УДК 004.8

Operational conceptual transition systems and their application to development of conceptual operational semantics of programming languages[*]

Anureev I.S. (Institute of Informatics Systems)

In the paper the notion of the conceptual operational semantics of a programming language is proposed. This formalism represents operational semantics of a programming language in terms of its conceptual model based on conceptual transition systems. The special kind of conceptual transition systems, operational conceptual transition systems, oriented to specification of conceptual operational semantics of programming languages is defined, the extension of the language of conceptual transition systems CTSL for operational conceptual transition systems is described, and the technique of the use of the extended CTSL as a domain-specific language of specification of conceptual operational semantics of programming languages is proposed. The conceptual operational semantics for the family of sample programming languages illustrate this technique.

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

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.

The notion of the conceptual model of a programming language is proposed in [3]. This formalism describes the instantiation semantics at the conceptual level. The conceptual model is specified in terms of conceptual transition systems (CTSs) in the language of conceptual transition systems CTSL [3].

[*] Partially supported by RFBR under grants 15-01-05974 and 17-07-01600 and SB RAS interdisciplinary integration project No.15/10.

In this paper, we introduce the notion of the conceptual operational semantics of a programming language. This formalism describes the operational semantics of a programming language in terms of its conceptual model. The dynamic part of the operational semantics is defined in terms of the special kind of CTS, operational CTSs, in the extension of CTSL for operational CTSs. Thus, CTSL acts as a domain-specific language oriented to specification of conceptual operational semantics of programming languages.

The paper has the following structure. The preliminary concepts and notation are given in section 2. The concepts and definitions related to the pattern matching which operational CTSs are based on is given in section 3. The operational CTSs is defined in section 4. The extension of the CTSL language for operational CTSs is described in section 5. Semantics of basic executable elements in CTSL is defined in section 6. The definition of the conceptual operational semantics of a programming language is introduced, and the technique of development of conceptual operational semantics 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 $[\$f\ \$a.\$a1.\ \ldots\ .\$a\$n \coloneqq \$v]$ be a shortcut for $[\$f\ \$a \coloneqq [[\$f\ \$a]\ \$a1.\ \ldots\ .\$a\$n \coloneqq \$v]]$. 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. Pattern matching

General CTSs only defines the state structure. The special kinds of CTSs also refine the structure of the transition relation. The refinement of operational CTSs is based on the pattern matching on the state structure.

A function $\$su \in \$\$cs \rightarrow \$\$cs^*$ is called a substituton. Let $\$\$su$ be a set of substitutions. A function $sub \in \$\$su \times \$\$cs^* \rightarrow \$\$cs^*$ is a substitution function if it is defined by the following rules (the first proper rule is applied):

- if $\$cs \in [support\ \$su]$, then $[sub\ \$su\ \$cs] = [\$su\ \$cs]$;
- $[sub\ \$su\ \$ato] = \$ato$;
- $[sub\ \$su\ \$cs{:}\{\$t^+\}] = [sub\ \$su\ \$cs]{:}\{[sub\ \$su\ \$t^+]\}$;
- $[sub\ \$su\ \$cs{::}\{\$t^+\}] = [sub\ \$su\ \$cs]{::}\{[sub\ \$su\ \$t^+]\}$;
- $[sub\ \$su\ (\$cs^*)] = ([sub\ \$su\ \$cs^*])$;
- $[sub\ \$su\ \$cs^*] = [\$cs^*\ \$e\ \hookleftarrow^*\ [sub\ \$su\ \$e]]$.

A structure $\$p$ is a pattern in $⟦\$cs,\ \$su⟧$ if $[sub\ \$su\ \$p] = \$cs$. A structure $\$p$ is a pattern in $⟦\$cs⟧$ if $\$p$ is a pattern in $⟦\$cs,\ \$su⟧$ for some $\$su$. Let $\$\$p$ be a set of patterns. A structure $\$in$ is an instance in $⟦\$p,\ \$su⟧$ if $[sub\ \$su\ \$p] = \$in$. A structure $\$in$ is an instance in $⟦\$p⟧$ if $\$in$ is an instance in $⟦\$p,\ \$su⟧$ for some $\$su$. Let $\$\$in$ be a set of instances.

A structure $\$cs1$ is weakly equal to a structure $\$cs2$ ($\$cs1 =_w \$cs2$) if the following properties hold:

- if $\$cs1 \in \$\$ato$, then $\$cs2 \in \$\$ato$, and $\$cs1 = \$cs2$;
- if $\$cs1 = v1{::}\{t^+1\}$, then $\$cs2 = v2{::}\{t^+2\}$, $v1 =_w v2$, and $\{t^+1\} =_w \{t^+2\}$ for some $\$va2$ and $\$t^+2$;
- if $\$cs1 = v1{:}\{t^+1\}$, then $\$cs2 = v2{::}\{t^+2\}$, $v1 =_w v2$, and $\{t^+1\} =_w \{t^+2\}$ for some $\$va2$ and $\$t^+2$;
- if $\$cs1 \in \$\$ccs$, then $\$cs2 \in \$\$ccs$, $[len\,\$cs1] = [len\,\$cs2]$, and $[\$cs1 \,..\, \$n] =_w \$cs2 \,..\, \$n]$ for each $1 \le \$n \le [len\,\$cs1]$.

A structure set $\$set1$ is weakly equal to a structure set $\$set2$ ($\$set1 =_w \$set2$) if the following properties hold:

- if $\$set1 = \emptyset$, then $\$set2 = \emptyset$;
- if $\$cs1 \in \$set1$, then there exists $\$cs2 \in \$set2$ such that $\$cs1 =_w \$cs2$, and $\$set1 \setminus \{\$cs1\} =_w \$set2 \setminus \{\$cs2\}$.

An element $\$e$ is linear in $[\![\$e^*]\!]$ if $\$e1$ occurs in $\$e$ exactly once for each element $\$e1$ of $\$e^*$. An element $\$ps$ of the form ($\$p$ ($\$va^*$) ($\$sv^*$)) is a pattern specification if the elements of the sequence $\$va^*\;\$sv^*$ are pairwise distinct, and $\$p$ is linear in $[\![\$va^*\;\$sv^*]\!]$. The elements $\$p$, ($\$va^*$), and ($\$sv^*$) are called a pattern, state variable specification and sequence variable specification in $[\![\$ps]\!]$. The elements of $\$va^*$ and $\$sv^*$ are called state variables and sequence variables in $[\![\$ps]\!]$. Let $\$\$ps$ be a set of pattern specifications. Let $\$\$va$ and $\$\$sv$ be sets of state and sequence variables.

A structure $\$in$ is an instance in $[\![\$ps, \$su]\!]$ if $[support\;\$su] = \{\$va^*\} \cup \{\$sv^*\}$, $[\$su\;\$va] \in \$\$s$ for $\$va \in \{\$va^*\}$, $[\$su\;\$sv] \in \$\$s^*$ for $\$sv \in \{\$sv^*\}$, and $\$in = w\;[sub\;\$p[\![\$ps]\!]\;\$su]$. A structure $\$in$ is an instance in $[\![\$ps]\!]$ if $\$in$ is an instance in $[\![\$ps,\;\$su]\!]$ for some $\$su$.

A function $\$mt \in \$\$s \times \$\$ps \to \$\$su$ is a matching tactic if $[\$mt\;\$s\;\$ps] = \$su$ implies that $\$s$ is an instance in $[\![\$ps,\;\$su]\!]$. A structure $\$in$ is an instance in $[\![\$ps,\;\$mt,\;\$su]\!]$ if $[\$mt\;\$in\;\$ps] = \$su$. A structure $\$in$ is an instance in $[\![\$ps,\;\$mt]\!]$ if $\$in$ is an instance in $[\![\$ps,\;\$mt,\;\$su]\!]$ for some $\$su$.

A substitution $\$su$ is a matching result in $[\![\$ps,\;\$mt,\;\$in]\!]$ if $\$in$ is an instance in $[\![\$ps,\;\$mt,\;\$su]\!]$. A substitution $\$su$ is a matching result in $[\![\$ps,\;\$mt]\!]$ if $\$su$ is a matching result in $[\![\$ps,\;\$mt,\;\$in]\!]$ for some $\$in$. A value $\$v$ is a matching result in $[\![\$va, \$ps,\;\$mt,\;\$su, \$in]\!]$ if $\$in$ is an instance in $[\![\$ps,\;\$mt,\;\$su]\!]$, and $\$v = [\$su\;\$va]$. A value $\$v$ is a matching result in $[\![\$va, \$ps,\;\$mt,\;\$in]\!]$ if $\$v$ is a matching result in $[\![\$va, \$ps,\;\$mt,\;\$su, \$in]\!]$ for some $\$su$. Let $\$\$mr$ be a set of matching results.

4. Operational conceptual transition systems

Operational CTSs (OCTSs) are the special kind of CTSs used to describe operational semantics of program systems and programming languages. Let $\$\$octs$ be a set of OCTSs.

The structure of $\hookrightarrow_{[\![\$octs]\!]}$ is based on program transition relations, program transition rules and atomic transition relations. Program transition relations and program transition rules include a specification of a pattern and their application is based on the matching of the first element of a program with the pattern. Atomic transition relations are applied in the case of the empty program.

An object $\$ptr$ of the form $(\$ps,\ \$f)$ is a program transition relation in $[\![\$mt]\!]$ if $\$f \in \{\$su \cup (\{cstate\}\text{: }\$s,\ \{cvalue\}\text{: }\$v[\![\$s]\!]) \mid \$su \in \$\$mr[\![\$ps,\ \$mt]\!],\text{ and }\$s \in \$\$s\} \to \$\$tr$. Thus, $\$ptr$ specifies a parametric transition relation, where the values of the parameter are the results of the pattern matching. The special elements $\{cstate\}$ and $\{cvalue\}$ refer to the current state and the value in the current state. Let $\$\$ptr$ be a set of program transition relation. The objects $\$p[\![\$ps]\!]$, $(\$va^*[\![\$ps]\!])$, $(\$sv^*[\![\$ps]\!])$ and $\$f$ are called a pattern, state variable specification, sequence variable specification and value in $[\![\$ptr]\!]$. The elements of $\$va^*$ and $\$sv^*$ are called state and sequence variables in $[\![\$ptr]\!]$. If $\$su$ is the result of matching the first element $\$e$ of the program $\$p[\![\$s]\!]$ with the pattern of $\$ptr$, then a transition from $\$s$ to $\$s1$ initiated by $\$ptr$ and denoted by $\$s \hookrightarrow_{[\![\$ptr,\ \$su]\!]} \$s1$ is defined as $(\$s,\ \$s1) \in [\$f\ \$su \cup (\{cstate\}\text{: }\$s,\ \{cvalue\}\text{: }\$v[\![\$s]\!])]$. Let $\$s \hookrightarrow_{[\![\$f[\![\$ptr]\!]]\!]} \$s1$ denote $\$s \hookrightarrow_{[\![\$ptr,\ \$su]\!]} \$s1$ for some $\$su$.

A partial function $\$ptrs \in \$e \to \$\$ptr$ is a program transition relation specification if $[support\ \$ptrs]$ is finite. A relation $\$ptr$ is a relation in $[\![\$ptrs]\!]$ if $[\$ptrs\ \$na] = \$ptr$ for some $\$na \in \$\$e$. An element $\$na$ is a name in $[\![\$ptr,\ \$ptrs]\!]$ if $[\$ptrs\ \$na] = \$atr$. An element $\$na$ is a name in $[\![\$ptrs]\!]$ if $\$na$ is a name in $[\![\$ptr,\ \$ptrs]\!]$ for some $\$atr$. Thus, $\$ptrs$ defines a finite set of named program transition relations.

An element $\$r$ of the form $(\$ps\ \$b)$ is called a (program) transition rule. The objects $\$p[\![\$ps]\!]$, $(\$va^*[\![\$ps]\!])$, $(\$sv^*[\![\$ps]\!])$ and $\$b$ are called a pattern, state variable specification, sequence variable specification and body in $[\![\$r]\!]$. The elements of $\$va^*$ and $\$sv^*$ are called state and sequence variables in $[\![\$r]\!]$. Let $\$\$r$ be a set of transition rules. If $\$su$ is the result of matching the first element $\$e$ of the program $\$p[\![\$s]\!]$ with the pattern of $\$r$, then a transition from $\$s$ initiated by $\$r$ replaces $\$e$ in $\$p$ by $[sub\ \$su \cup (\{cstate\}\text{: }\$s,\ \{cvalue\}\text{: }\$v[\![\$s]\!])\ \$b]$.

A structure $\$rs$ is a rule specification if $[\$rs\ .\ \{\$na\}] \in \$\$r \cup \{und\}$ for each $\$na \in \$\$e$. A rule $\$r$ is a rule in $[\![\$rs]\!]$ if $[\$rs\ .\ \{\$na\}] = \$r$ for some $\$na \in \$\$e$. An element $\$na$ is a name in

$[\![\$r,\ \$rs]\!]$ if $[\$rs\ .\ \{\$na\}]\ =\ \$r$. An element $\$na$ is a name in $[\![\$rs]\!]$ if $\$na$ is a name in $[\![\$r,\ \$rs]\!]$ for some $\$r$. Thus, $\$rs$ defines a finite set of named transition rules.

A structure $\$rs$ is a rule specification in $[\![\$s]\!]$ if $\$rs = [\$s\ .\ \{rules\}]$, and $[\$rs\ .\ \{\$na\}] \in \$\$r \cup \{und\}$ for each $\$na \in \$\$e$. It specifies the set of named transition rules in $[\![\$s]\!]$.

Let $[support\ \$ptrs] \cap [support\ \$rs] = \emptyset$.

A structure $\$pto$ of the form $(\$na^*)$ is a program transition order in $[\![\$ptrs,\ \$rs]\!]$ if $\{\$na^*\} \subseteq [support\ \$ptrs] \cup [support\ \$rs]$, and the elements of $\$na^*$ are pairwise distinct. It specifies the order of application of program transition relations and transition rules, i. e. the order of matching the first element of the program with their patterns.

A structure $(\$na^*)$ is a program transition order in $[\![\$s]\!]$ if $(\$na^*) = [\$s\ .\ \{(program\ transition\ order)\}]$, and the elements of $\$na^*$ are pairwise distinct. It specifies the order of application of program transition relations and transition rules in $[\![\$s]\!]$.

A relation $\$atr \in \$\$tr$ is called an atomic transition relation. Let $\$\$atr$ be a set of atomic transition relations. If $\$p[\![\$s]\!] = ()$, then a transition from $\$s$ to $\$s1$ initiated by $\$atr$ and denoted by $\$s \hookrightarrow_{[\![\$atr]\!]} \$s1$ is defined as $(\$s,\ \$s1) \in \$atr$.

A partial function $\$atrs \in \$\$s \to \$\$atr$ is an atomic transition relation specification if $[support\ \$atrs]$ is finite. A relation $\$atr$ is a relation in $[\![\$atrs]\!]$ if $[\$atrs\ \$na]\ =\ \$atr$ for some $\$na \in \$\$e$. An element $\$na$ is a name in $[\![\$atr,\ \$atrs]\!]$ if $[\$atrs\ nm] = \$atr$. An element $\$na$ is a name in $[\![\$atrs]\!]$ if $\$na$ is a name in $[\![\$atr,\ \$atrs]\!]$ for some $\$atr$. Thus, $\$atrs$ defines a finite set of named atomic transiton relations.

A structure $\$ator$ of the form $(\$na^*)$ is an atomic transition order in $[\![\$atrs]\!]$ if $\{\$na^*\} \subseteq [support\ \$atrs]$, and the elements of $\$na^*$ are pairwise distinct. It specifies the order of application of atomic transition relations.

A structure $(\$na^*)$ is an atomic transition order in $[\![\$s]\!]$ if $(\$na^*) = [\$s\ .\ \{(atomic\ transition\ order)\}]$, and the elements of $\$na^*$ are pairwise distinct. It specifies the order of application of atomic transition relations in $[\![\$s]\!]$.

Let $\$\$bi$ be a set of elements called backtracking invariants.

