\ \{(current\ scope)\}]$. A scope $\$sc$ is a scope in $[\![\$va, \$s,\ \$cl]\!]$ if $\$va$ is a variable in $[\![\$s, \$sc,\ \$cl]\!]$, $0 \leq \$sc \leq [\$s\ .\ \{(current\ scope)\}]$, and $\$va$ is not a variable in $[\![\$s,\ \$sc1, \$cl]\!]$ for each $\$sc < \$sc1 \leq [\$s\ .\ \{(current\ scope)\}]$.

The attribute $(type\ \$va\ \$sc\ \$cl)$ specifies the type of the variable $\$va$ in $[\![\$sc,\ \$cl]\!]$. A type $\$t$ is a type in $[\![\$va, \$s, \$sc, \$cl]\!]$ if $[\$s\ .\ \{(type\ \$va\ \$sc\ \$cl)\}] = \$t$.

Axiom: If $[\$s\ .\ \{(variable\ \$va\ \$sc\ \$cl)\}] \neq und$, then $[\$s\ .\ \{(type\ \$va\ \$sc\ \$cl)\}] \in \$\$t[\![\$s]\!]$.

A type $\$t$ is a type in $[\![\$va, \$s, \$cl]\!]$ if $\$t$ is a type in $[\![\$va, \$s, \$sc, \$cl]\!]$, and $\$sc$ is a scope in $[\![\$va, \$cl]\!]$ for some $0 \leq \$sc \leq [\$s\ .\ \{(current\ scope)\}]$. A type $\$t$ is a type in $[\![\$va, \$s, \$cl]\!]$ if $\$t$ is a type in $[\![\$va, \$s, \$sc, \$cl]\!]$, where $\$sc$ is a scope in $[\![\$va, \$s, \$cl]\!]$.

The attribute $(value\ \$va\ \$sc\ \$cl)$ specifies the value of the variable $\$va$ in $[\![\$sc,\ \$cl]\!]$. A value $\$v$ is a value in $[\![\$va, \$s, \$sc]\!]$ if $[\$s\ .\ \{(value\ \$va\ \$sc)\}] = \$v$. A value $\$v$ is a value in $[\![\$va, \$s,\ \$cl]\!]$ if $\$v$ is a value in $[\![\$va, \$s, \$sc,\ \$cl]\!]$, where $\$sc$ is a scope in $[\![\$va, \$s,\ \$cl]\!]$.

Axiom: If $[\$s\ .\ \{(value\ \$va\ \$sc\ \$cl)\}] \neq und$, then $[\$s\ .\ \{(type\ \$va\ \$sc\ \$cl)\}] = \$t$, and $[\$s\ .\ \{(value\ \$va\ \$sc\ \$cl)\}] \in [content\ \$t]$ for some $\$t \in \$\$t[\![\$s]\!]$.

7.3.2. Constructs

An object $\$o$ is a typed name if $\$o = \$na\ \$t$. Let $\$\$tna$ be a set of typed names.

The function declaration is represented by the element $(function\ \$f\ (\$tna1\ \dots\ \$tna\$n)\ \$t\ \$c^*)$. It specifies the declaration of the function $\$f$, the arguments $\$na[\![\$tna1]\!]$, ..., $\$na[\![\$tna\$n]\!]$ of the types $\$t[\![\$tna1]\!]$, ..., $\$t[\![\$tna\$n]\!]$, the return type $\$t$, and the body $\$c^*$.

Axiom: Function declarations are elements of the program body.

The return statement is represented by the element $(return\ \$c)$. It specifies the return statement with the return element $\$c$. If $\$v$ is a value of $\$c$, then it returns $\$v$.

The function call is represented by the element $(call\ \$f\ \$c^*)$. It specifies the call of the function $\$f$ with the arguments $\$c^*$.

7.4. MPL4: procedures

The MPL4 language is an extension of MPL3 that adds the procedures feature: declarations and calls of procedures, and the exit statement.

Axiom: Procedure overloading is prohibited.

Axiom: The sets of function names and procedure names are disjoint.

7.4.1. Types, values, states

The exception $(exit{:}\{type\}){::}\{exc\}$ specifies the execution of the exit statement. Let $\$\$ex1$ be a set of such exceptions.

For MPL4, $\$\$t[\![\text{MPL4}]\!] = \$\$t[\![\text{MPL3}]\!]$, $\$\$v[\![\text{MPL4}]\!] = \$\$v[\![\text{MPL3}]\!] \cup \$\$ex[\![\text{MPL4}]\!]$, and $\$\$ex[\![MPL4]\!] = \$\$ex[\![MPL3]\!] \cup \$\$ex1$.

