УДК 004.8

Kinds and language of conceptual transition systems*

Anureev I.S. (Institute of Informatics Systems)

The language CTSL of specification of conceptual transition systems which are a formalism for description of dynamic discrete systems on the basis of their conceptual structure is proposed. The basic kinds of conceptual transition systems are considered. The basic predefined elements and operations of the CTSL language are defined.

Keywords: transition systems, conceptual structures, ontologies, ontological elements, conceptual transition systems, conceptuals, CTSL

1. Introduction

Development of formalisms, languages and tools for describing the conceptual structure of various systems is an important problem of the modern knowledge industry. Description of changes of the conceptual structure of the system when it functions is an another important problem.

The formalism of description (specification) of systems – conceptual transition systems (CTSs) – that solves these problems was proposed in [1]. To our knowledge, CTSs are the only formalism which meets the following requirements (as is shown in [1]):

1. It describes the conceptual structure of the specified system.
2. It describes the content of the conceptual structure of the specified system, i. e. it describes the specified system in the context of the conceptual structure.
3. It describes the change of the conceptual structure of the specified system.
4. It describes the change of the content of the conceptual structure of the specified system, i. e. it describes the change of the specified system in the context of the conceptual structure.
5. It is quite universal to specify typical ontological elements (concepts, attributes, concept instances, relations, relation instances, individuals, types, domains, and so on.).
6. It provides a quite complete classification of ontological elements, including the determination of their new kinds and subkinds.
7. It is based on the conception 'state – transition' of the usual transition systems, keeping

*Partially supported by RFBR under grants 15-01-05974 and 15-07-04144.

their simplicity and universality and adding a conceptual 'filling'. This requirement is important since the simplicity of determination of transition systems makes them an universal formalism to describe the behavior of various systems (algorithms, programs, software models, computer systems, and so on.).

8. It supports reflection of any order, i. e. allows to specify: the system (reflection of the order 0), the specification of the system (reflection of the order 1), the specification of the specification of the system (reflection of the order 2) and so on. Specifications of the higher order (with reflection of the higher order) impose restrictions on the specifications of the lower order (with reflection of the lower order).

Comparison of CTSs with the formalisms such that abstract state machines [2, 3], ontological transition systems [5, 6] and domain-specific transition systems [7] which partially meet these requirements was made in [1].

In contrast to abstract state machines [11, 12] and ontological transition systems [6], there is no specification language which describes CTSs. The language CTSL (Conceptual Transition System Language) of specification of CTSs is defined in this paper.

Contrary to state which has the detailed conceptual structure in CTSs, the transition relation in CTSs is quite general. Kinds of CTSs which concretize the transition relation are considered in this paper. They are defined in CTSL that thus describes concretizations of the transition relation which are important in practice.

The paper has the following structure. The preliminary concepts and notation are given in section 2. The main constructs of the CTSL language are described in section 3. The basic kinds of CTSs, predefined elements and operations in CTSL are defined in sections 4, 5 and 6, respectively.

2. Preliminaries

Let *bool* = *{true, false}*; *int*, *nat* and *nat0* denote the sets of integers, natural numbers and natural numbers with zero, respectively; *obj*, *fun*, *set*, *lab*, *arg*, and *val* denote sets of objects, functions, sets, labels, function arguments and function values, respectively.

The names of the variables which take the values from the set with the name aw, where a is a symbol, and w is a word, are denoted by $\dot{a}w$, $\dot{a}w_1$, $\dot{a}w'$ and so forth. For example, $\dot{s}et$, $\dot{s}et_1$, $\dot{s}et'$ are the names of the variables which take the values from the set *set*. Depending on the context, the name of a variable is interpreted as either the variable, or the value of the variable.

Let $sup(\dot{fun})$ and ω denote the support of $\dot{fun}$ and the indeterminate value of $\dot{fun}$, respectively.

Let $\dot{fun}(\dot{arg}_1 \leftarrow \dot{val}_1, \ldots, \dot{arg}_{\dot{nat}} \leftarrow \dot{val}_{\dot{nat}})$ denote the function $\dot{fun}'$ such that $\dot{fun}'(\dot{arg}) = \dot{fun}(\dot{arg})$, if $\dot{arg}$ is distinct from $\dot{arg}_1$, ..., $\dot{arg}_{\dot{nat}}$, and $\dot{fun}'(\dot{arg}_{\dot{nat}'}) = \dot{val}_{\dot{nat}'}$, if $1 \leq \dot{nat}' \leq \dot{nat}$.

Let $\{\dot{arg}_1{:}\dot{val}_1, \ldots, \dot{arg}_{\dot{nat}}{:}\dot{val}_{\dot{nat}}\}$ denote the function $\dot{fun}$ such that $sup(\dot{fun}) = \{\dot{arg}_1, \ldots, \dot{arg}_{\dot{nat}}\}$, and $\dot{fun}(\dot{arg}_1) = \dot{val}_1$, ..., $\dot{fun}(\dot{arg}_{\dot{nat}}) = \dot{val}_{\dot{nat}}$. The arguments $\dot{arg}_1$, ..., $\dot{arg}_{\dot{nat}}$ are pairwise distinct.

The terms used in the paper are context-dependent. Contexts have the form $[\![\dot{obj}_1, \ldots, \dot{obj}_{\dot{nat}}]\!]$, where the embedded contexts $\dot{obj}_1$, ..., $\dot{obj}_{\dot{nat}}$ have the form: $\dot{lab}{:}\dot{obj}$, $\dot{lab}{:}$ or $\dot{obj}$.

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 $\dot{obj}[\![\dot{obj}_1, \ldots, \dot{obj}_{\dot{nat}}]\!]$ denote the object $\dot{obj}$ in the context $[\![\dot{obj}_1, \ldots, \dot{obj}_{\dot{nat}}]\!]$.

Let *cts* denote a set of conceptual transition states [8]. Let *ato*, *ele*, *eleStr*, *ordStr* and *sta* denote sets of atoms, elements, element structures, ordered structures and conceptual states in $[\![\dot{cts}]\!]$.

Let *samp*, *body*, *cond* and *var* be sets of elements called samples, bodies, conditions and variables, respectively. Let *varSet* be a set of unsorted structures. Elements of $\dot{varSet}$ are called variables.

3. The CTSL language

The CTSL language is a language of specification of CTSs. Atoms, elements and the transition relation are key notions of CTSL.

3.1. Atoms and elements

Atoms in CTSL represent atoms of CTSs which are specified by CTSL. An object $\dot{obj}$ is called an atom in CTSL, if

- either $\dot{obj}$ is a sequence of Unicode symbols except for the whitespace symbols and the symbols *″*, *’ }*, *}*, *(*, *)*, *[*, *]*, *:*, *;* and *,*;
- or $\dot{obj}$ has the form *″*$\dot{obj}_1$*″*, where $\dot{obj}_1$ is a sequence of Unicode symbols in which each occurrence of the character *″* is preceded by the symbol *’*.

Elements in CTSL represent elements of CTSs which are specified by CTSL. They are defined as in CTSs [1] except that the whitespace symbols and the semicolon are element delimiters along with comma. For example, $(\dot{e}le_1,\ \dot{e}le_2)$, $(\dot{e}le_1;\ \dot{e}le_2)$ and $(\dot{e}le_1\ \ \dot{e}le_2)$ represent the same element.

The definition of the transition relation in CTSL uses the notion of substitution.

3.2. Substitutions

A function $\dot{f}un \in ele \to ele$ is called a substitution. Let sub be a set of substitutions.