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

A tuple $\$octs$ of the form $(\$ato,\ \$\$is, \$ptrs,\ \$rs,\ \$pto,\ \$atrs,\ \$ator,\ \$\$bi,\ \$mt)$ is an operational CTS if the system $\$cts$ of the form $(\$\$at,\ \$\$is, \hookrightarrow)$ is a CTS with backtracking in $[\![\$\$bi]\!]$ and with direct stop, $[support\ \$ptrs]\ \cap\ [support\ \$rs]\ =\ \emptyset$, $\$s$ is consistent for each $\$s \in \$\$rs[\![\hookrightarrow]\!]$, and $\hookrightarrow$ is defined by the following rules (the first proper rule is applied):

- if $\$ptr = [\$ptrs\ \$na]$, $\$e$ is an instance in $[\![\$ps[\![\$ptr]\!], \$mt, \$su]\!]$, and $\$s \hookrightarrow_{[\![\$ptr, \$su]\!]} \$s1$, then
  $((execute\ program\ element)\ \$e\ (\$na\ \$na^*))\ \$e^* \#\ \$s\ \hookrightarrow$
  $(bactracking\ \$s\ ((execute\ program\ element)\ \ \$e\ (\$na^*)))\ \$e^*\ \#\ \$s1;$
- if $\$ptr = [\$ptrs\ \$na]$, and $\$e$ is not an instance in $[\![\$ps[\![\$ptr]\!], \$mt]\!]$, then $((execute\ program\ element)\ \$e\ (\$na\ \$na^*))\ \$e^*\ \#\ \$s\ \hookrightarrow$
  $((execute\ program\ element)\ \$e\ (\$na^*))\ \$e^*\ \#\ \$s;$
- if $(\$r = [\$rs\ .\ \{\$na\}]$, or $\$r = [[\$s\ .\ \{rules\}]\ .\ \{\$na\}])$, and $\$e$ is an instance in $[\![\$ps[\![\$r]\!], \$mt, \$su]\!]$, then
  $((execute\ program\ element)\ \$e\ (\$na\ \$na^*))\ \$e^*\ \#\ \$s$
  $\hookrightarrow_{[sub\ \$su \cup (\{cstate\}:\$s, \{cvalue\}:\$v[\![\$s]\!])\ \$b[\![\$r]\!]]}$
  (backtracking $s ((execute program element) $e ($na^*))) $e^* # $s;
- if $(\$r = [\$rs\ .\ \{\$na\}]$, or $\$r = [[\$s\ .\ \{rules\}]\ .\ \{\$na\}])$, and $\$e$ is not an instance in $[\![\$ps[\![\$r]\!], \$mt]\!]$, then
  $((execute\ program\ element)\ \$e\ (\$na\ \$na^*))\ \$e^*\ \#\ \$s \hookrightarrow$
  $((execute\ program\ element)\ \$e\ (\$na^*))\ \$e^*\ \#\ \$s;$
- $((execute\ program\ element)\ \$e\ ())\ \$e^*\ \#\ \$s \hookrightarrow \$e^*\ \#\ und\ \#\ \$s;$
- if $\$atr = [\$atrs\ \$na]$, $\$s \hookrightarrow [\![\$atr]\!]\ \$s1$, and $\$p[\![\$s1]\!] \neq ()$, then $((execute\ atomic\ transition)\ (\$na\ \$na^*))\ \#\ \$s \hookrightarrow \$s1;$
- if $\$atr = [\$atrs\ \$na]$, $\$s \hookrightarrow [\![\$atr]\!]\ \#\ \$v\ \#\ \$s1$, and $\$p[\![\$s1]\!] = ()$, then $((execute\ atomic\ transition)\ (\$na\ \$na^*))\ \#\ \$s \hookrightarrow$
  $(backtracking\ \$s\ ((execute\ atomic\ transition)\ (\$na^*)))\ \#\ \$s1;$
- $((execute\ atomic\ transition)\ ())\ \#\ \$s \hookrightarrow [\$s\ true{:}\{stop\}];$
- $\$e\ \$e^*\ \#\ \$s\ \hookrightarrow$
  $((execute\ program\ element)\ \$e\ [\$s\ ..\ (program\ transition\ order)])\ \$e^*\ \#\ \$s;$
- $\#\ \$s\ \hookrightarrow ((execute\ atomic\ transition),\ [\$s\ ..\ (atomic\ transition\ order)])\ \#\ \$s.$

A state $\$s$ is consistent in $[\![\$cts]\!]$ if the following properties hold:

- the set of bactracking invariants in $[\![\$s]\!]$ is finite;
- $[support\ \$ptrs] \cap [support\ [\$s\ .\ \{rules\}]] = \emptyset;$
- $[support\ \$rs] \cap [support\ [\$s\ .\ \{rules\}]] = \emptyset;$
- if $\$na1 \prec_{[\![\$pto]\!]} \$na2$, $\$na1 \in [\$s\ .\ \{(program\ transition\ order)\}]$, and $\$na2 \in [\$s\ .\ \{(program\ transition\ order)\}]$, then $\$na1 \prec_{[\![[\$s\ .\ \{(program\ transition\ order)\}]]\!]} \$na2;$

if $\$na1 \prec_{[\![\$ator]\!]} \$na2$, $\$na1 \in [\$s\,.\,\{(atomic\ transition\ order)\}]$, and $\$na2 \in [\$s\,.\,\{(atomic\ transition\ order)\}]$, then $\$na1 \prec_{[\![[\$s\,.\,\{(atomic\ transition\ order)\}]]\!]} \$na2$.

5. The CTSL language

The CTSL language is extended for operational CTSs by adding transition rules and extended transition rules.

The transition rule $((\$p,\ (\$va^*),\ (\$sv^*)),\ \$b)$ with the name $\$na$ is represented in CTSL[o] by the state $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ then\ \$b)::\{\$na\}$.

Extended transition rules are transition rules enriched by the mechanisms of evaluation of pattern variable matching results, imposition of constraints on pattern variable matching results and their values and propagation of abnormal values (undefined values and exceptions) from pattern variable matching results and the attribute $value$.

Let $\{\$eva^*\} \subseteq \{\$va^*\}$, the elements of the sequence $\$eva^*$ are pairwise disjoint, $\$set = \{\$eva :: \{*\} \mid \$eva \in \$eva^*\}$, $\{\$va^*1\} \cup \{\$va^*2\} \cup \{\$va^*3\} \subseteq \{\$va^*\} \cup \$set$, the elements of the sequence $\$va^*1\ \$va^*2\ \$va^*3$ are pairwise disjoint, and $\$se \in \{[se],\ und,\ exc,\ abn\}$. The state
$(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se$
$where\ \$co\ then\ \$b)$
is called an extended rule. It is defined as follows:

- If $\$co \neq true$, then
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ und\ (\$va^*1)\ exc\ (\$va^*2)\ abn\ (\$va^*3)\ \$se$
  $where\ \$co\ then\ \$b)$
  is a shortcut for
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ und\ (\$va^*1)\ exc\ (\$va^*2)\ abn\ (\$va^*3)\ \$se$
  $where\ true\ then\ (if\ \$co\ then\ \$b\ else\ und))$.
- The rule
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ und\ (\$va^*1)\ exc\ (\$va^*2)$
  $abn\ (\$va^*31\ \$eva::\{*\}\ \$va^*32)\ \$se\ where\ true\ then\ \$b)$
  is a shortcut for
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ und\ (\$va^*1)\ exc\ (\$va^*2)$
  $abn\ (\$va^*31\ \$va^*32)\ \$se\ where\ true\ then\ (if\ (\$eva :: \{*\}\ is\ abnormal)$
  $then\ \$eva::\{*\}\ else\ \$b))$.
- If $\{\$va^*3\} \cap \$set = \emptyset$, then
  $(rule\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ und\ (\$va^*1)$

$exc\ (\$va^{*}21\ \$eva::\{*\}\ \$va^{*}22)\ abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ \$b)$

is a shortcut for

$(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\$eva^{*})\ und\ (\$va^{*}1)\ exc\ (\$va^{*}21\ \$va^{*}22)$

$abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ (if\ (\$eva::\{*\}\ is\ exception)\ then\ \$eva::\{*\}\ else\ \$b))$.

- If $(\{\$va^{*}2\} \cup \{\$va^{*}3\}) \cap \$set = \emptyset$, then

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\$eva^{*})\ und\ (\$va^{*}11\ \$eva :: \{*\}\ \$va^{*}12)$

  $exc\ (\$va^{*}2)\ abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ \$b)$

  is a shortcut for

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\$eva^{*})\ und\ (\$va^{*}11\ \$va^{*}12)\ exc\ (\$va^{*}2)$

  $abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ (if\ (\$eva::\{*\}\ is\ undefined)\ then\ \$eva::\{*\}\ else\ \$b))$.

- If $(\{\$va^{*}1\} \cup \{\$va^{*}2\} \cup \{\$va^{*}3\}) \cap \$set = \emptyset$, then

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\$eva^{*}\ \$eva :: \{*\})\ und\ (\$va^{*}1)\ exc\ (\$va^{*}2)$

  $abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ \$b)$

  is a shortcut for

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\$eva^{*})\ und\ (\$va^{*}1)\ exc\ (\$va^{*}2)$

  $abn\ (\$va^{*}3)\ \$se\ where\ true\ then\ (let\ w\ be\ \$eva\ in\ (subst\ (\$eva :: \{*\}: w)\ \$b)))$,

  where w is a new state that does not occur in the initial form.

- If $(\{\$va^{*}1\} \cup \{\$va^{*}2\} \cup \{\$va^{*}31,\ \$va,\ \$va^{*}32\}) \cap \$set = \emptyset$, then

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\ )\ und\ (\$va^{*}1)\ exc\ (\$va^{*}2)$

  $abn\ (\$va^{*}31\ \$va\ \$va^{*}32)\ \$se\ where\ true\ then\ \$b)$

  is a shortcut for

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\ )\ und\ (\$va^{*}1)\ exc\ (\$va^{*}2)$

  $abn\ (\$va^{*}31\ \$va^{*}32)\ \$se\ where\ true\ then\ (if\ (\$va\ is\ abnormal)\ then\ \$va\ else\ \$b))$.

- If $(\{\$va^{*}1\} \cup \{\$va^{*}21,\ \$va,\ \$va^{*}22\}) \cap \$set = \emptyset$, then

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\ )\ und\ (\$va^{*}1)\ exc\ (\$va^{*}21\ \$va\ \$va^{*}22)$

  $abn\ (\ )\ \$se\ where\ true\ then\ \$b)$

  is a shortcut for

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\ )\ und\ (\$va^{*}1)\ exc\ (\$va^{*}21\ \$va^{*}22)\ abn\ (\ )\ \$se$

  $where\ true\ then\ (if\ (\$va\ is\ exception)\ then\ \$va\ else\ \$b))$.

- If $\{\$va^{*}11,\ \$va,\ \$va^{*}12\} \cap \$set = \emptyset$, then

  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\ )\ und\ (\$va^{*}11\ \$va\ \$va^{*}12)\ exc\ (\ )\ abn\ (\ )\ \$se$

  $where\ true\ then\ \$b)$

  is a shortcut for

$(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\$va^{*}11\ \$va^{*}12)\ exc\ (\,)\ abn\ (\,)\ \$se$
$where\ true\ then\ (if\ (\$va\ is\ undefined)\ then\ \$va\ else\ \$b))$.

- The rule
  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ abn\ where\ true\ then\ \$b)$ is a shortcut for
  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ where\ true$
  $then\ (if\ (cvalue\ is\ abnormal)\ then\ else\ \$b)$.
- The rule
  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ exc\ where\ true\ then\ \$b)$ is a shortcut for
  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ where\ true$
  $then\ (if\ (cvalue\ is\ exception)\ then\ else\ \$b)$.
- The rule
  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ und\ where\ true\ then\ \$b)$ is a shortcut for
  $(rule\ \$p\ var\ (\$va^{*})\ seq\ (\$sv^{*})\ val\ (\,)\ und\ (\,)\ exc\ (\,)\ abn\ (\,)\ where\ true$
  $then\ (if\ (cvalue\ is\ undefined)\ then\ else\ \$b)$.

A pattern variable $\$va$ is evaluated if the matching result for \$va is evaluated. The sequence $\$eva^{*}$ contains evaluated pattern variables. The special variable $\$eva::\{*\}$ references to the value of the matching result for \$va. A pattern variable $\$va$ is quoted if the matching result for \$va is not evaluated.

The state $\$co$ imposes of constraints on the values of the variables $\$va^{*}$, $\$sv^{*}$, $\$eva::\{*\}^{*}$.

The undefined value und is propagated through the variables $\$v^{*}1$. Exceptions are propagated through the variables $\$v^{*}2$. Abnormal values are propagated through the variables $\$v^{*}3$.

The sequence $\$se$ specifies propagation of abnormal values through the attribute $value$. The undefined value is propagated through the attribute $value$ when $\$se = und$. Exceptions are propagated through the attribute $value$ when $\$se = exc$. Abnormal values are propagated through the attribute $value$ when $\$se = abn$.

The executable elements $(if\ \$con\ then\ \$e^{*}1\ else\ \$e^{*}2)$ and $(let\ \$va\ be\ \$e^{*}1\ in\ \$e^{*}2)$ are defined in section 6.7. The executable elements $(\$e\ is\ abnormal)$, $(\$e\ is\ exception)$ and $(\$e\ is\ undefined)$ are defined in section 7.4.

Let $\$\$er$ be a set of extended transition rules.

The objects $var\ (\$va^*)$, $seq\ (\$sv^*)$, $val\ (\$eva^*)$, $abn\ (\$va^*1)$, $und\ (\$va^*2)$, $exc\ (\$va^*3)$ and $where\ \$co$ in extended transition rules can be omitted. The omitted objects correspond to $var\ (\,)$, $seq\ (\,)$, $val\ (\,)$, $abn\ (\,)$, $und\ (\,)$, $exc\ (\,)$ and $where\ true$.

6. Semantics of executable elements in CTSL

To define operational semantics of executable elements in CTSL the special denotations for program and atomic transition relations are introduced.

Let $(transition\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ then\ \$f)::\{\$na\}$ denote the program transition relation $((\$p,\ (\$va^*),\ (\$sv^*)),\ \$f)$ with the name $\$na$. The objects $var\ (\$va^*)$ and $seq\ (\$sv^*)$ can be omitted. The omitted objects correspond to $var\ ()$ and $seq\ ()$.

Let $(atomic\ transition\ \$f)::\{\$na\}$ denotes the atomic transition relation defined by the characteristic function $\$f \in \$s \times \$s \to \$b$ with the name $\$na$.

For simplicity, we omit the names of transition relations and transition rules.

6.1. Values

The executable elements handling the transition value are defined in this section.

An element $\$e$ of the form $\$v::\{q\}$ is called a quoted element. It is defined as follows:

$(rule\ v::\{q\}\ \ var\ (v)\ abn\ then\ v::\{q\}::\{transition\})$;

$(transition\ v::\{q\}::\{transition\}\ var\ (v)\ then\ \$f)$,

where $\$v::\{q\}::\{transition\}; \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ \$v\ \#\ \$s$.

The value v is called a quoted value in $[\![\$e]\!]$. The element $\$e$ returns the quoted value v.

The element und is defined by the rule

$(rule\ und\ abn\ then\ und::\{q\})$.

The element ex is defined by the rule

$(rule\ v::\{exc\}\ var\ (v)\ abn\ then\ v::\{exc\}::\{q\})$.

The element $(\$e\ is\ undefined)$ specifies that $\$e$ equals und. It is defined by the rule

$(rule\ (e\ is\ undefined)\ var\ (e)\ abn\ then\ (e::\{q\} = und))$.

The element $(\$e\ is\ defined)$ specifies that $\$e$ does not equal und. It is defined by the rule

$(rule\ (e\ is\ defined)\ var\ (e)\ abn\ then\ (e::\{q\}\ ! = und))$.

The element $(\$e\ is\ exception)$ specifies that $\$e$ is an exception. It is defined by the rule

$(rule\ (e\ is\ exception)\ var\ (e)\ abn\ then\ (e\ is\ exception)::\{transition\})$;

$(transition\ (e\ is\ exception)::\{transition\}\ var\ (e)\ then\ \$f)$,

where

$(e\ is\ exception) :: \{transition\}; \$e^* \ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^* \# [if\ [\$e \in \$\$exc]\ then\ true\ else\ und] \# \$s.$

The element $(\$e\ is\ abnormal)$ specifies that $\$e$ is abnormal. It is defined by the rule

$(rule\ (e\ is\ abnormal)\ var\ (e)\ abn\ then\ ((e\ is\ undefined)\ or\ (e\ is\ exception)));$

The element $(\$e\ is\ normal)$ specifying that $\$e$ is normal. It is defined by the rule

$(rule\ (e\ is\ normal)\ var\ (e)\ abn\ then\ ((e\ is\ defined)\ and\ (not\ (e\ is\ exception))));$

The element $\$e$ of the form $(catch::\{und\}\ \$va\ \$e^*)$ is called a value handler. It is defined as follows:

$(transition\ (catch::\{und\}\ va\ e\_s)\ var\ (va)\ seq\ (e\_s)\ then\ \$f),$

where $(catch::\{und\}\ \$va\ \$e^*); \$e^*1 \# \$v \# \$s \hookrightarrow_{[\![\$f]\!]} (sub\ (\$va{:}\ \$v)\ \$e^*)\ e^*1 \# true \# \$s.$

The elements $\$va$ and $\$e^*1$ are called a variable and body in $[\![\$e]\!]$. The element $\$e$ replaces all occurences of the variable $\$va$ in the body $\$e^*1$ by the current value, resets the current value to $true$ and executes the modified body.

The element $\$e$ of the form $(catch\ \$va\ \$e^*)$ is called an exception handler. It is defined as follows:

$(rule\ (catch\ va\ e\_s)\ var\ (va)\ seq\ (e\_s)\ und\ then\ (catch::\{und\}\ va\ e\_s)),$

The elements $\$va$ and $\$e^*$ are called a variable and body in $[\![\$e]\!]$. If the current value is defined, the element $\$e$ replaces all occurences of the variable $\$va$ in the body $\$e^*1$ by the current value, resets the current value to $true$ and executes the modified body. It propagates und.

The element $(current\ value)$ returns the current value. It is defined by the rule

$(rule\ (current\ value)\ abn\ then\ cvalue::\{q\}).$

The element $((to\ value)\ \$e)$ replaces the current value to $\$v$, where $\$v$ is the value of $\$e$. It is defined as follows:

$(rule\ ((to\ value)\ e)\ var\ (e)\ val\ (e)\ then\ ((to\ value)\ e::\{*\})::\{transition\});$

$(transition\ ((to\ value)\ v)::\{transition\}\ var\ (v)\ then\ \$f),$

where $((to\ value)\ \$v)::\{transition\};\ \$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# \$v \# \$s.$

The element ((catch exception) t) catches an exception of the type $\$t$. It is defined by the rule

$(rule\ ((catch\ exception)\ t)\ var\ (t)\ und$

$then\ (catch\ va\ ($

$if\ ((va\ is\ exception)\ and\ ((va::\{q\}\ .\ \{type\})\ = t::\{q\}))$

$then\ ((to\ value)\ true)\ else\ ((to\ value)\ va::\{q\})))).$

6.2. Integers

The executable elements handling integers are defined in this section.