The attribute $(procedure\ \$na)$ specifies procedures. A name $\$na$ is a procedure in $[\![\$s]\!]$ if $[\$s\ .\ \{(procedure\ \$na)\}] \neq und$. Let $\$\$pr$ be a set of procedures.

The attribute $(arity\ \$pr)$ specifies the arity of the procedure $\$pr$. A number $\$n$ is an arity in $[\![\$pr, \$s]\!]$ if $[\$s\ .\ \{(arity\ \$pr)\}] = \$n$.

The attribute $(argument\ \$pr\ \$n)$ specifies the $\$n$-th argument of the procedure $\$pr$. A name $\$na$ is an argument in $[\![\$pr, \$n]\!]$ if $[\$s\ .\ \{(argument\ \$pr\ \$n)\}] = \$na$, and $1 \leq \$n \leq [\$s\ .\ \{(arity\ \$pr)\}]$.

The attribute $((argument\ type)\ \$t\ \$pr\ \$n)$ specifies the type of the $\$n$-th argument of the procedure $\$pr$. A type $\$t$ is a type in $[\![\$pr, \$n]\!]$ if $[\$s\ .\ \{((argument\ type)\ \$t\ \$pr\ \$n)\}] \neq und$, and $1 \leq \$n \leq [\$s\ .\ \{(arity\ \$pr)\}]$.

The attribute $(body\ \$pr)$ specifies the body of the procedure $\$pr$. A sequence $\$c^*$ is a body in $[\![\$pr]\!]$ if $[\$s\ .\ \{(body\ \$pr)\}] = (\$c^*)$.

A call level is redefined in MPL4 as a number of embedded function and procedure calls.

7.4.2. Constructs

The procedure declaration is represented by the element $(procedure\ \$pr\ (\$tna1\ ...\ \$tna\$n)\ \$c^*)$. It specifies the declaration of the procedure $\$pr$, the arguments $\$na[\![\$tna1]\!]$, ..., $\$na[\![\$tna\$n]\!]$ of the types $\$t[\![\$tna1]\!]$, ..., $\$t[\![\$tna\$n]\!]$, and the body $\$c^*$.

Axiom: Procedure declarations are elements of the program body.

The exit statement is represented by the element $exit$.

The procedure call is represented by the element $(call\ \$pr\ \$c^*)$. It specifies the call of the procedure $\$pr$ with the arguments $\$c^*$.

7.5. MPL5: pointers

The MPL5 language is an extension of MPL4 that adds the pointers feature: the pointer types, the operations of pointer content access, variable address access and pointer deletion, statements of pointer content assignment and pointer deletion.

7.5.1. Types, values, states

An element $(pointer\ \$t)$ is called a pointer type in $[\![\$t]\!]$. An element $\$e$ is a pointer type if $\$e$ is a pointer type in $[\![\$t]\!]$ for some $\$t$. Let $\$\$pt$ be a set of pointer types.

The absolute type $pointer$ specifies pointers in MPL5. An element $\$e \in \$\$ats$ is a pointer if $\$e$ has the absolute type $pointer$, and the value in $[\![\$e]\!]$ belongs to $\$\$n$. Thus, pointers are represented in MPL5 by natural numbers categorized by the type $pointer$. Let $\$\$po$ be a set of pointers.

For MPL5, $\$\$t[\![MPL5]\!] = \$\$t[\![MPL4]\!] \cup \$\$pt$, $\$\$v[\![\mathrm{MPL5}]\!] = \$\$v[\![\mathrm{MPL4}]\!] \cup \$\$po$, and $\$\$ex[\![MPL5]\!] = \$\$ex[\![MPL4]\!]$.

The attribute $(pointer\ \$po)$ specifies pointers in states. A pointer $\$po$ is a pointer in $[\![\$s]\!]$ if $[\$s\ .\ \{(pointer\ \$po)\}] \neq und$.

The attribute $((content\ type)\ \$po)$ specifies the content type of the pointer $\$po$. A type $\$t$ is a content type in $[\![\$po,\ \$s]\!]$ if $\left[\$s\ .\ \left\{\left((content\ type)\$po\right)\right\}\right] = \$t$. It specifies the type of the content to which the pointer $\$po$ refers.

Axiom: If $[\$s\ .\ \{(pointer\ \$po)\}] \neq und$, then $[\$s\ .\ \{((content\ type)\ \$po)\}] \in \$\$t[\![\$s]\!]$.