A function $subF \in ele \to ele$ is called a substitution function in $[\![\dot{s}ub]\!]$, if the following properties hold:

- if $\dot{e}le \in sup(\dot{s}ub)$, then $subF[\![\dot{s}ub]\!](\dot{e}le) = \dot{s}ub(\dot{e}le)$;
- if $\dot{e}le' \in sup(\dot{s}ub)$, and $\dot{s}ub(\dot{e}le') = (\dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at_0})$, then
$$subF[\![\dot{s}ub]\!](\dot{e}le\ .::\ \dot{e}le') = (subF[\![\dot{s}ub]\!](\dot{e}le),\ \dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at_0}));$$
- if $\dot{e}le' \in sup(\dot{s}ub)$, and $\dot{s}ub(\dot{e}le') \notin ordStr \cup \{()\}$, then $subF[\![\dot{s}ub]\!](\dot{e}le\ .::\ \dot{e}le') = [\![\omega]\!]$;
- if $\dot{e}le' \in sup(\dot{s}ub)$, and $\dot{s}ub(\dot{e}le') = (\dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at_0})$, then
$$subF[\![\dot{s}ub]\!](\dot{e}le'\ ::.\ \dot{e}le) = (\dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at_0}),\ subF[\![\dot{s}ub]\!](\dot{e}le));$$
- if $\dot{e}le' \in sup(\dot{s}ub)$, and $\dot{s}ub(\dot{e}le') \notin ordStr \cup \{()\}$, then $subF[\![\dot{s}ub]\!](\dot{e}le'\ ::.\ \dot{e}le) = [\![\omega]\!]$;
- if $\dot{e}le' \in sup(\dot{s}ub)$, $\dot{s}ub(\dot{e}le') = \{\dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at_0}\}$, and $\mathrm{subF}[\![\dot{s}ub]\!](\dot{e}le),\ \dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at_0}$ are pairwise distinct, then $subF[\![\dot{s}ub]\!](\dot{e}le\ .::u\ \dot{e}le') = \{subF[\![\dot{s}ub]\!](\dot{e}le),\ \dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at_0}\}$;
- if $\dot{e}le' \in sup(\dot{s}ub)$, and $\dot{s}ub(\dot{e}le') \notin unoStr \cup \{\{\}\}$, then $subF[\![\dot{s}ub]\!](\dot{e}le\ .::u\ \dot{e}le') = [\![\omega]\!]$;
- if $\dot{e}le' \in sup(\dot{s}ub)$, $\dot{s}ub(\dot{e}le') = \{\dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at_0}\}$, and $\mathrm{subF}[\![\dot{s}ub]\!](\dot{e}le),\ \dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at_0}$ are not pairwise distinct, then $subF[\![\dot{s}ub]\!](\dot{e}le\ .::u\ \dot{e}le') = [\![\omega]\!]$;
- if $\dot{a}to \notin sup(\dot{s}ub)$, then $subF[\![\dot{s}ub]\!](\dot{a}to) = \dot{a}to$;
- if $() \notin sup(\dot{s}ub)$, then $subF[\![\dot{s}ub]\!](()) = ()$;
- if $\{\} \notin sup(\dot{s}ub)$, then $subF[\![\dot{s}ub]\!](\{\}) = \{\}$;
- if $\dot{o}rdStr \in sup(\dot{s}ub)$, and $\dot{o}rdStr$ has the form $(\dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at})$, then
$$subF[\![\dot{s}ub]\!](\dot{o}rdStr) = (subF[\![\dot{s}ub]\!](\dot{e}le_1),\ \ldots,\ subF[\![\dot{s}ub]\!](\dot{e}le_{\dot{n}at}));$$
- if $\dot{u}noStr \in sup(\dot{s}ub)$, and $\dot{u}noStr$ has the form $\{\dot{e}le_1,\ \ldots,\ \dot{e}le_{\dot{n}at}\}$, then
$$subF[\![\dot{s}ub]\!](\dot{u}noStr) = \{subF[\![\dot{s}ub]\!](\dot{e}le_1),\ \ldots,\ subF[\![\dot{s}ub]\!](\dot{e}le_{\dot{n}at})\};$$
- if $\dot{l}abStr \in sup(\dot{s}ub)$, and $\dot{l}abStr$ has the form $\{\dot{l}ab_1{:}\dot{e}le_1,\ \ldots,\ \dot{l}ab_{\dot{n}at}{:}\dot{e}le_{\dot{n}at}\}$, then
$$subF[\![\dot{s}ub]\!](\dot{l}abStr) =$$
$$\{subF[\![\dot{s}ub]\!](\dot{l}ab_1){:}subF[\![\dot{s}ub]\!](\dot{e}le_1),\ \ldots,\ subF[\![\dot{s}ub]\!](\dot{l}ab_{\dot{n}at}){:}subF[\![\dot{s}ub]\!](\dot{e}le_{\dot{n}at})\}.$$

An element $\dot{e}le$ is called an instance in $[\![\dot{e}le',\ \dot{s}ub]\!]$, if $subF[\![\dot{s}ub]\!](\dot{e}le') = \dot{e}le$. An element $\dot{e}le'$ is called a sample in $[\![\dot{e}le,\ \dot{s}ub]\!]$, if $subF[\![\dot{s}ub]\!](\dot{e}le') = \dot{e}le$. Let $samp$ be a set of samples.

3.3. The transition relation

The transition relation in CTSL is defined on programs. Programs in CTSL include macros and the executable element. Macros are used to simplify the representation of the executable element. The notion of the executable element is defined in section 4.2.

An element $\dot{e}le$ of the form *(macro* $\dot{v}ar$ $\dot{b}ody$*)* is called a macro. The elements $\dot{v}ar$ and $\dot{b}ody$ are called a variable and a body in $[\![\dot{e}le]\!]$. Let *macro* be a set of macros.

An element $\dot{e}le$ of the form *(prog* $\dot{m}acro_1 \ldots \dot{m}acro_{\dot{n}at0}$ $\dot{e}le'$*)* is called a program in CTSL. The element $\dot{e}le'$ is called an executable element in $[\![\dot{e}le]\!]$. Let *prog* be a set of programs in CTSL.

A function $traRel \in prog \times tra \to bool$ is called the transition relation in *prog:*, if $traRel(\dot{p}rog, \dot{t}ra)$ if and only if either $\dot{p}rog$ has the form *(prog* $\dot{e}le$*)*, and $traRel(\dot{t}ra(1)(\$exeEle \leftarrow \dot{e}le))$, or $\dot{p}rog$ has the form

$$(prog\ (macro\ \dot{v}ar\ \dot{b}ody)\ \dot{m}acro_1 \ldots \dot{m}acro_{\dot{n}at0}\ \dot{e}le),$$

and

$$traRel((prog\ subF[\![\{\dot{v}ar{:}\dot{b}ody\}]\!]\ (\dot{m}acro_1) \ldots subF[\![\{\dot{v}ar{:}\dot{b}ody\}]\!](\dot{m}acro_{\dot{n}at0})$$
$$subF[\![\{\dot{v}ar{:}\dot{b}ody\}]\!](\dot{e}le)),\ \dot{t}ra).$$

The conceptual *$exeEle* is defined in section 4.2

4. Basic kinds of CTSs

Basic kinds of CTSs are defined in this section.

4.1. CTSs with transition values

A CTS with transition values is characterized by the fact that its transitions can return values.

The conceptual *(0:val)* is called a transition value specifier. Let *$val* denote *(0:val)*. An element $\dot{e}le$ is called a value in $[\![\dot{t}ra]\!]$, if $traRel(\dot{t}ra)$, and $\dot{e}le = \dot{t}ra(2)(\$val)$. Thus, the individual *val* specifies a transition value. A transition $\dot{t}ra$ returns a value $\dot{e}le$, if $\dot{e}le$ is a value in $[\![\dot{t}ra]\!]$.

A conceptual $\dot{c}on$ is called an exception, if $\dot{c}on(1) = exception$. Thus, the concept *exception* specifies exceptions. Let *exc* be a set of exceptions. A transition $\dot{t}ra$ returns (or generates) an exception $\dot{e}xc$, if $\dot{e}xc$ is a value in $[\![\dot{t}ra]\!]$. The element $\dot{e}xc(0)$ specifies usually information about the generated exception, and the elements $\dot{e}xc(\dot{i}nt)$, where $\dot{i}nt < 0$, concretizes usually a kind of this information. A transition $\dot{t}ra$ is normally executed, if $\dot{t}ra$ returns no exception.