The element ($\$e\ is\ nat$) specifies that $\$e$ is a natural number. It is defined as follows:

$(rule\ (e\ is\ nat)\ var\ (e)\ abn\ then\ (e\ is\ nat)::\{transition\})$;

$(transition\ (e\ is\ nat)::\{transition\}\ var\ (e)\ then\ \$f)$,

where

$(\$e\ is\ nat) :: \{transition\}\ \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [if\ [\$e \in \$\$n]\ then\ true\ else\ und]\ \#\ \$s$.

The element ($\$e\ is\ int$) specifies that $\$e$ is an integer. It is defined as follows:

$(rule\ (e\ is\ int)\ var\ (e)\ abn\ then\ (e\ is\ int)::\{transition\})$;

$(transition\ (e\ is\ int)::\{transition\}\ var\ (e)\ then\ \$f)$,

where

$(\$e\ is\ int)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [if\ [\$e \in \$\$in]\ then\ true\ else\ und]\ \#\ \$s$.

The element $\$i$ is defined by the rules

$(rule\ i\ var\ (i)\ abn\ where\ (i\ is\ int)\ then\ i::\{q\})$.

If $\$v1$ and $\$v2$ are values of $\$e1$ and $\$e2$, then the element ($\$e1 + \$e2$) returns [$\$v1 + \$v2$]. It is defined as follows:

$(rule\ (e1 + e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$then\ (e1::\{*\}\ +::\{integer\}\ e2::\{*\})::\{transition\})$;

$(transition\ (i1\ +::\{integer\}\ i2)::\{transition\}\ var\ (i1,\ i2)\ then\ f)$,

where $(\$i1\ +::\{integer\}\ \$i2)::\{transition\}$; $\$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [\$i1 + \$i2]\ \#\ \$s$.

The elements ($\$e1\ \$op\ \$e2$), where $\$op \in \{-,\ *,\ div,\ mod\}$, specifying the integer operations $-$, $*$, div and mod, are defined in the similar way.

If $\$v1$ and $\$v2$ are values of $\$e1$ and $\$e2$, then the element ($\$e1 < \$e2$) specifies that [$\$v1 < \$v2$]. It is defined as follows:

$(rule\ (e1 < e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$then\ (e1::\{*\}\ <::\{integer\}\ e2::\{*\})::\{transition\})$;

$(rule\ (e1::\{*\}\ <::\{integer\}\ e2::\{*\})::\{transition\}\ var\ (e1,\ e2)\ then\ f)$,

where $(\$i1\ <::\{integer\}\ \$i2)::\{transition\}$; $\$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [\$i1 < \$i2]\ \#\ \$s$.

The elements ($\$e1\ \$op\ \$e2$), where $\$op \in \{<=,\ >,\ >=\}$, specifying the integer relations $\leq$, $>$ and $\geq$, are defined in the similar way.

6.3. Boolean values

The executable elements handling boolean values are defined in this section.

The element $true$ is defined by the rule:

$(rule\ true\ abn\ then\ true::\{q\})$.

If $\$v1$ and $\$v2$ are values of $\$e1$ and $\$e2$, then the element $(\$e1\ and\ \$e2)$ specifies the conjunction of $\$v1$ and $\$v2$. It is defined by the rule:

$(rule\ (e1\ and\ e2)\ var\ (e1,\ e2)\ abn\ then\ (if\ e1\ then\ e2\ else\ und))$.

If $\$v1$ and $\$v2$ are values of $\$e1$ and $\$e2$, then the elements $(\$e1\ \$op\ \$e2)$, where $\$op \in \{or,\ =>,\ <=>\}$ specifying the disjunction, implication and equivalence of $\$v1$ and $\$v2$ are defined in the similar way.

If $\$v1$, $\$v2$, ..., $\$v\$n$ are values of $\$e1$, $\$e2$, ..., $\$e\$n$, then the element $(\$e1\ and\ \$e2\ and \ldots\ and\ \$e\$n)$ specifies the conjunction of $\$v1$, $\$v2$, ..., $\$v\$n$. It is defined by the rule

$(rule\ (e1\ and\ e2\ and\ e\_s)\ var\ (e1,\ e2)\ seq\ (e\_s)\ abn\ then\ ((e1\ and\ e2)\ and\ e\_s)$.

If $\$v1$, $\$v2$, ..., $\$v\$n$ are values of $\$e1$, $\$e2$, ..., $\$e\$n$, then the element $(\$e1\ or\ \$e2\ or\ \ldots\ or\ \$e\$n)$ specifying the disjunction of $\$v1$, $\$v2$, ..., $\$v\$n$ is defined in the similar way.

If $\$v$ is a value of $\$e$, then the element $(not\ \$e)$ specifies the negation of $\$v$. It is defined by the rule $(rule\ (not\ e)\ var\ (e)\ abn\ then\ (if\ e\ then\ und\ else\ true))$.

6.4. Conceptual structures

The executable elements handling conceptual structures are defined in this section.

The element $(\$e\ is\ atom)$ specifies that $\$e$ is an atom. It is defined as follows:

$(rule\ (e\ is\ atom)\ var\ (e)\ abn\ then\ (e\ is\ atom)::\{transition\})$;

$(transition\ (e\ is\ atom)::\{transition\}\ var\ (e)\ then\ \$f)$,

where

$(e\ is\ atom)::\{transition\}\ \$e^{*}\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^{*}\ \#\ [if\ [e \in \$\$ato]\ then\ true\ else\ und]\ \#\ \$s$.

The element $(\$e\ is\ compound)$ specifies that $\$e$ is a compound structure. It is defined by the rule

$(rule\ ((e\_s)\ is\ compound)\ seq\ (e\_s)\ abn\ then\ true)$.

The element $(\$e\ is\ (absolutely\ typed))$ specifies that $\$e$ is an absolutely typed structure. It is defined by the rule

$(rule\ (e::\{t\_s\}\ is\ (absolutely\ typed))\ var\ (e)\ seq\ (e\_s)\ abn\ then\ true)$.

The element $(\$e\ is\ (relatively\ typed))$ specifies that $\$e$ is a relatively typed structure. It is defined by the rule

$(rule\ (e:\{t\_s\}\ is\ (relatively\ typed))\ var\ (e)\ seq\ (e\_s)\ abn\ then\ true)$.

The element $(\$e\ is\ empty)$ specifies that $\$e$ is an empty structure. It is defined by the rule

$(rule\ (()\ is\ empty)\ abn\ then\ true)$.

The element $(\$e\ is\ nonempty)$ specifies that $\$e$ is not an empty structure. It is defined by the rule

$(rule\ (e\ is\ empty)\ var(e)\ abn\ then\ (not\ (e\ is\ empty)))$.

The empty structure is defined by the rule

$(rule\ ()\ abn\ then\ ()::\{q\})$.

The element $(\$ccs\ is\ (\$t\ *))$ specifies that the value of $(\$e\ is\ \$t)$ does not equal und for each element $\$e$ of $\$ccs$. It is defined by the rule

$(rule\ ((e\ e\_s)\ is\ (t\ *))\ var\ (e,\ t)\ seq\ (e\_s)\ abn\ then\ ((e\ is\ t)\ and\ ((e\_s)\ is\ (t\ *)))$;

$(rule\ (()\ is\ (t\ *))\ var\ (t)\ abn\ then\ true)$.

If $\$cs$ is a value of $\$e$, then the element $(len\ \$e)$ specifies the length of $\$cs$. It is defined as follows:

$(rule\ (len\ e)\ var\ (e)\ val\ (e)\ abn\ then\ (len\ e :: \{*\}) :: \{transition\})$;

$(transition\ (len\ cs) :: \{transition\}\ var\ (cs)\ then\ \$f)$,

where $(len\ \$cs) :: \{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [len\ \$cs] \# \$s$.

If $\$cs1$ and $\$cs2$ are values of $\$e1$ and $\$e2$, then the element $(\$e1\ = \$e2)$ specifies the equality of $\$cs1$ and $\$cs2$. It is defined as follows:

$(rule\ (e1 = e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ then\ (e1 :: \{*\} = e2 :: \{q\}) :: \{transition\})$;

$(transition\ (cs1\ = cs2) :: \{transition\}\ var\ (cs1,\ cs2)\ then\ \$f)$,

where $(\$cs1\ = \$cs2) :: \{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [\$cs1 = \$cs2] \# \$s$.

If $\$cs1$ and $\$cs2$ are values of $\$e1$ and $\$e2$, then the element $(\$e1\,! = \$22)$ specifies the inequality of the structures $\$cs1$ and $\$cs2$. It is defined by the rule

$(rule\ (e1\,! = e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ then\ (not\ (e1 = e2)))$.

If $\$cs$ is a value of $\$e$, then the conceptual structure access operation $(\$e\ .\ \$mt)$ returns $[\$cs\ .\ \$mt]$. It is defined as follows:

$(rule\ (e\ .\ mt)\ var\ (e,\ t)\ val\ (e)\ abn\ then\ (e :: \{*\}\ .\ mt):\{transition\})$;

$(transition\ (cs\ .\ mt) :: \{transition\}\ var\ (cs,\ mt)\ then\ \$f)$,

where $(\$cs\ .\ \$mt) :: \{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [\$cs\ .\ \$mt] \# \$s$.

If $\$ccs$ and $\$n$ are values of $\$e1$ and $\$e2$, then the conceptual structure access operation $(\$e1\,..\ \$e2)$ returns $[\$ccs\,..\ \$n]$. It is defined as follows:

$(rule\ (e1\,..\ \ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$where\ ((e1 :: \{*\}\ is\ compound)\ and\ (e2 :: \{*\}\ is\ nat)\ and\ (e2 :: \{*\} >\ 0))$

$then\ (e1 :: \{*\}\,..\ e2 :: \{*\}):\{transition\})$;

$(transition\ (cs\,..\ \ n) :: \{transition\}\ var\ (cs,\ n)\ then\ \$f)$, where $(\$cs\,..\,\$n) :: \{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [\$cs\,..\,\$n] \# \$s$.

If $\$cs$ and $\$v$ are values of $\$e$ and $\$e1$, then the conceptual structure update operation $(\$e\,.\,\$mt := \$e1)$ returns $[\$cs\,.\,\$mt \coloneqq \$v]$. It is defined as follows:

$(rule\ (e\,.\,mt1\ \coloneqq e1)\ var\ (e,\ mt1,\ e1)\ val\ (e,\ e1)\ abn)$

$then\ (e::\{*\}\,.\,mt1\ \coloneqq e1::\{*\})::\{transition\}))$;

$(transition\ (cs\,.\,mt \coloneqq\ v)::\{transition\}\ var\ (cs, mt, v)\ abn\ then\ \$f)$,

where $(\$cs\,.\,\$mt := \$v)::\{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [\$cs\,.\,\$mt \coloneqq \$v] \# \$s$.

The conceptual structure update operation $(\$e\,.\,\$mt :=)$ is a shortcut for $(\$e\,.\,\$mt := und)$.

The conceptual structure update operation $(\$e\,.\,\$mt1 := \$e1, \ldots, \$mt\$n := \$e\$n)$, where $\$n > 1$, is defined as follows:

$(rule\ (e\,.\,mt1\ \coloneqq e1\ cs\_s\ )\ var\ (e,\ mt1,\ e1)\ seq\ (cs\_s\ )\ abn\ then\ ((e\,.\,mt1\ \coloneqq e1)\,.\,cs\_s))$.

If $\$ccs$, $\$n$ and $\$v$ are values of \$e1, \$e2 and \$e3, then the conceptual structure update operation $(\$e1\,..\,\$e2 := \$e3)$ returns $[\$ccs\,..\,\$n := \$v]$. It is defined as follows:

$(rule\ (e1\,..\,e2\ \coloneqq e3)\ var\ (e1,\ e2,\ e3)\ val\ (e1,\ e2,\ e3)\ abn$

$where\ ((e1 :: \{*\}\ is\ compound)\ and\ (e2 :: \{*\}\ is\ nat)\ and\ (e2 :: \{*\}\ >\ 0))$

$then\ (e1 :: \{*\}\,..\,e2 :: \{*\}\ \coloneqq e3 :: \{*\}) :: \{transition\})$;

$(transition\ (ccs\,..\,n\ \coloneqq v) :: \{transition\}\ var\ (ccs, n, v)\ abn\ then\ \$f)$,

where $(\$ccs\,..\,\$n := \$v) :: \{transition\}$; $\$e^* \# \$s \hookrightarrow_{[\![\$f]\!]} \$e^* \# [\$ccs\,..\,\$n := \$v] \# \$s$.

If $\$ccs1$ and $\$ccs2$ are values of $\$e1$ and $\$e2$, then the element $(\$e1 + \$e2)$ specifies the concatenation of $\$ccs1$ and $\$ccs2$. It is defined by the rules

$(rule\ (e1 + e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ then\ (e1 :: \{*\} +:: \{q\}\ e2 :: \{*\}))$;

$(rule\ ((cs\_s1) +:: \{q\}\ (cs\_s2))\ seq\ (cs\_1,\ cs\_2)\ then\ (cs\_s1\ cs\_s2) :: \{q\})$.

If $\$e$ and $\$ccs$ are values of $\$e1$ and $\$e2$, then the element $(\$e1\,.+\,\$e2)$ specifies the addition of the element $\$e$ to the head of $\$ccs$. It is defined by the rules

$(rule\ (e1\,.+\,e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ then\ (e1 :: \{*\}\,.+:: \{q\}\ e2 :: \{*\}))$;

$(rule\ (e\,.+:: \{q\}\ \ (cs\_s\ ))\ var(e)\ seq\ (cs\_s)\ then\ (e\ cs\_s) :: \{q\})$.

If $\$e$ and $\$ccs$ are values of $\$e2$ and $\$e1$, then the element $(\$e1 +.\ \ \$e2)$ specifies the addition of the element $\$e$ to the tail of $\$ccs$. It is defined by the rules

$(rule\ (e1 + e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ then\ (e1 :: \{*\} +. :: \{q\}\ e2 :: \{*\}))$;

$(rule\ ((cs\_s\ ) +. :: \{q\}\ e)\ \ var(e)\ seq\ (cs\_s)\ then\ (cs\_s\ e) :: \{q\})$.

If $\$e$ and $\$n$ are values of $\$e1$ and $\$e2$, then the element $(repeat\ \$e1\ \$e2)$ returns $([repeat\ \$e\ \$n])$. It is defined by the rule

$(rule\ (repeat\ e\ n)\ var\ (e,\ n)\ val\ (e,\ n)\ abn\ where\ (n :: \{*\}\ is\ nat)$

$then\ (repeat::\{q\}\ e::\{*\}\ n::\{*\}))$.

The element $(repeat::\{q\}\ \$e1\ \$e2)$ is defined by the rules

$(rule\ (repeat::\{q\}\ e\ 0)\ var\ (e)\ abn\ then\ ())$;

$(rule\ (repeat::\{q\}\ e\ n)\ var\ (e,\ n)\ abn$

$then\ (let\ n1\ be\ (n-1)\ in\ ((repeat::\{q\}\ e\ n1)\ +.\ e::\{q\}))$.

The element $(unbracket\ \$ccs)$ is defined by the rule

$(rule\ (unbracket\ (cs\_s))\ seq\ (cs\_s)\ abn\ then\ cs\_s)$.

6.5. Sets

The element $(\$e\ is\ set)$ specifies that the elements of the compound structure $\$e$ are pairwise distinct is defined as follows:

$(rule\ (e\ is\ set)\ var\ (e)\ abn\ where\ (e\ is\ compound)\ then\ (e\ is\ set)::\{transition\})$;

$(transition\ (e\ is\ set)::\{transition\}\ var\ (e)\ then\ \$f)$,

where

$(e\ is\ set)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if$ [the elements of $\$e$ are pairwise distinct] $then\ true\ else\ und]\ \#\ \$s$.

If $\$e$ and $\$ccs$ are values of $\$e2$ and $\$e1$, then the element $(\$e1\ +.::\{set\}\ \$e2)$ specifies the addition of the element $\$e$ to the set $\$ccs$. It is defined by the rule

$(rule\ (e1\ +.::\{set\}\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$then\ (if\ (e2::\{*\}::\{q\}\ in\ e1::\{*\}::\{q\})\ then\ e1::\{*\}::\{q\}$

$else\ (e2::\{*\}\ +.::\{q\}\ e1::\{*\}))$.

If $\$e$ and $\$ccs$ are values of $\$e2$ and $\$e1$, then the element $(\$e1\ -.::\{set\}\ \$e2)$ specifies the deletion of the element $\$e$ from the set $\$ccs$. It is defined by the rule

$(rule\ (e1\ -.::\{set\}\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ where\ (e1::\{*\}\ is\ set)$

$then\ (e1::\{*\}\ -.::\{set\}\ e2::\{*\})::\{transition\})$;

$(transition\ (ccs\ -.::\{set\}\ e)::\{transition\}\ var\ (ccs,\ e)\ then\ \$f)$,

where $(\$ccs\ -.::\{set\}\ \$e)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ \$ccs1\ \#\ \$s$, $\$ccs1$ is a set, and $[\$ccs1 =_{set} \$ccs\ \$v]$.