The pointer $\$po$ has the type $(pointer\ \$t)$ in $[\![\$s]\!]$ if $\$t$ is a content type in $[\![\$po,\ \$s]\!]$. Thus, the type $(pointer\ \$t)$ specifies pointers with the content type $\$t$.

The attribute $(content\ \$po)$ specifies the content of the pointer $\$po$. A value $\$v$ is a content in $[\![\$po,\ \$s]\!]$ if $[\$s\ .\ \{(content\ \$po)\}] = \$v$.

Axiom: If $[\$s\,.\,\{(content\;\$po)\}] \neq und$, then $[\$s\,.\,\{((content\;type)\;\$po)\}] = \$t$, and $[\$s\,.\,\{(content\;\$po)\}] \in [content\;\$t]$ for some $\$t[\![\$s]\!]$.

The attribute $(pointer\;\$va\;\$sc\;\$cl)$ specifies variables in $[\![\$sc, \$cl]\!]$ by the pointers referring to their values. A name $\$na$ is a variable in $[\![\$p,\;\$s, \$sc, \$cl]\!]$ if $[\$s\,.\,\{(pointer\;\$na\;\$sc\;\$cl)\}] = \$p$. A variable $\$va$ represents $\$p$ in $[\![\$s, \$sc, \$cl]\!]$ if $\$va$ is a variable in $[\![\$p,\;\$s, \$sc, \$cl]\!]$. A name $\$na$ is a variable in $[\![\$s, \$sc, \$cl]\!]$ if $\$na$ is a variable in $[\![\$p,\;\$s, \$sc, \$cl]\!]$ for some $\$p \in \$\$p[\![\$s]\!]$. A pointer $\$p$ is a pointer in $[\![\$va, \$s,\;\$sc, \$cl]\!]$ if $\$va$ is a variable in $[\![\$p,\;\$s, \$sc, \$cl]\!]$.

The content of the pointer $\$p[\![\$va]\!]$ coincides with the value of the variable $\$va$. A type $\$t$ is a type in $[\![\$va, \$s, \$sc, \$cl]\!]$ if $[\$s\,.\,\{((content\;type)\;[\$s\,.\,\{(pointer\;\$na\;\$sc\;\$cl)\}])\}] = \$t$. A value $\$v$ is a value in $[\![\$va, \$s, \$sc, \$cl]\!]$ if $[\$s\,.\,\{(content\;[\$s\,.\,\{(pointer\;\$va\;\$sc\;\$cl)\}])\}] = \$v$.

Axiom: If $[\$s\,.\,\{(pointer\;\$va\;\$sc\;\$cl)\}] \neq und$, then $[\$s\,.\,\{(pointer\;\$va\;\$sc\;\$cl)\}] \in \$\$po[\![\$s]\!]$.

7.5.2. Constructs

Pointers are represented by elements of $\$\$po$.

The pointer content access operation is represented by the element $(*\;\$c)$. If $\$po$ is a value of $\$c$, then it returns the content in $[\![\$po]\!]$.

The variable address access operation is represented by the element $(\&\;va)$. It returns the pointer in $[\![\$s, \$va, [\$s\,.\,\{(current\;scope)\}], [\$s\,.\,\{(current\;call\;level)\}]]\!]$.

The pointer addition operation is represented by the element $(new\;\$pt)$. It adds a new pointer of the type $\$pt$.

The pointer content assignment statement is represented by the element $(*\;\$c1 := \;\$c2)$. If $\$po$ and $\$v$ are the values of $\$c1$ and $\$c2$, and they have the types $(pointer\;\$t)$ and $\$t$ for some $\$t$, then it assigns $\$v$ to the content of $\$po$.

The pointer deletion operation is represented by the element $(delete\;\$c)$. If $\$po$ is a value of $\$c$, then it specifies the deletion of the pointer $\$po$.

7.6. MPL6: jump statements

The MPL6 language is an extension of MPL5 that adds the jump statements feature: break statement, continue statement, goto statement and labelled statement.

7.6.1. Types, values, states

The exception $(break{:}\{type\}){::}\{exc\}$ specifies the execution of the break statement.

The exception $(continue{:}\{type\}){::}\{exc\}$ specifies the execution of the continue statement.

The exception $(goto\colon\{type\},\ \$l\colon\{label\})\colon\colon\{exc\}$ specifies the execution of the goto statement with the label $\$l$.

Let $\$\$ex1$ be a set of such exceptions.