A system $\dot{cts}$ is called a CTS with transition values, if $\dot{sta}(\$val) \neq \omega$ for all *sta* such that *sta* is admissible in $[\![\dot{cts}]\!]$.

4.2. CTSs with executable elements

A CTS with executable elements is a CTS with transition values which is characterized by the fact that its transitions are associated with executable elements, and executable elements can return values.

A function $traRel \in ele \times tra \to bool$ is called the transition relation in $[\![ele:]\!]$. It specifies transitions initiated by executable elements. An element $\dot{ele}$ is executed in $[\![\dot{tra}, traRel[\![ele:]\!]]\!]$, if $traRel[\![ele:]\!](\dot{ele}, \dot{tra})$. An element $\dot{ele}$ is executed in $[\![\dot{sta}, traRel[\![ele:]\!]]\!]$, if there exists $\dot{sta}'$ such that $\dot{ele}$ is executed in $[\![(\dot{sta}, \dot{sta}'), traRel[\![ele:]\!]]\!]$. An element $\dot{ele}$ executes (initiates) a transition $\dot{tra}$, if $\dot{ele}$ is executed in $[\![\dot{tra}, traRel[\![ele:]\!]]\!]$.

The conceptual *(0:exeEle)* is called an executable element specifier. Let *$exeEle* denote *(0:exeEle)*. Thus, the individual *exeEle* specifies an executable element.

A CTS with transition values $\dot{cts}$ is called a CTS in $[\![traRel[\![ele:]\!]]\!]$, if $traRel(\dot{tra})$ if and only if $\dot{tra}(1)(\$exeEle) \neq \omega$, and

- either $\dot{tra}(1)(\$exeEle)$ is executed in $[\![\dot{tra}, traRel[\![ele:]\!]]\!]$,
- or $\dot{tra}(1)(\$exeEle)$ is not executed in $[\![\dot{tra}, traRel[\![ele:]\!]]\!]$, and
  $\dot{tra}(2) = \dot{tra}(1)(\$val \leftarrow (-1{:}unknownElement, 0{:}\dot{ele}, 1{:}exception))$.

An element $\dot{val}$ is called a value in $[\![\dot{ele}, \dot{tra}]\!]$, if $\dot{ele}$ is executed in $[\![\dot{tra}]\!]$, and $\dot{val}$ is a value in $[\![\dot{tra}]\!]$.

Executable elements in such CTSs can be partitioned into defined and predefined ones. In this case, the transition relation $traRel[\![exe:]\!]$ is defined as the union of the transition relations $traRel[\![predef:]\!]$ and $traRel[\![def:]\!]$ such that $traRel[\![predef:]\!], traRel[\![def:]\!] \in ele \times tra \to bool$. An element $\dot{ele}$ is called predefined in $[\![traRel]\!]$, if there exists $\dot{tra}$ such that $traRel[\![predef:]\!](\dot{ele}, \dot{tra})$. An element $\dot{ele}$ is called defined in $[\![traRel]\!]$, if there exists $\dot{tra}$ such that $traRel[\![def:]\!](\dot{ele}, \dot{tra})$.

In the case of partitioning executable elements into predefined and defined ones, a CTS in $[\![traRel[\![ele:]\!]]\!]$ is redefined as follows: $traRel(\dot{tra})$ if and only if $\dot{tra}(1)(\$exeEle) \neq \omega$, and

- $\dot{tra}(1)(\$exeEle)$ is executed in $[\![\dot{tra}, traRel[\![predef:]\!]]\!]$, or
- $\dot{tra}(1)(\$exeEle)$ is not executed in $[\![\dot{tra}, traRel[\![predef:]\!]]\!]$, and $\dot{tra}(1)(\$exeEle)$ is executed in $[\![\dot{tra}, traRel[\![predef:]\!]]\!]$, or
- $\dot{tra}(1)(\$exeEle)$ is not executed in $[\![\dot{tra}, traRel[\![predef:]\!]]\!]$ and $[\![\dot{tra}, traRel[\![def:]\!]]\!]$, and $\dot{tra}(2) = \dot{tra}(1)(\$val \leftarrow (-1{:}unknownElement, 0{:}\dot{ele}, 1{:}exception))$.

CTSs with executable elements can be used, for example, to specify abstract machines of programming languages. In this case, executable elements are executable constructs of programming languages.

4.3. CTSs with execution contexts

CTSs with execution contexts generalize CTSs with executable elements and transition values. They are characterized by the fact that elements are executed in execution contexts, and their values are stored in these contexts.

An element *ėle* is called an execution context in $[\![\dot{sta}]\!]$, if

$$\dot{sta}((0{:}\dot{e}le,\ 1{:}executionContext)) \neq \omega.$$

Thus, the concept *executionContext* describes execution contexts. Let *exeCont* be a set of execution contexts in $[\![\dot{sta}]\!]$

Specifiers of transition values and executable elements are redefined in CTSs with execution contexts.

A conceptual *(-1:val, 0:ėxeCont, 1:executionContext)* is called a transition value specifier in $[\![\dot{e}xeCont]\!]$. Let *$val* denote *(-1:val, 0:ėxeCont, 1:executionContext)*. An element *ėle* is called a value in $[\![\dot{tra},\ \dot{e}xeCont]\!]$, if *traRel(ṫra)*, and $\dot{e}le = \dot{tra}(2)(\$val)$. Thus, the attribute *val* specifies a transition value in execution contexts. A transition *ṫra* returns a value *ėle* in $[\![\dot{e}xeCont]\!]$, if *ėle* is a value in $[\![\dot{tra},\ \dot{e}xeCont]\!]$. An element *v̇al* is called a value in $[\![\dot{e}le,\ \dot{tra},\ \dot{e}xeCont]\!]$, if *ėle* is executed in $[\![\dot{tra},\ \dot{e}xeCont]\!]$, and $\dot{v}al = \dot{tra}(2)(\$val)$.

A transition *ṫra* returns (or generates) an exception *ėxc* in $[\![exeCont]\!]$, if *ėxc* is a value in $[\![\dot{tra},\ \dot{e}xeCont]\!]$. A transition *ṫra* is normally executed in $[\![exeCont]\!]$, if *ṫra* returns no exception in $[\![exeCont]\!]$.

A conceptual *(-1:exeEle, 0:ėxeCont, 1:executionContext)* is called an executable element specifier in $[\![exeCont]\!]$. Let *$exeEle* denote *(-1:exeEle, 0:ėxeCont, 1:executionContext)*. Thus, the attribute *exeEle* specifies an executable element in execution contexts. The same element can be executed in different execution contexts.

A function $traRel \in ele \times ele \times tra \to bool$ is called the transition relation in $[\![exeCont{:}]\!]$. It specifies transitions initiated by elements which are executed in execution contexts. An element *ėle* is executed in $[\![\dot{tra},\ \dot{e}xeCont]\!]$, if $traRel[\![exeCont{:}]\!](\dot{e}le,\ \dot{e}xeCont,\ \dot{tra})$. An element *ėle* is executed in $[\![\dot{sta},\ \dot{e}xeCont]\!]$, if there exists *ṡta* such that *ėle* is executed in $[\![(\dot{sta},\ \dot{sta}'),\ \dot{e}xeCont]\!]$. An element *ėle* executes (initiates) a transition *ṫra* in $[\![\dot{e}xeCont]\!]$, if *ėle* is executed in $[\![\dot{tra},\ \dot{e}xeCont]\!]$.

A CTS $\dot{cts}$ is called a CTS in $[\![traRel[\![exeCont:]\!]]\!]$, if $\dot{sta}(\$val) \neq \omega$ for all $\dot{sta}$ and $\dot{exeCont}[\![\dot{sta}]\!]$ such that $\dot{sta}$ is admissible in $[\![\dot{cts}]\!]$, and $traRel(\dot{tra})$ if and only if there exists $\dot{exeCont}[\![\dot{sta}]\!]$ such that $\dot{tra}(1)(\$exeEle) \neq \omega$, and either $\dot{tra}(1)(\$exeEle)$ is executed in $[\![\dot{tra}, \dot{exeCont}]\!]$, or $\dot{tra}(1)(\$exeEle)$ is not executed in $[\![\dot{tra}, \dot{exeCont}]\!]$, and

$$\dot{tra}(2) = \dot{tra}(1)(\$val \leftarrow (-1{:}unknownElement, 0{:}\dot{ele}, 1{:}exception)).$$

4.4. CTSs with counters

CTSs with counters are CTS with executable elements and transition values which are characterized by the fact that they can define named counters and generate new elements based on them.