If $\$e$ and $\$ccs$ are the values of $\$e1$ and $\$e2$, then the element $(\$e1\ in\ \$e2)$ specifies that $\$e$ is an element of $\$ccs$. It is defined as follows:

$(rule\ (e1\ in::\{set\}\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn\ where\ (e2::\{*\}\ is\ compound)$

$then\ (e1::\{*\}\ in::\{set\}\ e2::\{*\})::\{transition\})$;

$(transition\ (e\ in::\{set\}\ ccs)::\{transition\}\ var\ (e, ccs)\ abn\ then\ \$f)$,

where

$(\$e\ in :: \{set\}\ \$ccs) :: \{transition\};\ \$e^*\ \ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if\ [\$e \in \$ccs]\ then\ true\ else\ und]\ \#\ \$s.$

If $\$ccs1$ and $\$ccs2$ are the values of $\$e1$ and $\$e2$, then the element $(\$e1\ includes{::}\{set\}\ \$e2)$ specifies that $\$ccs1$ includes the elements of $\$ccs1$. It is defined as follows:

$(rule\ (e1\ includes{::}\{set\}\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$where\ ((e1 :: \{*\}\ is\ compound)\ and\ (e2 :: \{*\}\ is\ compound))$

$then\ (e1 :: \{*\}\ includes{::}\{set\}\ e2 :: \{*\}) :: \{transition\});$

$(transition\ (ccs1\ includes{::}\{set\}\ ccs2) :: \{transition\}\ var\ (ccs1, ccs2)\ abn\ then\ \$f),$

where

$(\$ccs1\ includes :: \{set\}\ \$ccs2) :: \{transition\};\ \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if\ [\$ccs1$ includes the elements of $\$ccs2]\ then\ true\ else\ und]\ \#\ \$s.$

If $\$ccs1$ and $\$ccs2$ are the values of $\$e1$ and $\$e2$, then the element $(disjoint{::}\{set\}\ \$e1\ \$e2)$ specifies that $\$ccs1$ and $\$ccs1$ have no common elements. It is defined as follows:

$(rule\ (disjoint{::}\{set\}\ e1\ e2)\ var\ (e1,\ e2)\ val\ (e1,\ e2)\ abn$

$where\ ((e1 :: \{*\}\ is\ compound)\ and\ (e2 :: \{*\}\ is\ compound))$

$then\ (disjoint{::}\{set\}\ e1 :: \{*\}\ e2 :: \{*\}) :: \{transition\});$

$(transition\ (disjoint{::}\{set\}\ ccs1\ ccs2) :: \{transition\}\ var\ (ccs1, ccs2)\ abn\ then\ \$f),$

where

$(disjoint :: \{set\}\ \$ccs1\ \$ccs2) :: \{transition\};\ \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if\ [\$ccs1$ and $\$ccs2$ have no common elements$]\ then\ true\ else\ und]\ \#\ \$s.$

The elements $(\$e1\ in\ \$e2)$, $(\$e1\ includes\ \$e2)$ and $(disjoint\ \$e1\ \$e2)$ are shortcuts for $(\$e1\ in{::}\{set\}\ \$e2)$, $(\$e1\ includes{::}\{set\}\ \$e2)$ and $(disjoint{::}\{set\}\ \$e1\ \$e2)$.

6.6. States

The executable elements handling states are defined in this section.

The state access operation $(current\ state)$ returns the current state is defined by the rule:

$(rule\ (current\ state)\ abn\ then\ cstate{::}\{q\}).$

If $\$cs$ is a value of $\$e$, then the element $((to\ state)\ \$e)$ replaces the current state to $\$cs$. It is defined as follows:

$(rule\ ((to\ state)\ e)\ var\ (e)\ val\ (e)\ then\ ((to\ state)\ e{::}\{*\}){::}\{transition\});$

$(transition\ ((to\ state)\ cs){::}\{transition\}\ var\ (cs)\ then\ \$f),$

where $((to\ state)\ \ \$cs){::}\{transition\};\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$cs.$

The state access operation $(.\ \$mt)$ returns $[.\ \$mt]$. It is defined by the rule

$(rule\ (.\ mt)\ var\ (mt)\ abn\ then\ (cstate :: \{q\}\ .\ mt))$.

If $\$v$ is a value of $\$e$, then the state update operation $(mt := \$e)$ replaces the current state by $[.\ \$mt \coloneqq \$v]$. It is defined by the rule

$(rule\ (mt := \ e)\ var\ (mt,\ e)\ abn\ then\ (to\ state\ (cstate :: \{q\}\ .\ mt\ \coloneqq e)))$.

The state update operation $(.\ mt := \$e)$ is an alias for $(mt := \$e)$. It is defined by the rule

$(rule\ (.\ mt := \ e)\ var\ (mt,\ e)\ abn\ then\ (mt := \ e))$.

The state update operation $(mt :=)$ and $(.\ mt :=)$ are shortcuts for $(mt := und)$ and $(.\ mt := und)$.

The state update operation $(\$mt1 := \$e1,\ \dots,\ \$mt\$n := \$e\$n)$, where $\$n > 1$, is defined by the rule

$(rule\ (mt1\ \coloneqq e1\ cs\_s)\ var\ (mt1,\ e1)\ seq\ (cs\_s)\ abn)\ then\ (mt1\ \coloneqq e1);\ (cs\_s))$.

The state update operation $(.\ \$mt1 := \$e1,\ \dots,\ \$mt\$n := \$e\$n)$ is an alias for $(\$mt1 := \$e1,\ \dots,\ \$mt\$n := \$e\$n)$. It is defined by the rule

$(rule\ (.\ mt1\ \coloneqq e1\ cs\_s)\ var\ (mt1,\ e1)\ seq\ (cs\_s)\ abn)\ then\ (mt1\ \coloneqq e1\ cs\_s))$.

6.7. Statements

The executable elements called statements are defined in this section. They are similar to statements in programming languages.

The element $skip$ does nothing. It is defined as follows:

$(rule\ skip\ abn\ then\ skip :: \{transition\})$;

$(transition\ skip :: \{transition\}\ then\ \$f)$,

where $skip :: \{transition\}$; $\$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} e^*\ \#\ \$s$.

The element $\$e$ of the form $(seq\ e^*)$ is called a sequential composition. It is defined by the rule

$(rule\ (seq\ e\_s)\ var\ (e\_s)\ seq\ (e\_s)\ then\ e\_s)$.

The elements of $\$e^*$ are called elements in $[\![\$e]\!]$, and $\$e^*$ is called a body in $[\![\$e]\!]$. The element $\$e$ executes its elements sequentially from left to right.

The element $\$e$ of the form $(if\ \$co\ then\ \$e^*1\ else\ \$e^*2)$ is called a conditional element. It is defined as follows:

$(rule\ (if\ co\ then\ e\_s1\ else\ e\_s2)\ var\ (co)\ seq\ (e\_s1,\ e\_s2)\ val\ (co)\ abn$

$then\ (if\ co :: \{*\}\ then\ e\_s1\ else\ e\_s2) :: \{transition\})$;

$(transition\ (if\ v\ then\ e\_s1\ else\ e\_s2) :: \{transition\}\ var\ (v)\ seq\ (e\_s1,\ e\_s2)\ then\ \$f)$,

where

$(if\ \$v\ then\ \$e^*1\ else\ \$e^*2) :: \{transition\};\ \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]}$

$[if\ [\$v\ \neq und]\ then\ \$e^*1\ else\ \$e^*2]\ \$e^*\ \#\ \$s.$

The objects $\$co$, $\$e^*1$ and $\$e^*2$ are called a condition, then-branch and else-branch in $[\![\$e]\!]$. The element $(if\ \$con\ then\ \$e^*)$ is a shortcut for $(if\ \$con\ then\ \$e^*\ else\ skip)$.

The conditional element $(if :: \{exc\}\ \$con\ then\ \$e^*1\ else\ \$e^*2)$ is defined by the rule

$(rule\ (if :: \{exc\}\ con\ then\ e\_s1\ else\ e\_s2)\ var\ (con)\ seq\ (e\_s1,\ e\_s2)\ val\ (con)$

$exc\ (con,\ con :: \{*\})\ abn\ then\ (if\ con :: \{q\}\ then\ e\_s1\ else\ e\_s2)).$

The element $(if :: \{exc\}\ \$con\ then\ \$e^*)$ is a shortcut for $(if :: \{exc\}\ \$con\ then\ \$e^*\ else\ skip)$.

The conditional element

$(if\ \$se1\ \$co1\ then\ \$e^*1\ elseif\ \$se2\ \$co2\ then\ \$e^*2 \ldots\ elseif\ \$se\$n\ \$co\$n\ then\ \$e^*\$n\ else\ \$e^*)$,

where $\$se\$n1 \in \{[es],\ exc\}$ for each $1 \leq \$n1 \leq \$n$, is defined by the rules

$(rule\ (if\ co\ then\ e\_s1\ elseif\ e\_s2)\ var\ (co)\ seq\ (e\_s1,\ e\_s2)\ abn$

$then\ (if\ co\ then\ e\_s1\ else\ (if\ e\_s2)));$

$(rule\ (if\ co\ then\ e\_s1\ elseif :: \{exc\}\ e\_s2)\ var\ (co)\ seq\ (e\_s1,\ e\_s2)\ abn$

$then\ (if\ co\ then\ e\_s1\ else\ (if :: \{exc\}\ e\_s2)));$

$(rule\ (if :: \{exc\}\ co\ then\ e\_s1\ elseif\ e\_s2)\ var\ (co)\ seq\ (e\_s1,\ e\_s2)\ abn$

$then\ (if :: \{exc\}\ co\ then\ e\_s1\ else\ (if\ e\_s2)));$

$(rule\ (if :: \{exc\}\ co\ then\ e\_s1\ elseif :: \{exc\}\ e\_s2)\ var\ (co)\ seq\ (e\_s1,\ e\_s2)\ abn$

$then\ (if :: \{exc\}\ co\ then\ e\_s1\ else\ (if :: \{exc\}\ e\_s2))).$

The element $\$e$ of the form $(let\ \$va\ be\ \$e^*1\ in\ \$e^*2)$ is defined as follows:

$(rule\ (let\ va\ be\ e\_s1\ in\ e\_s2)\ var\ (va)\ seq\ (e\_s1, e\_s2)\ abn$

$then\ e\_s1;\ (let\ va\ be\ current\ value\ in\ e\_s2) :: \{transition\});$

$(transition\ (let\ va\ be\ current\ value\ in\ e\_s2) :: \{transition\}\ var\ (va)\ seq\ (e\_s2)\ then\ \$f)$,

where

$(let\ \$va\ be\ current\ value\ in\ \$e^*2) :: \{transition\}; \$e^*\ \#\ \$v\ \#\ \$s \hookrightarrow_{[\![\$f]\!]}$

$[sub\ (\$va{:}\ \$v)\ \$e^*2]; \$e^*\ \#\ \$s.$

The elements $\$va$, $\$e^*1$ and $\$e^*2$ are called a variable, value specifier and body in $[\![\$e]\!]$.

The element $\$e$ of the form $(let :: \{und\}\ \$va\ be\ \$e^*1\ in\ \$e^*2)$ is defined by the rules

$(rule\ (let :: \{und\}\ va\ be\ e\_s1\ in\ e\_s2)\ var\ (va)\ seq\ (e\_s1, e\_s2)\ abn$

$then\ e\_s1;\ (let :: \{und\}\ va\ be\ current\ value\ in\ e\_s2));$

$(rule\ (let :: \{und\}\ va\ be\ current\ value\ in\ e\_s2)\ var\ (va)\ seq\ (e\_s2)\ und$

$then\ (let\ va\ be\ current\ value\ in\ e\_s2){:}\{transition\}).$

The element $\$e$ of the form $(let :: \{abn\}\ \$va\ be\ \$e^*1\ in\ \$e^*2)$ is defined by the rules

$(rule\ (let :: \{abn\}\ va\ be\ e\_s1\ in\ e\_s2)\ var\ (va)\ seq\ (e\_s1, e\_s2)\ abn$
$then\ e\_s1;\ (let :: \{abn\}\ va\ be\ current\ value\ in\ e\_s2));$
$(rule\ (let :: \{abn\}\ va\ be\ current\ value\ in\ e\_s2)\ var\ (va)\ seq\ (e\_s2)\ abn$
$then\ (let\ va\ be\ current\ value\ in\ e\_s2) :: \{transition\}).$

The element $\$e$ of the form $(let :: \{exc\}\ \$va\ be\ \$e^*1\ in\ \$e^*2)$ is defined by the rules

$(rule\ (let :: \{exc\}\ va\ be\ e\_s1\ in\ e\_s2)\ var\ (va)\ seq\ (e\_s1, e\_s2)\ abn$
$then\ e\_s1;\ (let :: \{exc\}\ va\ be\ current\ value\ in\ e\_s2));$
$(rule\ (let :: \{exc\}\ va\ be\ current\ value\ in\ e\_s2)\ var\ (va)\ seq\ (e\_s2)\ exc$
$then\ (let\ va\ be\ current\ value\ in\ e\_s2) :: \{transition\}).$

The element $\$e$ of the form $(let :: \{seq\}\ \$va^*\ be\ \$e^*1\ in\ \$e^*2)$, where $[len\ va^*] = [len\ e^*1]$, is defined by the rules

$(rule\ (let :: \{seq\}\ va, va\_s\ be\ e1,\ e\_s1\ in\ e\_s2)\ var\ (va,\ e1)\ seq\ (va\_s, e\_s1, e\_s2)\ abn$
$then\ (let\ va\ be\ e1\ in\ (let :: \{seq\}\ va\_s\ be\ e\_s1\ in\ e\_s2)));$
$(rule\ (let :: \{seq\}\ be\ in\ e\_s2)\ seq\ (e\_s2)\ abn\ then\ e\_s2).$

The elements $\$va^*$, $\$e^*1$ and $\$e^*2$ are called a variable specification, value specification and body in $[\![\$e]\!]$. The elements of $\$va^*$ and $\$e^*1$ are called variables and value specifiers in $[\![\$e]\!]$.

The element $\$e$ of the form $(let :: \{seq, und\}\ \$va^*\ be\ \$e^*1\ in\ \$e^*2)$, where $[len\ va^*] = [len\ e^*1]$, is defined by the rules

$(rule\ (let :: \{seq, und\}\ va, va\_s\ be\ e1,\ e\_s1\ in\ e\_s2)\ var\ (va,\ e1)\ seq\ (va\_s, e\_s1, e\_s2)$
$abn\ then\ (let :: \{und\}\ va\ be\ e1\ in\ (let :: \{seq, und\}\ va\_s\ be\ e\_s1\ in\ e\_s2)));$
$(rule\ (let :: \{seq, und\}\ be\ in\ e\_s2)\ seq\ (e\_s2)\ abn\ then\ e\_s2).$

The element $\$e$ of the form $(let :: \{seq, abn\}\ \$va^*\ be\ \$e^*1\ in\ \$e^*2)$, where $[len\ va^*] = [len\ e^*1]$, is defined by the rules

$(rule\ (let :: \{seq, abn\}\ va, va\_s\ be\ e1,\ e\_s1\ in\ e\_s2)\ var\ (va,\ e1)\ seq\ (va\_s, e\_s1, e\_s2)$
$abn\ then\ (let :: \{abn\}\ va\ be\ e1\ in\ (let :: \{seq, abn\}\ va\_s\ be\ e\_s1\ in\ e\_s2)));$
$(rule\ (let :: \{seq, abn\}\ be\ in\ e\_s2)\ seq\ (e\_s2)\ abn\ then\ e\_s2).$

The element $\$e$ of the form $(let: \{seq, exc\}\ \$va^*\ be\ \$e^*1\ in\ \$e^*2)$, where $[len\ va^*] = [len\ e^*1]$, is defined by the rules

$(rule\ (let :: \{seq, exc\}\ va, va\_s\ be\ e1,\ e\_s1\ in\ e\_s2)\ var\ (va,\ e1)\ seq\ (va\_s, e\_s1, e\_s2)\ abn$
$then\ (let :: \{exc\}\ va\ be\ e1\ in\ (let :: \{seq, exc\}\ va\_s\ be\ e\_s1\ in\ e\_s2)));$
$(rule\ (let :: \{seq, exc\}\ be\ in\ e\_s2)\ seq\ (e\_s2)\ abn\ then\ e\_s2).$

The element $\$e$ of the form $(while\ \$con\ do\ \$e^*1)$ is called a while statement. It is defined by the rule

$(if\ (while\ con\ do\ e\_s)\ var\ (con)\ seq\ (e\_s)\ abn$
$then\ (if\ con\ then\ e\_s;\ (while\ con\ do\ e\_s))).$

The objects $\$con$ and $\$e^*1$ are called a condition and body in $[\![\$e]\!]$.

The element $\$e$ of the form $(foreach\ \$va\ in\ \$e1\ do\ \$e^*1)$ is called a foreach statement. It is defined by the rule

$(rule\ (foreach\ va\ in\ e1\ do\ e\_s)\ var\ (va,\ e1)\ seq\ (e\_s)\ val\ (e1)\ abn$
$then\ (foreach::\{q\}\ va\ in\ e1 :: \{*\}\ do\ e\_s)).$

The objects $\$va$, $\$e1$ and $\$e^*1$ are called an iteration variable, iteration structure specifier and body in $[\![\$e]\!]$. If $(\$v^*)$ is a value of $\$e1$, then the element $\$e$ executes sequentially $\$e^*1$ for values of $\$va$ from the structure $(\$v^*)$.