For MPL6, $\$\$t[\![\text{MPL6}]\!] = \$\$t[\![\text{MPL5}]\!]$, $\$\$v[\![\text{MPL6}]\!] = \$\$v[\![\text{MPL5}]\!] \cup \$\$ex[\![\text{MPL6}]\!]$, and $\$\$ex[\![MPL6]\!] = \$\$ex[\![MPL5]\!] \cup \$\$ex1$.

An element $\$l$ is a label if $\$l$ is a name. Let $\$\$l$ be a set of labels.

7.6.2. Constructs

The label statement with the label $\$l$ is represented by the element $(label\ \$l)$. It specifies the program point labelled by the label $\$l$. The labelled statement is represented by the sequence $(label\ \$l)\ \$c$. It specifies that the statement $\$c$ is labelled by the label $\$l$.

The break statement is represented by the element $break$.

The continue statement is represented by the element $continue$.

The goto statement is represented by the element $(goto\ \$l)$.

7.7. MPL7: dynamic arrays

The MPL7 language is an extension of MPL6 that adds the dynamic arrays feature: dynamic array types, the array element access operation and the array element assignment statement.

7.7.1. Types, values, states

An element $(array\ \$t)$ is called a dynamic array type in $[\![\$t]\!]$. An element $\$e$ is a dynamic array type if $\$e$ is a dynamic array type in $[\![\$t]\!]$ for some $\$t$. Let $\$\$dat$ be a set of dynamic array types.

An element $\$e$ is an array type if $\$e$ is a dynamic array type. Let $\$\$at$ be a set of array types.

The absolute type $(dynamic\ array)$ specifies dynamic arrays. An element $\$dar$ is a dynamic array if $\$dar = ((\$e^{*})\colon\{content\},\ \$t\colon\{type\})\colon\colon\{(dynamic\ array)\}$, and $\$e^{*}$ consists of the elements of $[content\ \$t]$. The elements \$se and \$t are called the content and the element type in $[\![\$dar]\!]$. Let $\$\$dar$ be a set of dynamic arrays.

An element $\$e$ is an array if $\$e$ is a dynamic array. Let $\$\$ar$ be a set of arrays.

For MPL7, $\$\$t[\![MPL7]\!] = \$\$t[\![MPL6]\!] \cup \$\$\text{d}at$, $\$\$v[\![\text{MPL7}]\!] = \$\$v[\![\text{MPL6}]\!] \cup \$\$dar$, and $\$\$ex[\![MPL7]\!] = \$\$ex[\![MPL6]\!]$.

The dynamic array $\$dar$ has the type $(array\ \$t)$ if $\$t$ is an element type in $[\![\$dar,\ \$s]\!]$. Thus, the type $(array\ \$t)$ specifies dynamic arrays with the element type $\$t$.

A value $\$v$ is a value in $[\![\$ar,\ \$n]\!]$ if $\big[[\$ar\ .\ \{content\}]..\ \$n\big] = \$v$. The element $\$v$ specifies the value of $\$n$-th element of $\$ar$.

7.7.2. Constructs

The array element access operation is represented by the element ($\$c1$ [$\$c2$]). If $\$ar$ and $\$n$ are the values of $\$c1$ and $\$c2$, then it returns the value in $[\![\$ar,\ \$n,\ \$s]\!]$.

The array element assignment operation is represented by the element ($\$c1\ [\$c2] \coloneqq \$c3$). If $\$dar$, $\$n$ and $\$v$ are the values of $\$c1$, $\$c2$ and $\$c3$, and $1 \leq \$n \leq [len\ [\$dar\ .\ \{content\}]]$, then it replaces the value of the $\$n$-th element of $\$dar$ by $\$v$. If $\$dar$, $\$n$ and $\$v$ are the values of $\$c1$, $\$c2$ and $\$c3$, and $\$n > [len\ [\$dar\ .\ \{content\}]]$, then it replaces the value of the $\$n$-th element of $\$dar$ by $\$v$ and the values of the elements of $\$dar$ from $\left[len\ [\$dar\ .\ \{content\}]\right] + 1$ to $\$n - 1$ by und.

The array element assignment operation is represented by the element ($\$c1\ [\$c2] \coloneqq \$c3$) where the expressions $\$c1$, $\$c2$ and $\$c3$ have the types $(array\ \$t)$, nat and $\$t$ for some $\$t$. It assigns $\$v$ to the $\$n$-th element of $\$ar$ where $\$ar$, $\$n$ and $\$v$ are the values of $\$c1$, $\$c2$ and $\$c3$ in $[\![\$s]\!]$.