A conceptual $(0{:}\dot{ele}, 1{:}counter)$ is called a counter specifier with name $\dot{ele}$. Let cou and nam be sets of counter specifiers and their names, respectively. An element $\dot{cou}$ is called a counter in $\dot{sta}$, if $\dot{sta}(\dot{cou}) \neq \omega$. Thus, the concept $counter$ defines counters. The element $\dot{sta}(\dot{cou})$ is called a value in $[\![cou{:}\dot{cou}, \dot{sta}]\!]$.

An element $\dot{con}$ is called generated in $[\![\dot{cou}]\!]$, if $\dot{con}$ has the form $(0{:}\dot{nat}, 1{:}\dot{nam})$, and $\dot{nam}$ is a name of $\dot{cou}$. Thus, the name of a counter is a concept for elements generated by this counter.

An element $\dot{ele}$ of the form $(newCount\ \dot{nam})$ is called an element generator in $[\![conc{:}\dot{nam}]\!]$ and defined as follows: $traRel(\dot{ele}, \dot{tra})$ if and only if either $\dot{tra}(1)(\dot{cou}) \neq \omega$, and $\dot{tra}(2) = \dot{tra}(1)(\dot{cou} \leftarrow \dot{tra}(1)(\dot{cou}) + 1, \$val \leftarrow (0{:}\dot{tra}(1)(\dot{cou}) + 1, 1{:}\dot{nam}))$, or $\dot{tra}(1)(\dot{cou}) = \omega$, and $\dot{tra}(2) = \dot{tra}(1)(\dot{cou} \leftarrow 1, \$val \leftarrow (0{:}1, 1{:}\dot{nam}))$.

An element $\dot{nam}$ is called a name in $[\![\dot{ele}]\!]$. An element generator generates a new element by the counter with the name which coincides with the name of the generator, and adds this counter, if it was not.

A CTS with executable elements and transition values $\dot{cts}$ is called a CTS with counters, if the element $(newCount\ \dot{nam})$ is predefined in $[\![\dot{cts}]\!]$.

4.5. CTSs with history varibles

CTSs with history varibles are CTSs with counters which are characterized by the fact that they can define variables storing the history of values of $\$val$.

A conceptual $(0{:}\dot{nat}, 1{:}hvar)$ is called a history variable specifier. Let $hvar$ be a set of history variable specifiers. An element $\dot{hvar}$ is called a history variable in $\dot{sta}$, if $\dot{sta}(\dot{hvar}) \neq \omega$. Thus, the concept $hvar$ defines history variables. An element $\dot{sta}(\dot{hvar})$ is called a value in $[\![hvar{:}\dot{hvar}, \dot{sta}]\!]$.

An element $\dot{e}le$ of the form *(hvar* ($\dot{e}le_1$, ..., $\dot{e}le_{\dot{n}at}$) $\dot{e}le'$*)* is called a history variable generator in $[\![$*(*$\dot{e}le_1$, ..., $\dot{e}le_{\dot{n}at}$*),* $\dot{e}le'$$]\!]$ and defined as follows: *traRel(*$\dot{e}le$, $\dot{t}ra$*)* if and only if there exist $\dot{s}ta_0$, $\dot{s}ta_1$, ..., $\dot{s}ta_{\dot{n}at}$ such that $\dot{s}ta_0 = \dot{t}ra(1)$, *traRel((newCount hvar),* ($\dot{s}ta_{\dot{n}at'-1}$, $\dot{s}ta_{\dot{n}at'}$*))* for all $1 \le \dot{n}at' \le \dot{n}at$, and *traRel(*$\dot{s}ta_{\dot{n}at}$*(*$exeEle \leftarrow \dot{e}le''$, $val \leftarrow \dot{t}ra(1)(val)$, $\dot{t}ra(2)$*)*, where $\dot{e}le''$ is the result of replacement $\dot{e}le_1$, ..., $\dot{e}le_{\dot{n}at}$ in $\dot{e}le'$ by $\dot{s}ta_1$*(*$val*)*, ..., $\dot{s}ta_{\dot{n}at}$*(*$val*)*, respectively.

The elements $\dot{e}le_1$, ..., $\dot{e}le_{\dot{n}at}$ are called variables in $[\![\dot{e}le]\!]$, and $\dot{e}le'$ are called a body in $[\![\dot{e}le]\!]$. A history variable generator generates new history variables corresponding to the variables of the generator and replace all occurences of the generator variables in its body by these history variables.

A CTS with counters $\dot{c}ts$ is called a CTS with history variables, if the element *(hvar* ($\dot{e}le_1$, ..., $\dot{e}le_{\dot{n}at}$*)* $\dot{e}le'$*)* is predefined in $[\![\dot{c}ts]\!]$.

4.6. CTSs with defined conceptuals

CTSs with defined conceptuals specify definitions of conceptuals.

A conceptual $\dot{c}on'$ is called a definition in $[\![\dot{c}on,\ \dot{s}ta]\!]$, if $\dot{n}at$ is an order in $[\![\dot{c}on]\!]$, $\dot{c}on' = \dot{c}on \cup \{(\dot{n}at + 1)$*:conDef*$\}$, and $\dot{s}ta(\dot{c}on') \ne \omega$. An element $\dot{s}ta(\dot{c}on')$ is called a body in $[\![$*def:*$\dot{c}on'$*,* $\dot{s}ta]\!]$. Thus, the definition of a conceptual of the order $\dot{n}at$ is characterized by the attribute *conDef* of the order $\dot{n}at + 1$.

A CTS $\dot{c}ts$ is called a CTS with defined conceptuals, if semantics of conceptuals *sem* is redefined by the following way:

- if $\dot{s}ta(\dot{c}on) \ne \omega$, then *sem(*$\dot{c}on$, $\dot{s}ta$*)* = $\dot{s}ta(\dot{c}on)$;
- if $\dot{s}ta(\dot{c}on) = \omega$, $\dot{c}on'$ is a definition in $[\![\dot{c}on,\ \dot{s}ta]\!]$, and $\dot{e}le$ is a body in $[\![$*def:*$\dot{c}on'$*,* $\dot{s}ta]\!]$, then *sem(*$\dot{c}on$, $\dot{s}ta$*)* = *sem(*$\dot{e}le$, $\dot{s}ta$*)*;
- otherwise, *sem(*$\dot{c}on$, $\dot{s}ta$*)* = ω.

4.7. CTS with transition rules

An element $\dot{e}le$ of the form *(rule* $\dot{e}le_1$ *var* $\dot{v}arSet$ *then* $\dot{e}le_2$*)* is called a transition rule, if the elements of $\dot{v}arSet$ are pairwise distinct. The elements $\dot{e}le_1$, $\dot{v}arSet$ and $\dot{e}le_2$ are called a sample, variable specifier and body in $[\![\dot{e}le]\!]$, respectively. The elements of $\dot{v}arSet$ are called variables in $[\![\dot{e}le]\!]$. Let *rul*, *samp*, and *body* be sets of transition rules and their samples and bodies, respectively.

A function $traRel \in ele \times rul \times tra \to bool$ is called the transition relation in $[\![$*rul:,* $\dot{c}ts]\!]$,

if *traRel⟦rule:⟧(ėle, ṙul, ṫra)* if and only if there exists *ṡub* such that *sup(ṡub) = v̇arSet*, *ėle* is an instance in *⟦ṡamp, ṡub⟧*, *traRel⟦ċts⟧(ṫra(1)($exeEle ← subF⟦ṡub⟧(ḃody)), ṫra(2))*, and *ṫra(2)($val) ≠ (-1:ruleNotExecutable, 0:ṙul, 1:exception)*.