The element $(foreach::\{q\}\ va\ in\ (\$v^*)\ do\ \$e^*)$ is defined by the rules

$(rule\ (foreach::\{q\}\ va\ in\ (v\ v\_s)\ do\ e\_s)\ var\ (va,\ v)\ seq\ (v\_s,\ e\_s)\ abn$
$then\ (let\ va\ be\ v :: \{q\}\ in\ e\_s);\ (foreach::\{q\}\ va\ in\ (v\_s)\ do\ e\_s));$
$(rule\ (foreach::\{q\}\ va\ in\ ()\ do\ e\_s)\ var\ (va)\ seq\ (e\_s)\ abn\ then).$

6.8. Countable concepts

The executable elements handling countable concepts are defined in this section.

A normal element $\$e$ is a countable concept in $[\![\$s]\!]$ if $[\$s\ .\ ((countable\ concept)\ \$e)] \in \$n > 0$. Thus, the parametric attribute $((countable\ concept)\ \$e)$ defines countable concepts. Let $\$\$cc$ be a set of countable concepts. A number $\$n$ is an order in $[\![\$cc,\ \$s]\!]$ if $\$n = [\$s\ .\ ((countable\ concept)\ \$cc)]$. Let $\$\$cco$ be a set of orders of countable concepts. An element $\$n::\{\$cc\}$ is called an instance in $[\![\$cc]\!]$. An element $\$n::\{\$cc\}$ is an instance in $[\![\$cc,\ \$s]\!]$ if $1 \leq \$n \leq \$cco[\![\$cc]\!]$.

The element $(\$e\ is\ (countable\ concept))$ specifies that $\$e$ is a countable concept. it is defined by the rule

$(rule\ (e\ is\ (countable\ concept))\ var\ (e)\ abn\ then\ ((.\ \{((countable\ concept)\ e)\}) > 0)).$

The element $(\$e\ is\ \$cc)$ specifies that $\$e$ is an instance of $\$cc$. It is defined by the rule

$(rule\ (n::\{cc1\}\ is\ cc2)\ var\ (n,\ cc1,\ cc2)\ abn\ where\ (cc2\ is\ (countable\ concept))$
$then\ ((cc1::\{q\} = cc2::\{q\})\ and\ (0 < n)\ and\ (n <= (.\ \{((countable\ concept)\ cc2)\})))).$

The element $((new\ instance)\ \$cc)$ generates a new instance of the countable concept $\$cc$ and adds this concept if it was not. It is defined by the rule

$(rule\ ((new\ instance)\ cc)\ var\ (cc)\ abn$
$then\ (let\ n\ be\ (.\ \{(countable\ concept)\ cc\})\ in$

$(if\ (n\ >\ 0)\ then\ (let\ n1\ be\ (n+1)\ in$

$(\{((countable\ concept)\ cc)\}\ \coloneqq\ n1);\ n1::\{cc\}::\{q\})$

$else\ (\{((countable\ concept)\ cc)\}\ \coloneqq\ 1);\ n1::\{cc\}::\{q\}).$

6.9. Rules

The executable elements handling rules are defined in this section.

The element $(\$e\ is\ rule)$ specifies that $\$e$ is a rule. It is defined as follows:

$(rule\ (e\ is\ rule)\ var\ (e)\ abn\ then\ (e\ is\ rule)::\{transition\});$

$(transition\ (e\ is\ rule)::\{transition\}\ var\ (e)\ then\ \$f),$

where

$(\$e\ is\ rule)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ [if\ [e \in \$\$r]\ then\ true\ else\ und]\ \#\ \$s.$

The element $(\$e\ is\ (extended\ rule))$ specifies that $\$e$ is an extended rule. It is defined as follows:

$(rule\ (e\ is\ (extended\ rule))\ var\ (e)\ abn\ then\ (e\ is\ (extended\ rule))::\{transition\});$

$(transition\ (e\ is\ (extended\ rule))::\{transition\}\ var\ (e)\ then\ \$f),$

where

$\big(\$e\ is\ (extended\ rule)\big) :: \{transition\}\ \$e^*\ \$s \hookrightarrow_{[\![\$f]\!]}$

$\$e^*\ \#\ [if\ [e \in \$\$er]\ then\ true\ else\ und]\ \#\ \$s.$

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

The element $(\$e\ is\ name)$ specifies that $\$e$ is a name. It is defined by the rule

$(rule\ (e\ is\ name)\ var\ (e)\ abn\ then\ (e\ is\ normal)).$

The element $\$r{:}\ \$na$ adds the rule $\$r$ with the name $\$n$ into $[.\ \{rules\}]$. It is defined by the rule

$(rule\ e::\{na\}\ var\ (e,\ na)\ abn\ where\ ((e\ is\ rule)\ and\ (na\ is\ name))$

$then\ (\{rules\} := ((.\{rules\})\ .\ \{na\} := e::\{q\}))).$

The element $\$er{:}\ \$na$ adds the rule $\$r$ with the name $\$n$ into $[.\ \{rules\}]$, where $\$er$ is a shortcut for $\$r$. It is defined as follows:

$(rule\ e::\{na\}\ var\ (e,\ na)\ abn\ where\ ((e\ is\ (extended\ rule))\ and\ (na\ is\ name))$

$then\ (\{rules\} := ((.\{rules\})\ .\ \{na\}\ \coloneqq\ (rule\ e)))).$

The element $(rule\ \$er)$ returns $\$r$, where $\$r$ is a shortcut for $\$er$. It is defined as follows:

$(transition\ (rule\ er)::\{transition\}\ var\ (er)\ then\ \$f),$

where $(rule\ \$er)::\{transition\}\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]} \$e^*\ \#\ \$r\ \#\ \$s$, where $\$r$ is a shortcut for $\$er$.

6.10. The pattern matching

The executable elements handling the pattern matching are defined in this section.

The conditional pattern matching element $\$e$ of the form

$(if\ \$e1\ matches\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ then\ \$e^*1\ else\ \$e^*2)$,

where $(\$p\ (\$va^*)\ (\$sv^*))$ is a pattern specification, executes $\$e^*1$ if $\$e1$ matches $\$p$ and executes $\$e^*2$, otherwise. It is defined as follows:

$(rule\ (if\ e1\ matches\ p\ var\ (va\_s)\ seq\ (sv\_s)\ then\ e\_s1\ else\ e\_s2)\ var\ (e1,\ p)$

$seq\ (va\_s,\ sv\_s,\ e\_s1,\ e\_s2)\ abn\ where\ (disjoint\ (va\_s)::\{q\}\ (sv\_s)::\{q\})$

$then\ (if\ e1\ matches\ p\ var\ (va\_s)\ seq\ (sv\_s)\ then\ e\_s1\ else\ e\_s2)::\{transition\})$;

$(transition\ (if\ e1\ matches\ p\ var\ (va\_s)\ seq\ (sv\_s)\ then\ e\_s1\ else\ e\_s2)::\{transition\}$

$var\ (e1,\ p)\ seq\ (va\_s,\ sv\_s,\ e\_s1,\ e\_s2)\ then\ \$f)$,

where

$(if\ \$e1\ matches\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ then\ \$e^*1\ else\ e^*2)::\{transition\};\ \$e^*\ \#\ \$s \hookrightarrow_{[\![\$f]\!]}$

$[if\ [\$e1\ is\ an\ instance\ in\ [\![(\$p\ (\$va^*)\ (\$sv^*)),\ \$mt,\ \$su]\!]\ for\ some\ \$su]$

$then\ [sub\ \$su\ \cup\ (cstate{:}\ \$s,\ cvalue{:}\ \$v[\![\$s]\!])\ \$e^*1]$

$else\ [sub\ (cstate{:}\ \$s,\ cvalue{:}\ \$v[\![\$s]\!])\ \$e^*2]];\ \$e^*\ \#\ \$s.$

Thus, the semantics of the conditional pattern matching elements combines the semantics of the conditional element with the semantics of transition rules. The elements $\$e1$, $\$p$, $\$va^*$, $\$sv^*$, $\$e^*1$ and $\$e^*2$ are called a matched structure, pattern, state variable specification, sequence variable specification, $then$-branch and $else$-branch in $[\![\$e]\!]$. The elements of $\$va^*$ and $\$sv^*$ are called state and sequence variables in $[\![\$e]\!]$.

Let $\{\$eva^*\} \subseteq \{\$va^*\}$, the elements of the sequence $\$eva^*$ are pairwise disjoint, $\$set\ =\ \{\$eva :: \{*\} \mid \$eva \in \$eva^*\}$, $\{\$va^*1\} \cup \{\$va^*2\} \cup \{\$va^*3\} \subseteq \{\$va^*\} \cup \$set$, the elements of the sequence $\$va^*1\ \$va^*2\ \$va^*3$ are pairwise disjoint, and $\$se \in \{[se],\ und,\ exc,\ abn\}$.

The semantics of the extension

$(if\ \$e1\ matches\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ abn\ (\$va^*1)\ und\ (\$va^*2)$

$exc\ (\$va^*3)\ \$se\ where\ \$co\ then\ \$e^*1\ else\ \$e^*2)$

of the conditional pattern matching element is similar to the semantics of extended transition rules.

The objects $var\ (\$va^*)$, $seq\ (\$sv^*)$, $val\ (\$eva^*)$, $abn\ (\$va^*1)$, $und\ (\$va^*2)$, $exc\ (\$va^*3)$ and $where\ \$co$ in conditional pattern matching elements can be omitted. The omitted objects correspond to $var\ (\,)$, $seq\ (\,)$, $val\ (\,)$, $abn\ (\,)$, $und\ (\,)$, $exc\ (\,)$ and $where\ true$.

The pattern matching element

$(\$e1\ matches\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ abn\ (\$va^*1)\ und\ (\$va^*2)$

$exc\ (\$va^*3)\ \$se\ where\ \$co\ then\ \$e^*1\ else\ \$e^*2)$

is a shortcut for

$(if\ \$e1\ matches\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)\ abn\ (\$va^*1)\ und\ (\$va^*2)$
$exc\ (\$va^*3)\ \$se\ where\ \$co\ then\ true\ else\ und)$.

The selection element $\$e$ of the form $(select\ \$va\ from\ \$e1\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*))$, where $(\$p\ (\$va^*)\ (\$sv^*))$ is a pattern specification, and $\$va \in \$va^*$, or $\$va \in \$sv^*$, selects the values of the variable $\$va$ such that an element of $\$e1$ matches the pattern $\$p$. It is defined by the rule

$(rule\ (select\ va\ from\ of\ e1\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s))$
$var\ (va, e1, p)\ seq\ (va\_s, sv\_s, t\_s)\ abn$
$where\ ((disjoint\ (va\_s) :: \{q\}\ (sv\_s) :: \{q\})\ and$
$((va::\{q\}\ in\ (va\_s)::\{q\})\ or\ (va::\{q\}\ in\ (sv\_s)::\{q\})))$
$then\ (select::\{check\}\ va\ from\ e1\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s)))$.

The elements $\$va$, $\$e1$, $\$p$, $\$va^*$ and $\$sv^*$ are called a selection variable, matched structure, pattern, state variable specification, sequence variable specification, $then$-branch and $else$-branch in $[\![\$e]\!]$. The elements of $\$va^*$ and $\$sv^*$ are called state and sequence variables in $[\![\$e]\!]$.

The element $(select::\{check\}\ \$va\ from\ \$e1\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*))$ is defined by the rules

$(rule\ (select::\{check\}\ va\ from\ e1::\{t\_s\}\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s))$
$var\ (va, e1,\ p)\ seq\ (va\_s,\ sv\_s,\ t\_s)\ abn$
$then\ (select::\{check\}\ va\ from\ (e1::\{t\_s\})\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s)))$;
$(rule\ (select::\{check\}\ va\ from\ e1{:}\{t\_s\}\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s))$
$var\ (va,\ e1,\ p)\ seq\ (va\_s,\ sv\_s,\ t\_s)\ abn$
$then\ (select::\{check\}\ va\ from\ (e1{:}\{t\_s\})\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s)))$;
$(rule\ (select::\{check\}\ va\ from\ ()\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s))$
$var\ (va, p)\ seq\ (e\_s, va\_s, sv\_s)\ abn\ then\ ())$;
$(rule\ (select::\{check\}\ va\ from\ (e1\ e\_s)\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s))$
$var\ (va, e1, p)\ seq\ (e\_s, va\_s, sv\_s)\ abn\ where\ (va::\{q\}\ in\ (va\_s)::\{q\})$
$then\ (if\ e1\ matches\ p\ var\ (va\_s)\ seq\ (sv\_s)$
$then\ (va\_s::\{q\}\ .+\ (select::\{check\}\ va\ from\ (e\_s)\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s)))$
$else\ (select::\{check\}\ va\ from\ (e\_s)\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s))))$;
$(rule\ (select::\{check\}\ va\ from\ (e1\ e\_s)\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s))$
$var\ (va, e1, p)\ seq\ (e\_s, va\_s, sv\_s)\ abn\ where\ (va::\{q\}\ in\ (sv\_s)::\{q\})$
$then\ (if\ e1\ matches\ p\ var\ (va\_s)\ seq\ (sv\_s)$
$then\ ((va\_s)::\{q\}\ .+\ (select::\{check\}\ va\ from\ (e\_s)\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s)))$

$else\ (select::\{check\}\ va\ from\ (e\_s)\ wrt\ p\ var\ (va\_s)\ seq\ (sv\_s)))).$

The semantics of the extension

$(select\ \$va\ from\ \$e1\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)$

$abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co)$

of the selection element is similar to the semantics of the extension of the conditional pattern matching element.

The semantics of the extension

$(select::\{seq\}\ \$va1^+\ from\ \$e1\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)$

$abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co)$

of the selection element is similar to the semantics of the above selection element extension except that the resulting sequence consists of the compound elements of the length $[\$va1^+]$. Each of these elements contains the values of the selection variables in the order of their occurences in $\$va1^+$.

The selection elements

$(select\ \$va\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)$

$abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co)$

and

$(select::\{seq\}\ \$va1^+\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)$

$abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co)$

are shortcuts for

$(let\ \$s\ be\ (current\ state)\ in\ (select\ \$va\ from\ \$s\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)\ val\ (\$eva^*)$

$abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co))$

and

$(let\ \$s\ be\ (current\ state)\ in\ (select::\{seq\}\ \ \$va1^+\ from\ \$s\ wrt\ \$p\ var\ (\$va^*)\ seq\ (\$sv^*)$

$val\ (\$eva^*)\ abn\ (\$va^*1)\ und\ (\$va^*2)\ exc\ (\$va^*3)\ \$se\ where\ \$co)).$

7. Examples of conceptual operational semantics of programming languages

An operational semantics of executable elements of $\$\$c[\![\$l]\!]$ in CTSL[o] is called a conceptual operational semantics of $\$l$. Thus, the conceptual operational semantics of $\$l$ is defined in terms of the conceptual model of $\$l$ in CTSL[o].

The conceptual operational semantics for the family of model programming languages (MPLs) is defined in this section. These languages has been described and their conceptual models has been defined in [1].

7.1. MPL1: types, typed variables and basic statements

The MPL1 language [1] 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.

The element ($\$c\ is\ type$) specifies types in MPL1. It is defined by the rules

$(rule\ (t\ is\ type)\ var\ (t)\ abn\ then\ (t::\{q\} = int::\{q\}));$

$(rule\ (t\ is\ type)\ var\ (t)\ abn\ then\ (t::\{q\} = nat::\{q\})).$

The element ($subtype\ \$t1\ \$t2$) checks that $\$t1$ is a subtype of $\$t2$. It is defined by the rule

$(rule\ (subtype\ t1\ t2)\ var\ (t1,\ t2)\ abn$

$then\ ((t1::\{q\} = nat::\{q\})\ and\ (t2::\{q\} = int::\{q\}))).$

The element ($\$na\ is\ variable$) specifies variables. It is defined by the rule

$(rule\ (va\ is\ variable)\ var\ (va)\ abn\ where\ (va\ is\ name)\ then\ (.\ \{(variable\ va)\})).$

The program is defined by the rule

$(rule\ (program\ n\ c\_s)\ var\ (n)\ seq\ (c\_s)\ abn\ where\ (n\ is\ name)$

$then\ ((collect\ body\ membes)\ c\_s)\ c\_s).$

Let $\$\$m$ be a set of elements called body members.

The element $((collect\ body\ members)\ \$c^*)$ collects information about members of the body $\$c^*$. It is defined by the rules

$(rule\ ((collect\ body\ members)\ (var\ va\ t)\ c\_s)\ var\ (va,\ t)\ seq\ (c\_s)\ abn$

$where\ ((va\ is\ name)\ and\ (not\ (va\ is\ variable))\ and\ (t\ is\ type))$

$then\ (\{(type\ va)\} \coloneqq t :: \{q\});\ (\{(variable\ va)\} \coloneqq true);$

$((collect\ body\ members)\ c\_s));$

$(rule\ ((collect\ body\ members)\ (var\ c1\ c2)\ c\_s)\ var\ (c1,\ c2)\ seq\ (c\_s)\ abn\ then\ und);$

$(rule\ ((collect\ body\ members)\ c\ c\_s)\ var\ (c)\ seq\ (c\_s)\ abn$

$then\ ((collect\ body\ members)\ c\_s));$

$(rule\ ((collect\ variables))\ then).$

Thus, the body members in MPL1 are variables.