7.8. MPL8: static arrays

The MPL7 language is an extension of MPL6 that adds the static arrays feature: static array types, the array element access operation and the array element assignment statement.

7.8.1. Types, values, states

An element $(array\ \$t\ \$n)$ is called a static array type in $[\![\$t]\!]$. An element $\$e$ is a static array type if $\$e$ is a static array type in $[\![\$t]\!]$ for some $\$t$. Let $\$\$sat$ be a set of static array types.

An element $\$e$ is an array type if $\$e$ is a dynamic array type, or $\$e$ is a static array type. Let $\$\$at$ be a set of array types.

The absolute type $(static\ array)$ specifies arrays. An element $\$sar$ is a static array if $\$sar = ((\$e^*){:}\{content\},\ \$t{:}\{type\}){::}\{(static\ array)\}$, and $\$e^*$ consists of the elements of $[content\ \$t]$. The elements $se and $t are called the content and the element type in $[\![\$sar]\!]$. Let $\$\$sar$ be a set of arrays.

An element $\$e$ is an array if $\$e$ is a dynamic array, or $\$e$ is a static array. Let $\$\$ar$ be a set of arrays

For MPL7, $\$\$t[\![MPL7]\!] = \$\$t[\![MPL6]\!] \cup \$\$sat$, $\$\$v[\![\mathrm{MPL7}]\!] = \$\$v[\![\mathrm{MPL6}]\!] \cup \$\$sar$, and $\$\$ex[\![MPL7]\!] = \$\$ex[\![MPL6]\!]$.

The array $\$sar$ has the type $(array\ \$t\ \$n)$ if $\$t$ is an element type in $[\![\$sar,\ \$s]\!]$, and $[len\ [\$sar\ .\ \{content\}]] = \$n$. Thus, the type $(array\ \$t\ \$n)$ specifies static arrays with the element type $\$t$ and the content of the length $\$n$.

7.8.2. Constructs

The array element access operation does not depend on the specific features of dynamic arrays. Therefore it is extended for static arrays by simple array redefinition.

The array element assignment operation is extended for static arrays as follows: if $\$sar$, $\$n$ and $\$v$ are the values of $\$c1$, $\$c2$ and $\$c3$, and $1 \leq \$n \leq [len\ [\$sar\ .\ \{content\}]]$, then $(\$c1\ [\$c2] \coloneqq \$c3)$ replaces the value of the $\$n$-th element of $\$sar$ by $\$v$.

7.9. MPL9: structures

The MPL9 language is an extension of MPL8 that adds the structures feature: the structure types, the structure field access operation, structure declarations, and the structure field assignment statement.

7.9.1. Types, values, states

The attribute $((structure\ type)\ \$na)$ specifies structure types in states. A name $\$na$ is a structure type in $[\![\$s]\!]$ if $[\$s\ .\ \{((structure\ type)\ \$na)\}] \neq und$. Let $\$\$st$ be a set of structure types.

For MPL9, $\$\$t[\![MPL9]\!] = \$\$t[\![MPL8]\!] \cup \$\$st$.

The attribute $(field\ \$na\ \$st)$ specifies the fields of the structure type $\$st$. A name $\$fi$ is a field in $[\![\$st, \$s]\!]$ if $[\$s\ .\ \{(field\ \$fi\ \$st)\}] \neq und$. Let $\$\$fi$ be a set of fields.

The attribute $(type\ \$fi\ \$st)$ specifies the type of the field $\$fi$ of the structure type $\$st$. A type $\$t$ is a type in $[\![\$fi,\ \$st,\ \$s]\!]$ if $[\$s\ .\ \{(type\ \$fi\ \$st)\}] = \$t$.

Axiom: If $\$fi$ is a field in $[\![\$st, \$s]\!]$, then $[\$s\ .\ \{(type\ \$st\ \$fi)\}] \in \$\$t[\![\$s]\!]$.

The absolute type $structure$ specifies structures. An element $\$str$ is a structure in $[\![\$s]\!]$ if $\big((\$v1{:}\{\$fi1\} \ldots\ \$v\$n{:}\{\$fi\$n\}){:}\{content\},\ \$st{:}\{type\}\big){:}{:}\{structure\} = \$str$, $\$n > 0$, the structure type $\$st$ has the fields $\$fi1$, …, $\$fi\$n$ and no other fields in $[\![\$s]\!]$, and the values $\$v1$, …, $\$v\$n$ in $[\![\$s]\!]$ have the types of the fields $\$fi1$, …, $\$fi\$n$ in $[\![\$st, \$s]\!]$. The elements $[\$str\ .\ \{content\}]$ and $\$st$ are called the content and the type in $[\![\$str]\!]$. The elements $\$fi1$, …, $\$fi\$n$ are called the fields in $[\![\$str]\!]$. The elements $\$v1$, …, $\$v\$n$ are called the values of these fields in $[\![\$st]\!]$. Let $\$\$str$ be a set of structures.