A transition rule *ṙul* is a transition rule in *⟦ṡta⟧*, if *ṡta(0:ṙul, 1:rule) ≠ ω*. Thus, the concept *rule* defines a set of transition rules in *⟦ṡta⟧*.

A CTS with executable elements *ċts* is a CTS in *⟦traRel⟦rul:, ċts⟧⟧*, if *traRel⟦def:⟧(ėle, ṫra)* if and only if there exists *ṙul* such that *ṙul* is a transition rule in *ṫra(1)*, and *traRel⟦rule:, ċts⟧(ėle, ṙul, ṫra)*.

There are shortcuts for the frequently used kinds of transition rules.

An element *ėle* of the form *(rule ṡamp var v̇arSet where ėle′ then ḃody)* is called a conditional transition rule, and *ėle′* is called a condition in *⟦ėle⟧*. It is a shortcut for the rule *(rule ṡamp var v̇arSet then (if ėle′ then ḃody else ($val ::= (-1:ruleNotExecutable, 0:ėle, 1:exception))))*. The elements *(... ::= ...)* and *(if ... then ... else ...)* are defined in sections 5.3 and 5.5, respectively.

An element *ėle* of the form *(rule ṡamp var v̇arSet hvar ȯrdStr then ḃody)* is called a transition rule with history variables, the element *ȯrdStr* is called a history variable specifier in *⟦ėle⟧*, and the elements of *ȯrdStr* are called history variables in *⟦ėle⟧*. It is a shortcut for the rule *(rule ṡamp var v̇arSet then (hvar ȯrdStr ḃody))*. The element *(hvar ...)* is defined in section 4.5.

An element *ėle* of the form *(rule ṡamp var v̇arSet catch ėle′ then ḃody)* is called a transition rule with the return handler, and the element *ėle′* is called a return variable specifier. It is a shortcut for the rule *(rule ṡamp var v̇arSet then (seq (catch ėle′) ḃody))*. The elements *(seq ...)* and *(catch ...)* are defined in sections 5.6 and 5.9, respectively.

Combinations of these kinds of rules can include not more than one occurence of the part *catch ėle* defined last, and any number of the parts *where ėle* and *hvar ȯrdStr* defined from right to left. For example, the rule *(rule ṡamp var v̇arSet hvar ȯrdStr′ where ėle hvar ȯrdStr″ catch ėle′ then ḃody)* is defined by the following sequence of transformations:

1. *(rule ṡamp var v̇arSet hvar ȯrdStr′ where ėle hvar ȯrdStr″ catch ėle′ then ḃody)* →
2. *ėle″ ≡ (rule ṡamp var v̇arSet hvar ȯrdStr′ where ėle catch ėle′ then (hvar ȯrdStr″ ḃody))* →
3. *(rule ṡamp var v̇arSet hvar ȯrdStr′ catch ėle′ then (if ėle then (hvar ȯrdStr″ ḃody)) else ($val ::= (-1:ruleNotExecutable, 0:ėle″, 1:exception)))* →
4. *(rule ṡamp var v̇arSet catch ėle′ then (hvar ȯrdStr′ (if ėle then (hvar ȯrdStr″ ḃody)) else*

($val ::= (-1:ruleNotExecutable, 0:ėle″, 1:exception)))) →

5. *(rule ṡamp var v̇arSet then (seq (catch ėle′) (hvar ȯrdStr′ (if ėle then (hvar ȯrdStr″ ḃody)) else ($val ::= (-1:ruleNotExecutable, 0:ėle″, 1:exception))))).*

The element *ṙul* adds the rule *ṙul* to *ṡta*: *traRel(ėle, ṫra)* if and only if *ṫra(2) = ṫra(1)((0:ṙul, 1:rule) ← true)*.

The element *ėle* of the form *(delete ṙul)* removes the rule *ṡta* from *ċts*: *traRel(ėle, ṫra)* if and only if *ṫra(2) = ṫra(1)((0:ṙul, 1:rule) ← ω)*.

The element *ėle* of the form *(deleteRules)* removes all rules from *ṡta*: *traRel(ėle, ṫra)* if and only if *ṫra(2) = ṫra(1)((0:ṙul$_1$, 1:rule) ← ω, ..., (0:ṙul$_{\dot{n}at}$, 1:rule) ← ω)*, where *{ṙul$_1$, ..., ṙul$_{\dot{n}at}$}* is a set of all transition rules in *ṫra(1)*.

4.8. CTSs with types

CTSs with types specify types of elements and literals of these types.

Let *type ⊂ ele*. An element *ṫype* is called a type. A function *lit ∈ type → 2^{ele}* is called a literal function in ⟦*type*⟧. An element *ėle* is called a literal in ⟦*ṫype*⟧, if *ėle ∈ lit(ṫype)*.

An element *(ėle is ṫype)* is called a characteristic element in ⟦*ṫype*⟧ and defined as follows: *traRel(ėle, ṫra)* if and only if either *ėle ∈ lit(ṫype)*, and *ṫra(2)($val) = true*, or *ėle ∉ lit(ṫype)*, and *ṫra(2)($val) = false*.

A CTS *ċts* is called a CTS with types in ⟦*type, lit*⟧, if *type* is a set of types, *lit* is a literal function in ⟦*type*⟧, and *(ėle is ṫype)* is a predefined element in *ċts* for each *ṫype*.

The set *type* of the CTSL language includes the following basic types:

- *element* such that *lit(element) = ele*;
- *atom* such that *lit(atom) = ato*;
- *emptyStr* such that *lit(emptyElement) = {(), {}}*;
- *emptyOrdStr* such that *lit(emptyOrdStr) = {()}*;
- *emptyUnoStr* such that *lit(emptyUnoStr) = {{}}*;
- *eleStr* such that *lit(eleStr) = eleStr*;
- *ordStr* such that *lit(ordStr) = ordStr*;
- *unoStr* such that *lit(unoStr) = unoStr*;
- *labStr* such that *lit(labStr) = labStr*;
- *int* such that *lit(int) = int*;
- *nat* such that *lit(nat) = nat*;

- *nat0* such that *lit(nat0) = nat0*;
- *bool* such that *lit(bool) = bool*;
- *rule* such that *lit(rule) = rule*;
- *macro* such that *lit(macro) = macro*;
- *program* such that *lit(program) = prog.*

5. Basic predefined elements in CTSL

Basic predefined elements in CTSL are defined in this section.

5.1. The element *omega*

The element *(omega)* is called an indeterminate return and defined as follows: $traRel(\dot{e}le, \dot{t}ra)$ if and only if $\dot{t}ra(2) = \dot{t}ra(1)(\$val \leftarrow \omega)$. It returns an indeterminate value.

5.2. The elements *ordStrToUnoStr* and *unoStrToOrdStr*

The element *ele* of the form *(ordStrToUnoStr ȯrdStr)* is called a converter in $[\![ordStr:, unoStr:]\!]$ and defined as follows: $traRel(\dot{e}le, \dot{t}ra)$ if and only if either $\dot{e}le = (\dot{e}le_1, \ldots, \dot{e}le_{\dot{n}at})$, and $\dot{t}ra(2) = \dot{t}ra(1)(\$val \leftarrow \{\dot{e}le_1, \ldots, \dot{e}le_{\dot{n}at}\})$, or $\dot{e}le \notin ordStr$, and $\dot{t}ra(2) = \dot{t}ra(1)(\$val \leftarrow (0{:}\dot{e}le, 1{:}exception))$. It converts elements of *ordStr* into elements of *unoStr*.