The variable declaration is defined by the rule

$(rule\ (var\ c\_s)\ seq\ (c\_s)\ abn\ then).$

The execution of the variable declaration does not collect information about the declared variable, since the corresponding actions are performed by the element $((collect\ body\ members)\ \$c^*)$.

The variable access is defined by the rule

$(rule\ va\ var\ (va)\ abn\ where\ (va\ is\ variable)\ then\ (.\ \{(value\ va)\}).$

The element $(type\ \$c)$ returns the type of the variable or the constant $\$c$. It is defined by the rules

$(rule\ (type\ va)\ var\ (va)\ abn\ (va)\ abn\ where\ (va\ is\ variable)\ then\ (.\ \{(type\ va)\})$;

$(rule\ (type\ n)\ var\ (n)\ abn\ (n)\ abn\ where\ (n\ is\ nat)\ then\ nat::\{q\})$;

$(rule\ (type\ i)\ var\ (i)\ abn\ (i)\ abn\ where\ (i\ is\ int)\ then\ int::\{q\})$.

The variable assignment is defined by the rule

$(rule\ (va\ \backslash{:=}\ c)\ var\ (va,\ c)\ val\ (va)\ exc\ (c,\ c::\{*\})\ abn$

$where\ ((va\ is\ variable)\ and$

$(let::\{seq\}\ t1,\ t2\ be\ (type\ va),\ (type\ c::\{*\})\ in\ (subtype\ t2\ t1)))$

$then\ (\{(value\ va)\} := c::\{*\}::\{q\}))$.

The block statement is defined by the rule

$(rule\ (block\ c\_s)\ seq\ (c\_s)\ abn\ then\ c\_s)$.

The if statement is defined by the rule

$(rule\ (\backslash if\ c\ then\ c\_s1\ else\ c\_s2)\ var\ (c)\ seq\ (c\_s1,\ c\_s2)\ abn\ (c)\ abn$

$then\ (if::\{exc\}\ c\ then\ (block\ c\_s1)\ else\ (block\ c\_s2)))$;

The while statement is defined by the rule

$(rule\ (\backslash while\ c\ do\ c\_s1)\ var\ (c)\ seq\ (c\_s1)\ abn\ (c)\ abn$

$then\ (while::\{exc\}\ c\ do\ (block\ c\_s1)))$.

Thus, then- and else- branches of the if statement and the body of the while statement behaves as blocks.

7.2. MPL2: variable scopes

The MPL2 language [1] 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 element $(scope)$ returns the current scope. It is defined by the rule

$(rule\ (scope)\ abn\ then\ (.\ \{(current\ scope)\}))$.

The same name $\$na$ can refer to different program objects. For example, $\$va$ refers to the variables with the name $\$va$ of the scopes from 0 to $[.\ \{(current\ scope)\}]$. To distinguish these program objects, they are versioned. The pair $(\$na,\ \$ve)$, where $\$ve$ is a version, refers to the only one program object (with the version $\$ve$). In the case of variables, the version coincides with the variable scope.

The element $(version\ \$va)$ returns the correct version of $\$va$ in the current context of program execution. It is defined by the rule

$(rule\ (version\ va)\ var\ (va)\ abn\ (va)\ abn\ where\ (va\ is\ name)$
$then\ (let\ sc\ be\ (scope)\ in\ (version\ sc\ w)))$.

The element $(version\ \$va\ \$sc)$ is defined by the rule

$(rule\ (version\ va\ sc)\ var\ (va,\ sc)\ abn\ (va,\ sc)\ abn$
$then\ (if\ (.\ \{(variable\ va\ sc)\})\ then\ sc::\{q\}\ else\ (if\ (sc\ =\ 0)\ then\ und$
$else\ (let\ sc1\ be\ (sc-1)\ in\ (version\ va\ sc1)))))$.

The element $(\$na\ is\ variable)$ is defined by the rule

$(rule\ (va\ is\ variable)\ var\ (va)\ abn\ then\ (version\ va))$.

The element $((collect\ body\ members)\ \$c^*)$ is defined by the rule

$(rule\ ((collect\ body\ members)\ c\_s)\ seq\ (c\_s)\ abn$
$then\ ((collect\ body\ members\ 1)\ ()\ c\_s)$.

The element $((collect\ body\ members)::\{1\}\ (\$va^*)\ \$c^*)$ is defined by the rules

$(rule\ ((collect\ body\ members)::\{1\}\ (va\_s)\ (var\ va\ t)\ c\_s)\ var\ (va,\ t)$
$seq\ (va_s,\ c_s)\ abn\ (va,\ t)\ abn\ where\ ((va\ is\ name)\ and\ (t\ is\ type))$
$then\ (let\ sc\ be\ (scope)\ in$
$(if\ (.\ \{(variable\ va\ sc)\})\ then\ und$
$else\ (\{(type\ va\ sc)\}\ :=t::\{q\});\ (\{(variable\ va\ sc)\}\ :=true);$
$((collect\ body\ members)::\{1\}\ (va\_s\ va)\ c\_s))))$;

$(rule\ ((collect\ body\ members)::\{1\}\ (va\_s)\ (var\ c1\ c2)\ c\_s)\ var\ (c1,\ c2)$
$seq\ (va\_s,\ c\_s)\ abn\ then\ und)$;

$(rule\ ((collect\ body\ members)::\{1\}\ (va\_s)\ c\ c\_s)\ var\ (c)\ seq\ (va\_s,\ c\_s)\ abn$
$then\ ((collect\ body\ members)::\{1\}\ (va\_s)\ c\_s))$;

$(rule\ ((collect\ body\ members)::\{1\}\ (va\_s))\ seq\ (va\_s)\ abn\ then\ (va\_s)::\{q\})$.

Thus, it returns the set of variables declared in the body $\$c^*$.

The variable access is defined by the rule

$(rule\ va\ var\ (va)\ abn\ (va)\ abn$
$then\ (let::\{und\}\ sc\ be\ (version\ va)\ in\ (.\ \{(value\ va\ sc)\})))$.

In the case when $\$c$ is a variable, the rule for the element $(type\ \$c)$ is replaced by the rule

$(rule\ (type\ va)\ var\ (va)\ abn\ (va)\ abn$
$then\ (let:\{und\}\ sc\ be\ (version\ va)\ in\ (.\ \{(type\ va\ sc)\})))$.

The element $(scope++)$ increases the value of the current scope by 1. It is defined by the rule

$(rule\ (scope++)\ abn\ \mathrm{then}\ (\{(current\ scope)\} := ((.\ \{(current\ scope)\}) + 1)))$.

The element $(scope--)$ decreases the value of the current scope by 1. It is defined by the rule

$(rule\ (scope--)\ abn\ then\ (\{(current\ scope)\} := ((.\ \{(current\ scope)\}) - 1)))$.

The variable assignment is defined by the rule

$(rule\ (va \backslash := c)\ var\ (va,\ c)\ val\ (c)\ abn\ (va)\ exc\ (c, c{::}\{*\})\ abn$

$then\ (let{::}\{und\}\ sc\ be\ (version\ va)$

$in\ (if\ (let{::}\{und,\ seq\}\ \ t1,\ t2\ be\ (type\ va\ sc), (type\ c{::}\{*\})\ in\ (subtype\ t2\ t1))$

$then\ (((value\ va\ sc)) := c{::}\{*\}{::}\{q\})\ else\ und)))$.

The block statement is defined by the rule

$(rule\ (block\ c\_s)\ seq\ (c\_s)\ abn$

$then\ (enter\ block);\ (let\ v\_s\ be\ ((collect\ body\ members)\ c\_s)$

$in\ x\ (catch{::}\{und\}\ v\ ((exit\ block)\ v\_s);\ v{::}\{q\})))$.

The element $(enter\ block)$ specifies the actions executed when the current state enters the block. It is defined by the rule

$(rule\ (enter\ block)\ abn\ then\ (scope++))$.

The element $((exit\ block)\ (\$va^*))$ specifies the actions executed when the current state exits the block. It is defined by the rule

$(rule\ ((exit\ block)\ (va\_s))\ seq\ (va\_s)\ abn\ then\ ((delete\ variables)\ va\_s);\ (scope--))$.

The element $((delete\ variables)\ \$va^*)$ deletes the local variables $\$va^*$ with the current scope. It is defined by the rules

$(rule\ ((delete\ variables)\ va\_s)\ seq\ (va\_s)\ abn$

$then\ (let\ sc\ be\ (scope)\ in\ ((delete\ variables){::}\{1\}\ sc\ va\_s)))$.

The element $((delete\ variables){::}\{1\}\ \$sc\ \$va^*)$ is defined by the rules

$(rule\ ((delete\ variables){::}\{1\}\ sc\ va\ va\_s)\ var\ (sc,\ va)\ seq\ (va\_s)\ abn\ (sc,\ y)\ abn$

$then\ (\{(variable\ va\ sc)\} \coloneqq);\ (\{(type\ va\ sc)\} \coloneqq);\ (\{(value\ va\ sc)\} \coloneqq);$

$((delete\ variables){::}\{1\}\ sc\ va\_s))$;

$(rule\ ((delete\ variables){::}\{1\}\ sc)\ var\ (sc)\ abn\ (sc)\ abn\ then)$.

7.3. MPL3: functions

The MPL3 language [1] is an extension of MPL2 that adds the functions feature: declarations and calls of functions, and the return statement. For simplicity, function overloading is prohibited.

The element $(call\ level)$ returns the current call level. It is defined by the rule

$(rule\ (call\ level)\ abn\ then\ (.\ \{(current\ call\ level)\}))$.

The element ($\$c\ is\ function$) specifies functions. It is defined by the rule

$(rule\ (f\ is\ function)\ var\ (f)\ abn\ where\ (f\ is\ name)\ then\ (.\ \{(function\ f)\}))$.

In the case when the first element of the body $\$c^*$ is a variable declaration, the rule for the element $((collect\ body\ members)::\{1\}\ (va^*)\ c^*)$ is replaced by the rule

```
$(rule\ ((collect\ body\ members)\ ::\ \{1\}\ (va\_s)\ (var\ va\ t)\ c\_s)\ var\ (va,\ t)\ seq\ (va\_s,\ c\_s)$
 $abn\ (va,\ t)\ abn\ where\ ((va\ is\ name)\ and\ (t\ is\ type))$
 $then\ (let::\{seq\}\ sc,\ cl\ be\ (scope),\ (if\ (sc = 0)\ then\ 0\ else\ (call\ level))$
   $in\ (if\ (.\{(variable\ va\ sc\ cl)\})\ then\ und$
      $else\ (\{(type\ va\ sc)\} := t::\{q\});\ (\{(variable\ va\ sc)\} := true);$
        $((collect\ body\ members)::\{1\}\ (va\_s\ va)\ c\_s))))$.
```

The element $((collect\ body\ members)::\{1\}\ (va^*)\ c^*)$ is also redefined by the extra rules

```
$(rule\ ((collect\ body\ members)\ ::\ \{1\}\ (function\ f\ (tna\_s)\ t\ c\_s1)\ c\_s)\ var\ (f,\ t)$
 $seq\ (tna\_s,\ c\_s1,\ c\_s)\ abn\ (f,\ t)\ abn$
 $where\ ((f\ is\ name)\ and\ \big(not\ (f\ is\ function)\big)\ and\ (t\ is\ type))$
 $then\ ((collect\ member\ arguments)\ f\ tna\_s);\ (\{((return\ type)\ f)\} \coloneqq t::\{q\});$
  $(\{(body\ f)\} \coloneqq\ (c\_s1)\ ::\ \{q\});\ (\{(function\ f)\}\ \coloneqq\ true);$
  $((collect\ body\ members)::\{1\}\ c\_s));$
$(rule\ ((collect\ body\ members)::\{1\}\ (function\ c\_s))\ seq\ (c\_s\ )\ abn\ then\ und)$.
```

Thus, body members in MPL3 are variables and functions.

The element $((collect\ member\ arguments)\ \$m\ \$tna^*)$ collects information about the typed arguments $\$tna^*$ of the body member $\$m$. It is defined by the rule

```
$(rule\ ((collect\ member\ arguments)\ m\ tna\_s)\ var\ (m)\ seq\ (tna\_s)\ abn$
 $then\ ((collect\ member\ arguments\ 1)\ m\ 0\ tna\_s))$.
```

The element $((collect\ member\ arguments\ 1)\ \ \$m\ \$n\ \$tna^*)$ is defined by the rules

```
$(rule\ ((collect\ member\ arguments\ 1)\ m\ n\ na\ t\ tna\_s)\ var\ (m,\ n,\ na,\ t)\ \ seq\ (tna\_s)$
 $abn\ where\ ((m\ is\ name)\ and\ (n\ is\ nat)\ and\ (na\ is\ name)\ and\ (t\ is\ type))$
 $then\ (let\ n1\ be\ (n + 1)\ \ in\ (\{((argument\ type)\ m\ n1)\} \coloneqq t::\{q\}));$
   $(\{(argument\ m\ n1)\}\ \coloneqq na::\{q\};\ ((collect\ member\ arguments\ 1)\ m\ n1\ tna\_s))));$
$(rule\ ((collect\ member\ arguments\ 1)\ m\ n)\ var\ (m,\ n)\ \ abn$
 $where\ ((m\ is\ name)\ \ and\ (n\ is\ nat))\ then\ (\{(arity\ m)\} \coloneqq n))$.
```

Thus, it collects information about function arguments.

The function declaration is defined by the rule:

$(rule\ (function\ c\_s)\ seq\ (c\_s)\ abn\ then)$.

The execution of the function declaration does not collect information about the declared function, since the corresponding actions are performed by the element $((collect\ body\ members)\ \$c^*)$.

The return statement is defined by the rule

$(rule\ (return\ c)\ var\ (c)\ val\ (c)\ exc\ (c, c::\{*\})\ abn$

$then\ (let::\{seq\}\ \ t1,\ t2\ be\ (.\ \{(current\ return\ type)\}),\ (type\ c::\{*\})$

$in\ (if\ \ (subtype\ t2\ t1)\ then\ (return:\{type\},\ c::\{*\}:\{value\})::\{exc\}\ else\ und))).$

The function call is defined by the rule

$(rule\ (call\ f\ a\_s)\ var\ (f)\ seq\ (a\_s)\ abn\ (f)\ abn$

$where\ ((f\ is\ function)\ and\ ((len\ a\_s::\{q\}) = (.\{(arity\ f)\})))$

$then\ (let::\{und, seq\}\ av\_s,\ b,\ cs,\ crt$

$be\ ((argument\ values)\ a\_s),\ (body\ f\ av\_s),\ (scope),\ (.\ \{(current\ return\ type)\})$

$in\ (call\ level++);\ (\{(current\ scope)\}\ \coloneqq\ 0);\ b;$

$(catch::\{und\}\ v$

$(if\ \ (v\ is\ (not\ admissible\ function\ body\ value))\ then\ und);$

$(\{(current\ return\ type)\}\ \coloneqq\ crt::\{q\});\ \ (call\ level--);\ (\{(current\ scope)\}\ \coloneqq\ cs);$

$(if\ v\ matches\ (return:\{type\},\ v1:\{value\})::\{exc\}\ var\ (v1)\ then\ ((to\ value)\ v1::\{q\})$

$else\ ((to\ value)\ v::\{q\}))))).$

The element $(\$v\ is\ (\text{not admissible function body value}))$ specifies values that are not admissible when a function body exits. It is defined by the rule

$\big(rule\ v\ is\ (not\ admissible\ function\ body\ value)\big)\ var\ (v)\ abn$

$then\ (not\ (v\ is\ exception))).$

Thus, the values that are not exceptions are not admissible in MPL3 when a function body exits.

The element $((argument\ values)\ \$a^*)$ returns the values of the arguments $\$a^*$. It is defined by the rules

$(rule\ ((argument\ values)\ a,\ a\_s)\ var\ (a)\ seq\ (a\_s)\ abn\ (a)\ abn$

$then\ (a\ .+\ ((argument\ values)\ a\_s)));$

$(rule\ ((argument\ values))\ then\ ()).$

The element $(body\ \$f\ (\$v^*))$ creates the block with the body of the function $\$f$ followed the declarations of the local variables corresponding to the arguments of $\$f$ and the assignment statements assigning the values $\$v^*$ to these variables.