For MPL9, $\$\$v[\![MPL9]\!] = \$\$v[\![MPL8]\!] \cup \$\$str$, and $\$\$ex[\![MPL9]\!] = \$\$ex[\![MPL8]\!]$.

7.9.2. Constructs

The structure declaration is represented by the element $(structure\ \$na\ (\$tna1\ \ldots\ \$tna\$n))$. It specifies the declaration of the structure type with the name $\$na$, and the fields $\$na[\![\$tna1]\!]$, ..., $\$na[\![\$tna\$n]\!]$ of the types $\$t[\![\$tna1]\!]$, ..., $\$t[\![\$tna\$n]\!]$.

Axiom: structure declarations are elements of the program body.

The structure field access operation is represented by the element $(\$c \setminus .\ \$fi)$. If $\$str$ is the value of $\$c$, then it returns the value in $[\![\$fi, \$str,\ \$s]\!]$.

The structure field assignment operation is represented by the element $(\$c1 \setminus .\ \$fi \coloneqq \$c2)$. If $\$str$ and $\$v$ are the values of $\$c1$ and $\$c2$, then it assigns $\$v$ to the field $\$fi$ of $\$str$.

8. Conclusion

In the paper the formalism of the conceptual model of a programming language has been proposed. It represents types of the programming language, values (in particular, the values of the types the programming language), exceptions (the special kind of values), states and executable constructs (in particular, the elements of programs in the programming language) of the abstract machine of the language, and the constraints (axioms) for these entities at the conceptual level. The new definition of conceptual transition systems oriented to specification of conceptual models of programming languages has been proposed, the language CTSL for redefined conceptual transition systems has been described, and the technique of the use of CTSL as a domain-specific language for specification of conceptual models of programming languages has been presented. We have conducted the incremental development of the conceptual models for the family of sample programming languages to illustrate this technique.

We plan to use the CTSL language as a domain specific language oriented to the development of the conceptual operational semantics of programming languages defined as the operational semantics of representations of executable constructs of the abstract machines of the programming languages in CTSL.

References

1. Prinz A., Møller-Pedersen B., Fischer J. Object-Oriented Operational Semantics. In: Grabowski J., Herbold S. (eds) System Analysis and Modeling. Technology-Specific Aspects of Models. SAM 2016. Lecture Notes in Computer Science, vol 9959. Springer, Cham. P. 132-147.
2. Wider A. Model transformation languages for domain-specific workbenches // Ph.D. thesis, Humboldt-Universitat zu Berlin. 2015.
3. Felleisen M., Findler R.B., Flatt M. // Semantics Engineering with PLT Redex, 1st edn. The MIT Press, Cambridge. 2009.
4. Kahn G. Natural semantics. In: Brandenburg F.J., Vidal-Naquet G., Wirsing M. (eds.) STACS 1987. LNCS. 1987. Vol. 247, P. 22–39.
5. Mosses P.D. Structural operational semantics modular structural operational semantics // J. Logic Algebr. Program. 2004. Vol. 60. P. 195–228.

6. Plotkin G.D.: A structural approach to operational semantics // Technical report. DAIMI FN-19, AARHUS UNIVERSITY (DK). 1981.
7. OMG Editor. OMG Meta Object Facility (MOF) Core Specification Version 2.4.2. // Technical report, Object Management Group. 2014.
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. Rosu G., Serbanuta T.F. An overview of the K semantic framework // J. Logic Algebr. Program. 2010. 79(6). P. 397–434.
10. Clavel M., Duran F., Eker S., Lincoln P., Marti-Oliet N., Meseguer J., Quesada J.F. Rewriting logic and its applications maude: specification and programming in rewriting logic // Theor. Comput. Sci. 2002. 285(2). P. 187–243.
11. Anureev I.S. Conceptual Transition Systems // System Informatics. 2015. Vol. 5. P. 1–41.
12. Anureev I.S. Kinds and language of conceptual transition systems // System Informatics. 2015. Vol. 5. P. 55–74.