The element *ele* of the form *(unoStrToOrdStr u̇noStr)* is called a converter in $[\![unoStr:, ordStr:]\!]$ and defined as follows: $traRel(\dot{e}le, \dot{t}ra)$ if and only if either $\dot{e}le = \{\dot{e}le_1, \ldots, \dot{e}le_{\dot{n}at}\}$, and $\dot{t}ra(2) = \dot{t}ra(1)(\$val \leftarrow (\dot{e}le'_1, \ldots, \dot{e}le'_{\dot{n}at}))$, where $(\dot{e}le'_1, \ldots, \dot{e}le'_{\dot{n}at})$ is a permutation of $(\dot{e}le_1, \ldots, \dot{e}le_{\dot{n}at})$, or $\dot{e}le \notin unoStr$, and $\dot{t}ra(2) = \dot{t}ra(1)(\$val \leftarrow (0{:}\dot{e}le, 1{:}exception))$. It converts elements of *unoStr* into elements of *ordStr*.

5.3. The assignment

The element $\dot{e}le$ of the form $(\dot{c}on ::= \dot{e}le')$ is called an assignment and defined as follows: $traRel(\dot{e}le, \dot{t}ra)$ if and only if there exists $\dot{s}ta$ such that $traRel(\dot{t}ra(1)(\$exeEle \leftarrow \dot{e}le'), \dot{s}ta)$, and $\dot{t}ra(2) = \dot{s}ta(\dot{c}on \leftarrow \dot{s}ta(\$val))$. It assign the value of $\dot{e}le'$ to $\dot{c}on$. The elements $\dot{c}on$ and $\dot{e}le'$ are called the left-hand and right-hand sides in $[\![\dot{e}le]\!]$, respectively.

5.4. The element *skip*

The element $\dot{ele}$ of the form *(skip)* is defined as follows: *traRel($\dot{ele}$, $\dot{tra}$)* if and only if $\dot{tra}(1) = \dot{tra}(2)$. It executes an action which does not change a state. In particular, it does not change the value of *$val*.

5.5. The conditional element

The element $\dot{ele}$ of the form *(if $\dot{cond}$ then $\dot{ele}_1$ else $\dot{ele}_2$)* is called a conditional element and defined as follows: *traRel($\dot{ele}$, $\dot{tra}$)* if and only if there exists $\dot{sta}$ such that *traRel($\dot{tra}$(1)(\$exeEle ← $\dot{cond}$), $\dot{sta}$)*, and

- $\dot{sta}$(\$val) = *true*, and *traRel($\dot{sta}$(\$exeEle ← $\dot{ele}_1$), $\dot{tra}$(2))*, or
- $\dot{sta}$(\$val) = *false*, and *traRel($\dot{sta}$(\$exeEle ← $\dot{ele}_2$), $\dot{tra}$(2))*, or
- $\dot{sta}$(\$val) $\notin$ {*true, false*}, and $\dot{tra}(2) = \dot{sta}$(\$val ← (0:$\dot{ele}$,1:exception)).

The elements $\dot{cond}$, $\dot{ele}_1$ and $\dot{ele}_2$ are called a condition, *then*-branch and *else*-branch in $[\![\dot{ele}]\!]$. The element $\dot{ele}$ executes *then*-branch or *else*-branch depending on the value of the condition.

The element *(if $\dot{cond}$ then $\dot{ele}$)* is a shortcut for the element *(if $\dot{cond}$ then $\dot{ele}$ else (skip))*.

5.6. The sequential composition

The element $\dot{ele}$ of the form *(seq $\dot{ele}_1$ ... $\dot{ele}_{\dot{nat}0}$)* is called a sequential composition. The elements $\dot{ele}_1$, ..., $\dot{ele}_{\dot{nat}}$ are called elements in $[\![\dot{ele}]\!]$, and their sequence is called a body in $[\![\dot{ele}]\!]$. The element $\dot{ele}$ executes its elements sequentially from left to right.

Semantics of the element *(seq)* coincides with semantics of the element *(skip)*.

The element $\dot{ele}$ of the form *(seq $\dot{ele}_1$, ..., $\dot{ele}_{\dot{nat}}$)* is defined as follows: *traRel($\dot{ele}$, $\dot{tra}$)* if and only if there exists $\dot{sta}$ such that *traRel($\dot{tra}$(1)(\$exeEle ← $\dot{ele}_1$), $\dot{sta}$)*, and *traRel($\dot{sta}$(\$exeEle ← (seq $\dot{ele}_2$, ..., $\dot{ele}_{\dot{nat}}$)), $\dot{tra}$(2))*.

5.7. Evaluators

The element *ele* of the form *(* $\dot{body}$ *)* is called an evaluator and defined as follows: *traRel($\dot{ele}$, $\dot{tra}$)* if and only if there exists $\dot{sta}$ such that *traRel($\dot{tra}$(1)(\$exeEle ← $\dot{body}$), $\dot{sta}$)*, and either $\dot{sta}$(\$val) $\notin$ *exc*, and *traRel($\dot{sta}$(\$exeEle ← $\dot{sta}$(\$val)), $\dot{tra}$(2))*, or $\dot{sta}$(\$val) $\in$ *exc*, and $\dot{tra}(2) = \dot{sta}$.

The element $\dot{body}$ is called a body in $[\![\dot{ele}]\!]$. The element $\dot{ele}$ first executes $\dot{body}$, and then executes the value of $\dot{body}$.

5.8. Quoters

The element $\dot{e}le$ of the form *(quote* $\dot{b}ody$*)* is called a quoter and defined as follows: *traRel(*$\dot{e}le$*,* $\dot{t}ra$*)* if and only if $\dot{t}ra(2) = \dot{t}ra(1)(\$val \leftarrow \dot{b}ody)$. The element $\dot{b}ody$ is called a body in $[\![\dot{e}le]\!]$. The element $\dot{e}le$ changes a state only in *$val*, and assign $\dot{b}ody$ to *$val*.

The object $'\dot{b}ody$ is a shortcut for the element *(quote* $\dot{b}ody$*)*. For example, *'true* is a shortcut for *(quote true)*.

5.9. Return handlers

The element $\dot{e}le$ of the form *(catch* $\dot{v}ar$ $\dot{b}ody$*)* is called a return handler and defined as follows: *traRel(*$\dot{e}le$*,* $\dot{t}ra$*)* if and only if there exists $\dot{s}ta$ such that $traRel[\![ele:]\!]((newCount\ retVar), (\dot{t}ra(1)(\$val \leftarrow true), \dot{s}ta))$, and

$$traRel(\dot{s}ta(exeEle \leftarrow subF[\![\{\dot{v}ar \leftarrow \dot{s}ta(\$val)\}]\!](\dot{b}ody)), \dot{t}ra(2)).$$

The elements $\dot{v}ar$ and $\dot{b}ody$ are called a return variable and body in $[\![\dot{e}le]\!]$, respectively. An element of the form *(*$\dot{n}at$*, retVar)* is called a return variable. The element $\dot{e}le$ stores the current value of *$val* into a new return variable $\dot{v}ar'$, resets the value of *$val* to *true* and executes the body of $\dot{e}le$ in which all occurences of the return variable in $[\![\dot{e}le]\!]$ are replaced by $\dot{v}ar'$. It models exception handling in CTSL.

5.10. Selectors

The element $\dot{e}le$ of the form *(select* $\dot{e}le'$ *from* $\dot{s}amp$ *var* $\dot{v}arSet$ *for* $\dot{c}ond$*)* is called a selector and defined as follows: *traRel(*$\dot{e}le$*,* $\dot{t}ra$*)* if and only if $\dot{t}ra(2) = \dot{t}ra(1)(\$val \leftarrow \dot{s}et)$, where $\dot{s}et \in unoStr$ is a set of $subF[\![\dot{s}ub]\!](\dot{e}le')$ such that $subF[\![\dot{s}ub]\!](\dot{s}amp)$ is a conceptual in $[\![\dot{t}ra(1)]\!]$, $sup(\dot{s}ub) = \dot{v}arSet$, and there exists $\dot{s}ta$ such that $traRel(\dot{t}ra(1)(\$exeEle \leftarrow subF[\![\dot{s}ub]\!](\dot{c}ond)), \dot{s}ta)$, and $\dot{s}ta(\$val) \neq (-1{:}notSelected, 1{:}exception)$.