$(rule\ (body\ f\ (v\_s))\ \ var\ (f)\ \ seq\ (v\_s)\ abn\ (f)\ abn$

$then\ (block::\{q\}\ .+\ (((create\ local\ variables)\ \ f\ 0\ v\_s)\ \ +(.\ \{(block\ f)\}))));$

The element $((create\ local\ variables)\ \$f\ \$n\ \$v^*)$ creates the declarations of the local variables corresponding to the arguments of $\$f$ and the assignment statements assigning the values $\$v^*$ to these variables. It is defined by the rules

$(rule\ ((create\ local\ variables)\ f\ n\ v\ v\_s)\ var\ (f,\ n,\ v)\ seq\ (v\_s)\ abn\ (f,\ n,\ v)\ abn$
$then\ (let :: \{seq\}\ n1,\ a,\ t$
$be\ (n+1),\ (.\ \{(argument\ f\ n1)\}),\ (.\ \{((argument\ type)\ f\ n1)\})\ in$
$((var\ a\ t)\ .+((a \backslash{:=}\ \ v::\{q\})\ .+((create\ local\ variables)\ \ f\ n1\ v\_s)))));$

$(rule\ ((create\ local\ variables)\ f\ n)\ var\ (f,\ n)\ abn\ (f,\ n)\ abn\ then\ ()).$

The element $(call\ level++)$ increases the value of the current call level by 1. It is defined by the rule

$(rule\ (call\ level\ \ ++)\ abn$
$then\ (\{(current\ call\ level)\} :=\ \ ((.\ \{(current\ call\ level)\})+1))).$

The element $(call\ level\ \ --)$ decreases the value of the current call level by 1. It is defined by the rule

$(rule\ (call\ level\ \ --)\ abn$
$then\ (\{(current\ call\ level)\} :=\ \ ((.\ \{(current\ call\ level)\})-1))).$

The element $(version\ \$na)$ is defined by the rules

$(rule\ (version\ va)\ var\ (va)\ abn\ (va)\ abn\ where\ (va\ is\ name)$
$then\ (let::\{seq\}\ sc,\ cl\ be\ (current\ scope),\ (current\ call\ level)\ in\ (version\ va\ sc\ cl)));$

The element $(version\ \$na\ \$sc\ \$cl)$ is defined by the rules

$(rule\ (version\ va\ y\ z)\ var\ (va,\ sc,\ cl)\ abn\ (va,\ sc,\ cl)\ abn$
$then\ (if\ (.\ \{(variable\ va\ sc\ cl)\})\ then\ sc$
$else\ (if\ (sc\ =\ 0)\ then\ und$
$else\ (let\ sc1\ be\ (sc-1)\ in\ (version\ va\ sc1\ cl))))).$

The variable access is defined by the rule

$(rule\ va\ var\ (va)\ abn\ (va)\ abn$
$then\ (let::\{und,\ seq\}\ sc,\ cl\ be\ (version\ va),\ (if\ (sc::\{q\}=\ 0)\ then\ 0\ else\ (call\ level))$
$in\ (.\ \{(value\ va\ sc\ cl)\}))).$

In the case when $\$c$ is a variable, the rule for the element $(type\ \$c)$ is replaced by the rule

$(rule\ (type\ va)\ var\ (va)\ abn\ (va)\ abn$
$then\ (let :: \{und,\ seq\}\ sc, cl$
$be\ (version\ va),\ (if\ (sc::\{q\}=\ 0)\ then\ 0\ else\ (call\ level))$
$in\ (.\ \{(type\ va\ sc\ cl)\}))).$

The variable assignment is defined by the rule

$(rule\ (va \setminus := c)\ var\ (va,\ e)\ val\ (c)\ abn\ (v)\ exc\ (c,\ c::\{*\})\ abn$

$then\ (let::\{und\}\ sc,\ cl,\ t1,\ t2$

$be\ (variant\ va), (if\ (sc::\{q\} =\ 0)\ then\ 0\ else\ (current\ call\ level)),$

$(.\{(type\ va\ sc\ cl)\}), (type\ c::\{*\})$

$in\ (if\ (subtype\ t2\ t1)\ then\ (\{(value\ x\ sc\ cl)\}\ \coloneqq c::\{*\}::\{q\})\ else\ und))).$

The element $((delete\ variables)\ \$va^*)$ is defined by the rules

$(rule\ ((delete\ variables)\ va\_s)\ seq\ (va\_s)\ abn$

$then\ (let::\{seq\}\ sc,\ cl\ be\ (scope),\ (if\ (sc::\{q\} =\ 0)\ then\ 0\ else\ (call\ level))$

$in\ ((delete\ variables)::\{1\}\ sc\ cl\ va\_s))).$

The element $((delete\ variables)::\{1\}\ \$va^*\ \$sc\ \$cl)$ is defined by the rules

$(rule\ \big((delete\ variables)::\{1\}\ sc\ cl\ va\ va\_s\big)\ var\ (sc,\ cl,\ va)\ seq\ (va\_s)\ abn\ (sc,\ cl,\ va)$

$abn\ then\ (\{(value\ va\ sc\ cl)\} \coloneqq);\ (\{(type\ va\ sc\ cl)\} \coloneqq);$

$(\{(variable\ va\ sc\ cl)\} \coloneqq);\ ((delete\ variables)::\{1\}\ sc\ cl\ va\_s));$

$(rule\ ((delete\ variables)::\{1\}\ sc\ cl)\ var\ (sc,\ cl)\ abn\ (sc,\ cl)\ abn\ then).$

7.4. MPL4: procedures

The MPL4 language [1] is an extension of MPL3 that adds the procedures feature: declarations and calls of procedures, and the exit statement. For simplicity, procedure overloading is prohibited. The sets of function names and procedure names are disjoint.

The element $(\$c\ is\ procedure)$ specifies procedures. It is defined by the rule

$(rule\ (pr\ is\ procedure)\ var\ (pr)\ abn\ where\ (pr\ is\ name)\ then\ (.\ \{(procedure\ pr)\})).$

The element $((collect\ body\ members)::\{1\}\ \ (va^*)\ c^*)$ is redefined by the extra rules

$(rule\ ((collect\ body\ members)::\{1\}\ (procedure\ pr\ (tna\_s)\ c\_s1)\ c\_s)$

$var\ (pr)\ seq\ (tna\_s,\ c\_s1,\ c\_s)\ abn\ (pr)\ abn$

$where\ ((pr\ is\ name)\ and\ (not\ (pr\ is\ procedure)))$

$then\ ((collect\ member\ arguments)\ pr\ tna\_s);$

$(\{(body\ pr)\} \coloneqq\ (c_{s1})\ ::\ \{q\});\ (\{(procedure\ pr)\}\ \coloneqq\ true);$

$((collect\ body\ members)::\{1\}\ c\_s));$

$(rule\ ((collect\ body\ members)::\{1\}\ (procedure\ c\_s))\ seq\ (c\_s\ )\ abn\ then\ und).$

Thus, body members in MPL4 are variables, functions and procedures.

The element $((collect\ member\ arguments)\ \$m\ \$ta^*)$ is extended to procedures. Its definition is not changed.

The procedure declaration is defined by the rule:

$(rule\ (procedure\ c\_s)\ seq\ (c\_s)\ abn\ then)$.

The execution of the procedure declaration does not collect information about the declared procedure, since the corresponding actions are performed by the element $((collect\ body\ members)\ \$c^*)$.

The exit statement is defined by the rule

$(rule\ exit\ abn\ then\ (exit{:}\{type\}){:}{:}\{exc\})$.

The element $(\$v\ is\ (not\ admissible\ function\ body\ value))$ is redefined by the extra rule

$(rule\ ((exit{:}\{type\})::\{exc\}\ is\ (not\ admissible\ function\ body\ value))\ var\ (v)\ abn$
$then\ true)$.

Thus, exceptions initiated by $exit$ statements are not admissible in MPL4 when a function body exits.

The procedure call is defined by the rule

```
$(rule\ (call\ pr\ a\_s)\ var\ (pr)\ seq\ (a\_s)\ abn\ (pr)\ abn$
 $where\ ((pr\ is\ procedure)\ and\ ((len\ a\_s{:}{:}\{q\}) = (.\{(arity\ pr)\})))$
 $then\ (let{:}{:}\{und, seq\}\ av\_s,\ b,\ cs$
  $be\ ((argument\ values)\ \ a\_s),\ (body\ pr\ av\_s),\ (scope)$
  $in\ (call\ level++);\ (\{(current\ scope)\} := 0);\ b;$
   $(catch{:}{:}\{und\}\ v$
     $(if\ (v\ is\ (not\ admissible\ procedure\ body\ value))\ then\ und);$
     $(call\ level--);\ (\{(current\ scope)\} := cs);$
     $(if\ v\ matches\ (exit{:}\{type\}){:}{:}\{exc\}\ then\ true\ else\ ((to\ value)\ v{:}{:}\{q\})))))$.
```

The element $(\$v\ is$ (not admissible procedure $body\ value))$ specifies exceptions that are not admissible when a procedure call exits. It is defined by the rule

$(rule\ ((return{:}\{type\},\ v{:}\{value\})::\{exc\}\ is\ (not\ admissible\ procedure\ body\ value))$
$var\ (v)\ abn\ then\ true)$.

Thus, exceptions initiated by $return$ statements are not admissible in MPL4 when a procedure body exits.

The elements $(body\ \$f\ (\$v^*))$ and $((create\ local\ variables)\ \$f\ \$n\ \$v^*)$ are extended to procedures. Their definitions are not changed.

7.5. MPL5: pointers

The MPL5 language [1] 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.

The element ($\$c\ is\ (pointer\ value)$) specifies pointers in MPL5. It is defined by the rule

$(rule\ (n::\{pointer\}\ is\ (pointer\ value))\ var\ (n)\ abn\ then\ (y\ is\ nat))$.

The element ($po\ is\ pointer$) specifies pointers in states. It is defined by the rule

$(rule\ (po\ is\ pointer)\ var\ (po)\ abn\ where\ (po\ is\ (pointer\ value))\ then\ (.\ \{(pointer\ po)\}))$.

The element ($\$po\ is\ (pointer\ \$t)$) specifies pointers with the given content type. It is defined by the rule

$(rule\ (po\ is\ (pointer\ t))\ var\ (po,\ t)\ abn\ where\ ((po\ is\ pointer)\ and\ (t\ is\ type))$

$then\ ((.\ \{((content\ type)\ po)\}) = t::\{q\}))$.

The element ($\$t\ is\ (pointer\ type)$) specifies pointer types. It is defined by the rule

$(rule\ ((pointer\ t)\ \ is\ (pointer\ type))\ var\ (t)\ abn\ then\ (t\ is\ type))$.

The element ($c\ is\ type$) is redefined by the extra rule

$(rule\ (c\ is\ type)\ abn\ then\ (c\ is\ (pointer\ type)))$.

The element $\$\$po$ is defined by the rule

$(rule\ x::\{pointer\}\ var\ (x)\ abn\ where\ (x\ is\ nat)\ then\ x::\{pointer\}::\{q\})$.

The element ($(content\ type)\ \$po$) returns the content type of $\$po$. It is defined by the rule

$(rule\ ((content\ type)\ po)\ var\ (po)\ abn\ where\ (po\ is\ pointer)$

$then\ (.\ \{((content\ type)\ po)\}))$.

The element ($type\ \$po$) is defined by the rule

$(rule\ (type\ po)\ var\ (po)\ abn\ where\ (po\ is\ pointer)$

$then\ (let\ t\ be\ (.\ \{((content\ type)\ po)\})\ in\ (pointer\ t)::\{q\}))$.

The pointer content access operation is defined by the rule

$(rule\ (*\ \ c)\ var\ (c)\ val\ (c)\ abn\ (c,\ c :: \{*\})\ abn$

$where\ (c::\{*\}\ is\ pointer)\ then\ (.\ \{(content\ c::\{*\})\}))$.

The pointer content assignment statement is defined by the rule

$(rule\ (*\ \ c1\ \coloneqq\ c2)\ var\ (c1,\ c2)\ val\ (c1,\ c2)\ abn\ (c1,\ c2,\ c1 :: \{*\},\ c2 :: \{*\})\ abn$

$where\ (c1::\{*\}\ is\ pointer)$

$then\ (let::\{und,\ seq\}\ t1,\ t2\ be\ (.\ \{((content\ type)\ c1::\{*\})\}),\ (type\ c2::\{*\})$

$in\ (if\ \ (subtype\ t2\ t1)\ then\ (\{(content\ c1::\{*\})\} := \ c2::\{*\}::\{q\})\ else\ und)))$.

The pointer addition operation is defined by the rule

$(rule\ (new\ (pointer\ t))\ var\ (t)\ abn\ (t)\ abn\ where\ (y\ is\ type)$

$then\ (let\ po\ be\ ((new\ instance)\ pointer)$

$in\ (\{((content\ type)\ po)\} := t::\{q\});\ (\{(pointer\ po)\} := true);\ po))$.

The pointer deletion operation is defined by the rule

```
$(rule\ (delete\ c)\ var\ (c)\ val\ (c)\ abn\ (c,\ c::\{*\})\ abn\ where\ (c::\{*\}\ is\ pointer)$
 $then\ (\{(content\ c::\{*\})\}:=);\ (\{((content\ type)\ c::\{*\})\}:=);\ (\{(pointer\ c::\{*\})\}:=))$.
```

The element $(version\ \$na\ \$sc\ \$cl)$ is defined by the rules

```
$(rule\ (version\ va\ y\ z)\ var\ (va,\ sc,\ cl)\ abn\ (va,\ sc,\ cl)\ abn$
 $then\ (if\ (.\ \{(pointer\ va\ sc\ cl)\})\ then\ sc$
  $else\ (if\ (sc\ =\ 0)\ then\ und$
   $else\ (let\ sc1\ be\ (sc-1)\ in\ (version\ va\ sc1\ cl)))))$.
```

The variable address access operation is defined by the rule

```
$(rule\ (\&\ va)\ var\ (va)\ abn\ (va)\ abn$
 $then\ (let\ ::\{und,\ seq\}\ sc,\ cl\ be\ (version\ va),$
    $(if\ (sc::\{q\}=0)\ then\ 0\ else\ (current\ call\ level))$
  $in\ (if\ sc::\{q\}\ then\ (let\ po\ be\ (.\ \{(pointer\ va\ sc\ cl)\})\ in\ (.\ \{(content\ po)\}))$
    $else\ und))$.
```

The variable access is defined by the rule

```
$(rule\ va\ var\ (va)\ abn\ (va)\ abn\ then\ (let::\{und\}\ po\ be\ (\&\ va)\ in\ (.\ \{(content\ po)\})))$.
```

The element (type va) is defined by the rule

```
$(rule\ va\ var\ (va)\ abn\ (va)\ abn$
 $then\ (let::\{und\}\ po\ be\ (\&\ va)\ in\ (.\ \{((content\ type)\ po)\})))$.
```

In the case when the first element of the body $\$c^*$ is a variable declaration, the rule for the element $((collect\ body\ members)::\{1\}\ (va^*)\ c^*)$ is replaced by the rule

```
$(rule\ ((collect\ body\ members)\ ::\ \{1\}\ (va\_s)\ (var\ va\ t)\ c\_s)\ var\ (va,\ t)\ seq\ (va\_s,\ c\_s)$
 $abn\ (va,\ t)\ abn\ where\ ((va\ is\ name)\ and\ (t\ is\ type))$
 $then\ (let::\{seq\}\ sc,\ cl\ be\ (scope),\ (if\ (sc::\{q\}=\ 0)\ then\ 0\ else\ (call\ level))$
   $in\ (if\ (.\{(pointer\ va\ sc\ cl)\})\ then\ und$
    $else\ (let\ po\ be\ ((new\ instance)\ pointer)$
      $in\ (\{((content\ type)\ po)\}:=\ t::\{q\});\ (\{(pointer\ po)\}:=\ true);$
        $((collect\ body\ members)::\{1\}\ (va\_s\ va)\ c\_s))))$.
```

The variable assignment is defined by the rule

```
$(rule\ (va\ \backslash:=\ c)\ var\ (va,\ c)\ abn\ (va,\ c,\ c::\{*\})\ abn$
 $then\ (let::\{und,\ seq\}\ sc,\ cl,\ po,\ t1,\ t2$
   $be\ (version\ va),\ (if\ (sc::\{q\}\ =\ 0)\ then\ 0\ else\ (call\ level)),\ (.\ \{(pointer\ va\ sc\ cl)\}),$
     $(.\ \{((content\ type)\ po)\}),\ (type\ c::\{*\})$
   $in\ (if\ (subtype\ t2\ t1)\ then\ (\{(content\ po)\}:=\ c::\{*\}::\{q\})\ else\ und)))$.
```

The element $((delete\ variables)::\{1\}\ \$va^*\ \$sc\ \$cl)$ is defined by the rules

$(rule\ \big((delete\ variables)::\{1\}\ sc\ cl\ va\ va\_s\big)\ var\ (sc,\ cl,\ va)\ seq\ (va\_s)\ abn\ (sc,\ cl,\ va)$

$abn\ \text{then}\ (\{(pointer\ va\ sc\ cl)\} \coloneqq);\ ((delete\ variables)::\{1\}\ sc\ cl\ va\_s));$

$(rule\ ((delete\ variables)::\{1\}\ sc\ cl)\ var\ (sc,\ cl)\ abn\ (sc,\ cl)\ abn\ \text{then}).$

7.6. MPL6: jump statements

The MPL6 language [1] is an extension of MPL5 that adds the jump statements feature: break statement, continue statement, goto statement and labelled statement.