The elements $\dot{e}le'$, $\dot{s}amp$, $\dot{v}arSet$ and $\dot{c}ond$ are called a selection specifier, sample, set of variables and condition in $[\![\dot{e}le]\!]$.

Thus, the element $\dot{e}le$ returns the set of instances of the selection specifier $\dot{e}le'$ for all substitutions $\dot{s}ub$ defined on variables from $\dot{v}arSet$ such that there exists a conceptual in $\dot{s}ta$ which is an instance of the sample $\dot{s}amp$ in $[\![\dot{s}ub]\!]$, and the instance of the condition $\dot{c}ond$ in $[\![\dot{s}ub]\!]$ does not return the exception *(-1:notSelected, 1:exception)*.

The element $\dot{e}le$ of the form *(select* $\dot{e}le'$ *from* $\dot{s}amp$ *var* $\dot{v}arSet$ *where* $\dot{c}ond$*)* is a shortcut for the element *(select* $\dot{e}le'$ *from* $\dot{s}amp$ *var* $\dot{v}arSet$ *for (if* $\dot{c}ond$ *then (skip) else ($val ::= (-1{:}notSelected, 1{:}exception))))*. The element $\dot{c}ond$ is called a condition in $[\![\dot{e}le]\!]$.

5.11. The conditional pattern matching

The element *ėle* of the form *(if ėle′ matches ṡamp var v̇arSet for ċond then ėle*$_1$ *else ėle*$_2$*)* is called a conditional pattern matching and defined as follows: *traRel(ėle, ṫra)* if and only if

1. either there exist *ṡub* and *ṡta* such that *sup(ṡub)* = *v̇arSet*, *ėle′* is an instance in ⟦*ṡamp, ṡub*⟧, *traRel(ṫra($exeEle* ← *subF*⟦*ṡub*⟧*(ċond)), ṡta), ṡta($val)* ≠ *(-1:notMatch, 1:exception)*, and *traRel(ṡta($exeEle* ← *subF*⟦*ṡub*⟧*(ėle*$_1$*)), ṫra(2))*.
2. or the condition 1 does not assert, and *traRel(ṫra(1)($exeEle* ← *ėle*$_2$*), ṫra(2))*.

The elements *ėle′*, *ṡamp*, *v̇arSet*, *ċond*, *ėle*$_1$, and *ėle*$_2$ are called a matched element, pattern, set of variables, condition, *then*-branch and *else*-branch in ⟦*ėle*⟧.

Thus, the element *ėle* executes the instance of the *then*-branch *ėle*$_1$ in ⟦*ṡub*⟧, if *ėle′* is the instance of the sample *ṡamp* in ⟦*ṡub*⟧, and the instance of *ċond* in ⟦*ṡub*⟧ does not return the exception *(-1:notMatch, 1:exception)*. Otherwise, the element *ėle* executes the *else*-branch *ėle*$_2$.

The element *ėle* of the form *(if ėle′ matches ṡamp var v̇arSet for ċond then ėle*$_1$*)* is a shortcut for the element *(if ėle′ matches ṡamp var v̇arSet for ċond then ėle*$_1$ *else (skip))*.

The element *ėle* of the form *(if ėle′ matches ṡamp var v̇arSet where ċond then ėle*$_1$ *else ėle*$_2$*)* is a shortcut for the element *(if ėle′ matches ṡamp var v̇arSet for (if ċond then (skip) else ($val ::= (-1:notSelected, 1:exception))) then ėle*$_1$ *else ėle*$_2$*)*. The element *ċond* is called a condition in ⟦*ėle*⟧.

5.12. Iterators

The element *ėle* of the form *(foreach v̇ar in ėle′ do ḃody)* is called an iterator and defined as follows: *traRel(ėle, ṫra)* if and only if there exists *ṡta* such that *traRel(ṫra(1)($exeEle* ← *ėle′), sta)*, and

- *ṡta($val)* is an empty structure, and *ṫra(2)* = *ṡta*, or
- *ṡta($val)* ∈ *exc*, and *ṫra(2)* = *ṡta*, or
- *ṡta($val)* ∈ *ato* ∪ *labStr* \ *exc*, and *ṫra(2)* = *ṡta($val* ← *(0:ėle, 1:exception))*, or
- *ṡta($val)* ∈ *ordStr* ∪ *unoStr*, and
  traRel(ṡta($exeEle ← *(hvar v̇ar (foreach1 v̇ar in ṡta($val) do ḃody))), ṫra(2))*.

The elements *v̇ar*, *ėle′* and *ḃody* are called an iteration variable, iteration structure specifier and body in ⟦*ėle*⟧, respectively.

The element *(foreach1 v̇ar in ėle′ do ḃody)* is defined by the rules:

```
(if (foreach1 x in () do y) var (x, y) then (skip))
```

```
(if (foreach1 x in {} do y) var (x, y,) then (skip))

(if (foreach1 x in (v .:: w) do z) var (x, y, v, w)
 then (seq (x ::= 'v) z (foreach1 x in w do z)))

(if (foreach1 x in (v .::u w) do z) var (x, y, v, w)
 then (seq (x ::= 'v) z (foreach1 x in w do z)))
```

Thus, the element $\dot{e}le$ executes sequentially $\dot{b}ody$ in $[\![var{:}\dot{v}ar,\ val{:}\dot{e}le'']\!]$ for elements of the structure $\dot{s}tr$, where $\dot{s}tr$ is the value of $\dot{e}le'$. Executing $\dot{b}ody$ in $[\![var{:}\dot{v}ar,\ val{:}\dot{e}le'']\!]$ is executing $\dot{b}ody$ when the value of the variable $\dot{v}ar$ is equal to $\dot{e}le''$.

5.13. The element *throw*

The element $\dot{e}le$ of the form *(throw* $\dot{b}ody$*)* is defined by the rule:

(if (throw x) var (x) where (x is exception) then ($val ::= 'x))

The element $\dot{b}ody$ is called a body in $[\![\dot{e}le]\!]$.

5.14. Branching

The branching elements specify the order of execution of elements called branches and what branches are executed.

The element $\dot{e}le$ of the form *(orBranching* $\dot{e}le_1$ *...* $\dot{e}le_{\dot{n}at0}$*)* is called an or-branching and defined as follows: *traRel(*$\dot{e}le$*,* $\dot{t}ra$*)* if and only if either $\dot{n}at0 = 0$, and $\dot{t}ra(1) = \dot{t}ra(2)$, or $\dot{n}at0 > 0$, and there exists $\dot{s}ta$ such that *traRel(*$\dot{t}ra(1)$*($exeEle* $\leftarrow$ $\dot{e}le_1$*),* $\dot{s}ta$*)*, and

- $\dot{s}ta$*($val) = (-1:failBranch, 1:execution)*, and *traRel(*$\dot{s}ta$*($exeEle* $\leftarrow$ *(orBranching or* $\dot{e}le_2$ *...* $\dot{e}le_{\dot{n}at}$*)),* $\dot{t}ra(2)$*)*, or
- $\dot{s}ta$*($val)* $\neq$ *(-1:failBranch, 1:execution)*, and $\dot{t}ra(2) = \dot{s}ta$.

The elements $\dot{e}le_1$, ..., $\dot{e}le_{\dot{n}at}$ are called branches in $[\![\dot{e}le]\!]$.

Thus, the element $\dot{e}le$ executes branches sequentially until the next branch is normally executed, i. e. is executed without returning the exception *(-1:failBranch, 1:execution)*.

The element $\dot{e}le$ of the form *(andBranching* $\dot{e}le_1$ *...* $\dot{e}le_{\dot{n}at0}$*)* is called an and-branching and defined as follows: *traRel(*$\dot{e}le$*,* $\dot{t}ra$*)* if and only if either $\dot{n}at0 = 0$, and $\dot{t}ra(1) = \dot{t}ra(2)$, or $\dot{n}at0 > 0$, and there exists $\dot{s}ta$ such that *traRel(*$\dot{t}ra(1)$*($exeEle* $\leftarrow$ $\dot{e}le_1$*),* $\dot{s}ta$*)*, and

- *ṡta($val)* $\notin$ *exc* $\setminus$ *{(-1:stopBranch, 1:execution)}*, and *traRel(ṡta($exeEle* $\leftarrow$ *(orBranching or* $\dot{e}le_2$ ... $\dot{e}le_{\dot{n}at}$*)), ṫra(2))*, or
- *ṡta($val)* $\in$ *exc* $\setminus$ *{(-1:stopBranch, 1:execution)}*, and $\dot{t}ra(2) = \dot{s}ta$.

The elements $\dot{e}le_1$, ..., $\dot{e}le_{\dot{n}at}$ are called branches in $[\![\dot{e}le]\!]$.

Thus, the element $\dot{e}le$ executes branches sequentially until the next branch return an exception which is distinct from the exception *(-1:stopBranch, 1:execution)*.

6. Basic operations in CTSL

Basic operations in CTSL are defined in this section. Let *ope* be a set of operations.

6.1. Boolean operations

The set *ato* includes atoms *true* and *false* which specify the corresponding boolean values.

The element $\dot{e}le$ of the form $(\dot{e}le_1$ *and* $\dot{e}le_2)$ specifies the boolean operation of conjunction. Semantics of $\dot{e}le$ coincides with semantics of the element

(if ele_1 *then (if (*ele_2*) then 'true else 'false) else 'false).*

The elements $(\dot{e}le_1$ $\dot{o}pe$ $\dot{e}le_2)$, where $\dot{o}pe \in$ *{or, =>, <=>}* specifying the boolean operations of disjunction, implication and equivalence are defined in the similar way.

The element $\dot{e}le$ of the form $(not\ \dot{e}le')$ specifies the boolean operation of negation. Semantics of $\dot{e}le$ coincides with semantics of the element $(if\ \dot{e}le'\ then$ *'false else 'true).*

6.2. Equality and inequality of elements

The element $\dot{e}le$ of the form $(\dot{e}le_1 = \dot{e}le_2)$ specifies the operation of equality $=$ on elements. Semantics of $\dot{e}le$ coincides with semantics of the pseudoelement

```
(hvar ($x, $y)
 (seq ($x ::= ėle1) ($y ::= ėle2)
      (if ⟦$sta($x) = $sta($y)⟧ then ($val ::= 'true) else ($val ::= 'false)))
```

Pseudoelements are extension of elements by constructs $[\![\dot{o}bj]\!]$, where $\dot{o}bj$ is either a property or an expression. The object *$sta* denotes the current state.

The element $\dot{e}le$ of the form $(\dot{e}le_1$ *!=* $\dot{e}le_2)$ specifies the operation of inequality *!=* on elements. It is defined in the similar way.

6.3. Integer operations and relations

The element *ėle* of the form *(ėle*$_1$ *+ ėle*$_2$*)* specifies the integer addition *+*. Semantics of *ėle* coincides with semantics of the pseudoelement

```
(hvar ($x, $y)
 (seq
  ($x ::= ėle1) (if ⌈$sta($x) ∉ int⌉ then ($val ::= '(0:ėle, 1:exception)))
  ($y ::= ėle1) (if ⌈$sta($y) ∉ int⌉ then ($val ::= '(0:ėle, 1:exception)))
  ($val ::= ⌈$sta($x) + $sta($y)⌉)))
```

The element *ėle* of the form *(ėle*$_1$ *div ėle*$_2$*)* specifies the quotient of *ėle*$_1$ divided by *ėle*$_2$. Semantics of *ėle* coincides with semantics of the pseudoelement

```
(hvar ($x, $y)
 (seq
  ($x ::= ėle1) (if ⌈ṡta($x) ∉ int⌉ then ($val ::=  '(0:ėle, 1:exception)))
  ($y ::= ėle2)
  (if ⌈ṡta($y) ∉ int \ {0}⌉ then ($val ::=  '(0:ėle, 1:exception)))
  ($val ::= ⌈$sta($x) div $sta($y)⌉)))
```

The elements *(ėle*$_1$ *ȯpe ėle*$_2$*)*, where *ope* ∈ *{-, *, mod}* specifying the integer operations -, *** and *mod* are defined in the similar way.

The element *ėle* of the form *(ėle*$_1$ *< ėle*$_2$*)* specifies the integer relation *<*. Semantics *ėle* coincides with semantics of the pseudoelement

```
(hvar ($x, $y)
 (seq
  ($x ::= ėle1) (if ⌈$sta($x) ∉ int⌉ then ($val ::= '(0:ėle, 1:exception)))
  ($y ::= ėle2) (if ⌈$sta($y) ∉ int⌉ then ($val ::= '(0:ėle, 1:exception)))
  (if ⌈$sta($x) < $sta($y)⌉ then 'true else 'false)))
```

The elements *(ėle*$_1$ *ȯpe ėle*$_2$*)*, where *ope* ∈ *{<=, >, >=}* specifying the integer relations *<=*, *>* and *>=* are defined in the similar way.

7. Conclusion

In this paper the language CTSL of CTSs is proposed and the following kinds of CTSs are defined: CTSs with transition values specifying values of transitions, CTSs with executable elements specifying elements which can be executed, CTSs with execution contexts specifying contexts in which elements are executed, CTSs with counters specifying generation of new

elements which are instances of the given concepts, CTSs with history variables specifying variables which store a history of changing the conceptual *$val*, CTSs with defined conceptuals specifying definitions of conceptuals, CTSs with transition rules specifying executed elements based on the pattern matching and reduction of their execution semantics to execution semantics of other elements, CTSs with types specifying types of elements and literals of these types. Basic predefined elements and operations used in applications of CTSs are also presented.

We plan to use the CTSL language to solve problems of designing and prototyping software systems as well as specification of operational and axiomatic semantics of programming languages.

In the case of specification of operational semantics of a programming language, a CTS specifies the abstract machine of the language.

In the case of specification of axiomatic semantics of a programming language, a CTS specifies a generator of verification conditions for programs in the language, based on its axiomatic semantics.

References

1. Anureev I.S. Conceptual Transition Systems // System Informatics. 2015. Vol. 5. P. 1–41.
2. Gurevich Y. Abstract state machines: An Overview of the Project // Foundations of Information and Knowledge Systems. Lect. Notes Comput. Sci. 2004. Vol. 2942. P. 6-13.
3. Gurevich Y. Evolving Algebras. Lipari Guide // Specification and Validation Methods. Oxford University Press, 1995. P. 9-36.
4. Anureev I.S. Operational Ontological Approach to Formal Programming Language Specification // Programming and Computer Software. 2009. Vol. 35. N 1. P. 35-42.
5. Anureev I.S. Ontological Transition Systems // Bulletin of the Novosibirsk Computing Center, Series Computer Science. 2007. Vol. 26. P. 1-17.
6. Anureev I.S. A Language of Actions in Ontological Transition Systems // Bulletin of the Novosibirsk Computing Center, Series Computer Science. 2007. Vol. 26. P. 19-38.
7. Anureev I.S. Domain-Specific Transition Systems and their Application to a Formal Definition of a Model Programming Language // Bulletin of the Novosibirsk Computing Center, Series Computer Science. 2014. Vol. 34. P. 23–42.
8. Anureev I.S. Conceptual Transition Systems // System Informatics. 2015. N 1. (In Russian). (To appear).
9. Anureev I.S. Kinds and Language of Conceptual Transition Systems // System Informatics. 2015. N 1. (In Russian). (To appear).
10. Huggins J. Abstract State Machines Web Page. URL: http://www.eecs.umich.edu/gasm (accessed: 01.09.2015).

11. AsmL: The Abstract State Machine Language. Reference Manual. 2002. URL: http://research.microsoft.com/fse/asml/doc/AsmL2 Reference.doc (accessed: 01.09.2015).
12. XasM — An Extensible, Component-Based Abstract State Machines Language. URL: http://xasm.sourceforge.net/XasmAnl00/XasmAnl00.html (accessed: 01.09.2015).