The element $(e\ is\ label)$ specifies labels. It is defined by the rule

$(rule\ (e\ is\ label)\ var\ (e)\ abn\ \text{then}\ (x\ is\ normal)).$

The break statement is defined by the rule

$(rule\ break\ abn\ \text{then}\ (break{:}\{type\})::\{exc\}).$

The continue statement is defined by the rule

$(rule\ continue\ abn\ \text{then}\ (continue{:}\{type\})::\{exc\}).$

The goto statement is defined by the rule

$(rule\ (goto\ l)\ var\ (l)\ abn\ where\ (l\ is\ label)\ then\ (goto{:}\{type\},\ l{:}\{label\})::\{exc\}).$

The element $(\$v\ is\ (not\ admissible\ function\ body\ value))$ is redefined by the extra rules

$(rule\ ((break{:}\{type\})::\{exc\}\ is\ (not\ admissible\ function\ body\ value))\ abn\ \text{then}\ true);$

$(rule\ ((continue{:}\{type\}) :: \{exc\}\ is\ (not\ admissible\ function\ body\ value))\ abn$

$then\ true);$

$(rule\ ((goto{:}\{type\},\ l{:}\{label\}) :: \{exc\}\ is\ (not\ admissible\ function\ body\ value))$

$var\ (l)\ abn\ \text{then}\ (l\ is\ label)).$

The element $(\$v\ is\ (not\ admissible\ procedure\ body\ value))$ is redefined by the extra rules

$(rule\ ((break{:}\{type\})::\{exc\}\ is\ (not\ admissible\ procedure\ body\ value))\ abn\ \text{then}\ true);$

$(rule\ ((continue{:}\{type\}) :: \{exc\}\ is\ (not\ admissible\ procedure\ body\ value))\ abn$

$then\ true);$

$(rule\ ((goto{:}\{type\},\ l{:}\{label\}) :: \{exc\}\ is\ (not\ admissible\ procedure\ body\ value))$

$var\ (l)\ abn\ \text{then}\ (l\ is\ label)).$

Thus, exceptions initiated by $break$, $continue$ and $goto$ statements are not admissible in MPL6 when a function or procedure body exits.

The label statement is defined by the rule

$(rule\ (label\ l)\ var\ (l)\ abn\ where\ (l\ is\ label)$

$then\ (catch\ v\ (if\ v\ matches\ (goto{:}\{type\},\ l1{:}\{label\})::\{exc\}\ var\ (l1)$

$where\ ((l1\ is\ label)\ and\ (l1::\{q\} = l::\{q\}))\ then\ else\ v::\{q\})))$.

The block statement is defined by the rule

$(rule\ (block\ c\_s)\ seq\ (c\_s)\ abn$

$then\ (enter\ block);$

$(let::\{und,\ seq\}\ va\_s,\ l\_s\ be\ ((collect\ body\ members)\ c\_s),\ ((collect\ labels)\ c\_s)$

$in\ c\_s;\ ((catch\ goto)\ (l\_s)\ c\_s);$

$(catch::\{und\}\ v\ ((exit\ block)\ va\_s);\ v::\{q\}))).$

The element $((collect\ labels)\ \$c^*)$ collects labels from the label statements occurring in $\$c^*$. It is defined by the rules

$(rule\ ((collect\ labels)\ (label\ l)\ c\_s)\ var\ (l)\ seq\ (c\_s)\ abn$

$where\ (l\ is\ label)\ then\ (l::\{q\} .+ ((collect\ labels)\ c\_s)));$

$(rule\ ((collect\ labels)\ c\ c\_s)\ var\ (c)\ seq\ (c\_s)\ then\ ((collect\ labels)\ c\_s));$

$(rule\ ((collect\ labels))\ then\ ()).$

The element $((catch\ goto)\ (\$l^*)\ \$c^*)$ catches the exceptions initiated by $goto$ statements when the current block exits. It is defined by the rule

$(rule\ ((catch\ goto)\ (l\_s)\ c\_s)\ seq\ (l\_s,\ c\_s)\ abn$

$then\ (catch\ v$

$(if\ v\ matches\ (goto:\{type\},\ l:\{label\})::\{exc\}\ var\ (l)\ where\ (l::\{q\}\ in::\{set\}\ (l\_s)::\{q\})$

$then\ v::\{q\};\ c\_s;\ ((catch\ goto)\ (l\_s)\ c\_s)\ else\ v::\{q\}))).$

The while statement is defined by the rules

$(rule\ (\backslash while\ con\ do\ c\_s)\ var\ (con)\ seq\ (c\_s)\ exc\ (con)\ abn$

$then\ (while::\{exc\}\ con\ do\ (block\ c\_s;\ ((delete\ exception)\ continue)));$

$((delete\ exception)\ break)).$

7.7. MPL7: dynamic arrays

The MPL7 language [1] 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.

The element $(\$t\ is\ (dynamic\ array\ type))$ specifies dynamic array types. It is defined by the rule

$(rule\ ((array\ t)\ \ is\ (dynamic\ array\ type))\ var\ (t)\ abn\ then\ (t\ is\ type)).$

The element $(\$t\ is\ (array\ type))$ specifies array types. It is defined by the rule

$(rule\ (t\ is\ (array\ type))\ var\ (t)\ abn\ then\ (t\ is\ (dynamic\ array\ type))).$

The element $(c\ is\ type)$ is redefined by the extra rule

$(rule\ (c\ is\ type)\ abn\ then\ (c\ is\ (array\ type)))$.

The element $(\$e\ is\ (dynamic\ array))$ specifies dynamic arrays. It is defined by the rule

$(rule\ ((v{:}\{content\},\ t{:}\{type\}) :: \{(dynamic\ array)\}\ is\ (dynamic\ array))\ var\ (v,\ t)$
$abn\ where\ (t\ is\ type)\ then\ (v\ is\ ((array\ content)\ t)))$.

The element $(\$e\ is\ array)$ specifies arrays. It is defined by the rule

$(rule\ (e\ is\ array)\ var\ (e)\ abn\ then\ (e\ is\ (dynamic\ array)))$.

The element $(\$v\ is\ ((array\ content)\ \$t))$ is defined by the rules

$(rule\ (()\ is\ ((array\ content)\ t))\ var\ (t)\ abn\ then\ true)$;

$(rule\ ((v\ v\_s)\ is\ ((array\ content)\ t))\ var\ (v,t,v\_s)\ abn\ where\ (v\ is\ t)$
$then\ ((v\_s)\ is\ ((array\ content)\ t)))$.

The element $\$ar$ is defined by the rule

$(rule\ ar\ var\ (ar)\ abn\ where\ (ar\ is\ array)\ then\ ar::\{q\})$.

The element $((element\ type)\ \$ar)$ returns the element type of $\$ar$. It is defined by the rule

$(rule\ ((element\ type)\ ar)\ var\ (ar)\ abn\ where\ (ar\ is\ array)\ \ then\ (ar\ .\ \{type\}))$.

The element $(\$dar\ is\ (array\ \$t))$ specifies dynamic arrays with the given element type. It is defined by the rule

$(rule\ (dar\ is\ (array\ t))\ var\ (dar,\ t)\ abn$
$where\ ((dar\ is\ (dynamic\ array))\ and\ (t\ is\ type))$
$then\ (((element\ type)\ dar) = t::\{q\}))$.

The element $(type\ \$dar)$ is defined by the rule

$(rule\ (type\ dar)\ var\ (dar)\ abn\ where\ (ar\ is\ (dynamic\ array))$
$then\ (let\ t\ be\ ((element\ type)\ dar)\ in\ (array\ t)::\{q\}))$.

The array content access operation is defined by the rule

$(rule\ (content\ c)\ var\ (c)\ val\ (c)\ abn\ (c,\ c :: \{*\})\ abn\ where\ (c :: \{*\}\ is\ array)$
$then\ (c::\{*\}\ .\ \{content\}))$.

The len operation for arrays is defined by the rule

$(rule\ (len\ c)\ var\ (c)\ val\ (c)\ abn\ (c,\ c :: \{*\})\ abn\ where\ (c :: \{*\}\ is\ array)$
$then\ (content\ c::\{*\}::\{q\}))$.

The array element access operation is defined by the rule

$(rule\ (c1\ [\ c2\ ])\ var\ (c1,\ c2)\ val\ (c1,\ c2)\ abn\ (c1,\ c1::\{*\},\ c2,\ c2::\{*\})\ abn$
$where\ ((c1::\{*\}\ is\ array)\ and\ (c2::\{*\}\ is\ nat))\ then\ ((content\ c1::\{*\})\ ..\ c2::\{*\}))$.

The array element assignment statement is defined by the rule

$(rule\ (c1\ [\ c2\ ]\ \coloneqq c3)\ var\ (c1,c2,c3)\ val\ (c1,c2,c3)$

$abn\ (c1, c2, c3, c1::\{*\}, c2::\{*\}, c3::\{*\})\ abn$

$where\ ((c1::\{*\}\ is\ (dynamic\ array))\ and\ (c2::\{*\}\ is\ nat))$

$then\ (let::\{und,\ seq\}\ t1,\ t2\ be\ ((element\ type)\ c1::\{*\}),\ (type\ c3::\{*\})$

$in\ (if\ \ (subtype\ t2\ t1)$

$then\ (c1::\{*\}\ .\ \{content\} := \ ((content\ c1::\{*\})\ ..\ c2::\{*\}\ \coloneqq\ c3::\{*\}::\{q\}))$

$else\ und))).$

7.8. MPL8: static arrays

The MPL8 language [1] is an extension of MPL7 that adds the static arrays feature: static array types, the array element access operation and the array element assignment statement.

The element ($\$t\ is\ (static\ array\ type)$) specifies static array types. It is defined by the rule

$(rule\ ((array\ t\ n)\ is\ (static\ array\ type))\ var\ (t)\ abn\ then\ ((t\ is\ type)\ and\ (n\ is\ nat))).$

The element ($\$t\ is\ (array\ type)$) is redefined by the extra rule

$(rule\ (t\ is\ (array\ type))\ var\ (t)\ abn\ then\ (t\ is\ (static\ array\ type))).$

The element ($\$e\ is\ (static\ array)$) specifies dynamic arrays. It is defined by the rule

$(rule\ ((v{:}\{content\},\ t{:}\{type\})::\{(static\ array)\}\ is\ (static\ array))\ var\ (v,\ t)\ abn$

$where\ (t\ is\ type)\ then\ (v\ is\ ((array\ content)\ t))).$

The element ($\$e\ is\ array$) is redefined by the extra rule

$(rule\ (e\ is\ array)\ var\ (e)\ abn\ then\ (e\ is\ (static\ array))).$

The element ($\$sar\ is\ (array\ \$t\ \$n)$) specifies static arrays with the given element type and length. It is defined by the rule

$(rule\ (sar\ is\ (array\ t\ n))\ var\ (sar,\ t)\ abn$

$where\ ((sar\ is\ (dynamic\ array))\ and\ (t\ is\ type))$

$then\ ((((element\ type)\ sar) = t::\{q\})\ and\ ((len\ (sar\ .\ \{content\})) = n))).$

The element ($type\ \$sar$) is defined by the rule

$(rule\ (type\ sar)\ var\ (sar)\ abn\ where\ (sar\ is\ (static\ array))$

$then\ (let::\{seq\}\ t,\ n\ be\ ((element\ type)\ sar),\ (len\ sar)\ in\ (array\ t\ n)::\{q\})).$

The array element assignment statement is redefined by the extra rule

$(rule\ (c1\ [\ c2\ ]\ \coloneqq c3)\ var\ (c1, c2, c3)\ val\ (c1, c2, c3)$

$abn\ (c1, c2, c3, c1::\{*\}, c2::\{*\}, c3::\{*\})\ abn$

$where\ ((c1::\{*\}\ is\ (static\ array))\ and\ (c2::\{*\}\ is\ nat)\ and$

$(c2::\{*\}\ <=\ (len\ c1::\{*\})))$

$then\ (let{::}\{und,\ seq\}\ t1,\ t2\ be\ ((element\ type)\ c1{::}\{*\}),\ (type\ c3{::}\{*\})$
$in\ (if\ (subtype\ t2\ t1)$
$then\ (c1{::}\{*\}.\{content\} := ((content\ c1{::}\{*\})\,..\ c2{::}\{*\} \coloneqq c3{::}\{*\}{::}\{q\}))$
$else\ und))).$

7.9. MPL9: structures

The MPL9 language [1] 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.

The element $((collect\ body\ members\ 1)\ \ (va^*)\ c^*)$ is redefined by the extra rules

$(rule\ ((collect\ body\ members\ 1)\ (structure\ st\ (tna\_s))\ var\ (st)\ seq\ (tna\_s, c\_s)\ abn\ (st)$
$abn\ where\ ((st\ is\ name)\ and\ (not\ (st\ is\ (structure\ type))))$
$then\ ((declare\ fields)\ st\ tfi\_s);\ (\{((structure\ type)\ st)\} \coloneqq true));$
$((collect\ body\ members\ 1)\ c\_s));$
$(rule\ ((collect\ body\ members\ 1)\ (structure\ st\ (tna\_s))\ var\ (st)\ seq\ (tna\_s,\ c\_s)$
$abn\ (st)\ abn\ then\ und).$

Thus, body members in MPL8 are variables, functions, procedures and structure types.

The element $((declare\ fields)\ \$st\ \$tna^*)$ declares the fields of $\$st$. It is defined by the rules

$(rule\ ((declare\ fields)\ st\ na\ t\ tna\_s)\ var\ (st,\ na, t)\ seq\ (tna\_s)\ abn$
$where\ ((na\ is\ name)\ \ and\ (t\ is\ type))$
$then\ (\{(type\ na\ st)\} \coloneqq t{::}\{q\}); (\{(fields\ na\ st)\} \coloneqq true);\ ((declare\ fields)\ \$st\ tna\_s);$
$(rule\ ((declare\ fields)\ st)\ var\ (st)\ abn\ then).$

The structure declaration is defined by the rule

$(rule\ (structure\ st\ (tna\_s))\ var\ (st)\ seq\ (tna\_s)\ abn\ then).$

The execution of the structure declaration does not collect information about the declared structure type, since the corresponding actions are performed by the element $((collect\ body\ members)\ \$c^*)$.

The element $(\$na\ is\ (structure\ type))$ specifies structure types. It is defined by the rule

$(rule\ (na\ is\ (structure\ type))\ var\ (na)\ abn\ where\ (na\ is\ normal)$
$then\ (.\ \{((structure\ type)\ na)\})).$

The element $(c\ is\ type)$ is redefined by the extra rule

$(rule\ (c\ is\ type)\ abn\ then\ (c\ is\ (structure\ type))).$

The element $(fields\ \$st)$ returns the sequence of fields of $\$st$. It is defined by the rule

$(rule\ (fields\ st)\ var\ (st)\ abn\ where\ (st\ is\ (structure\ type))$

$then\ (select\ fi\ wrt\ (v.\ \{(field\ fi\ st)\})\ var\ (v, fi)\ abn))$.

The element $(\$fi\ is\ (field\ \$st))$ checks that $\$fi$ is a field of $\$st$. It is defined by the rule

$(rule\ (fi\ is\ (field\ st))\ var\ (fi, st)\ abn\ where\ (st\ is\ (structure\ type))$

$then\ (.\ \{(field\ fi\ st)\}))$.

The element $(type\ \$fi\ \$st)$ returns the type of the field $\$fi$ of $\$st$. It is defined by the rule

$(rule\ (type\ fi\ st)\ var\ (fi, st)\ abn\ where\ (st\ is\ (structure\ type))\ then\ (.\ \{(type\ fi\ st)\}))$.

The element $(\$str\ is\ structure)$ specifies structures. It is defined by the rule

$(rule\ ((co{:}\{content\},\ st{:}\{type\}) :: \{structure\}\ is\ structure)\ var\ (co,\ st)\ abn$

$where\ (st\ is\ (structure\ type))\ then\ (co\ is\ ((structure\ content)\ st)))$.

The element $(\$e\ is\ ((structure\ content)\ \$st))$ is defined by the rules

$(rule\ (()\ is\ ((structure\ content)\ st))\ var\ (st)\ abn\ then\ true)$;

$(rule\ ((v{:}\{fi\}\ e\_s)\ is\ ((structure\ content)\ st))\ var\ (v, fi, st, e\_s)\ abn$

$where\ ((fi\ is\ (field\ st))\ and\ (let\ t\ be\ (type\ fi\ st)\ in\ (v\ is\ t)))$

$then\ ((e\_s)\ is\ ((structure\ content)\ st)))$.

The element $(\$str\ is\ \$st))$ specifies structures with the given type. It is defined by the rule

$(rule\ (str\ is\ st)\ var\ (str,\ st)\ abn$

$where\ ((str\ is\ structure)\ and\ (st\ is\ (structure\ type)))$

$then\ ((str\ .\ \{type\}) = st{::}\{q\}))$.

The element $\$str$ is defined by the rule

$(rule\ str\ var\ (str)\ abn\ where\ (str\ is\ structure)\ then\ str{::}\{q\})$.

The element $(type\ \$str)$ is defined by the rule

$(rule\ (type\ str)\ var\ (str)\ abn\ where\ (str\ is\ structure)\ then\ (str\ .\ \{type\}))$.

The element $(fields\ \$str)$ returns the sequence of fields of $\$str$. It is defined by the rule

$(rule\ (fields\ str)\ var\ (str)\ abn\ where\ (c{::}\{*\}\ is\ structure)$

$then\ (let\ st\ be\ (type\ str)\ in\ (fields\ st)))$.

The element $(\$fi\ is\ (field\ \$str))$ checks that $\$fi$ is a field of $\$str$. It is defined by the rule

$(rule\ (fi\ is\ (field\ str))\ var\ (fi, str)\ abn\ where\ (str\ is\ structure)$

$then\ (let\ st\ be\ (type\ str)\ in\ (fi\ is\ (field\ st))))$.

The element $(type\ \$fi\ \$str)$ returns the type of the field $\$fi$ of $\$str$. It is defined by the rule

$(rule\ (type\ fi\ str)\ var\ (fi, str)\ \ abn\ where\ (str\ is\ structure)$

$then\ (let\ st\ be\ (type\ \$str)\ in\ (type\ fi\ st)))$.

The structure field access operation is defined by the rule

$(rule\ (c\ .\ fi)\ var\ (c,\ fi)\ val\ (c)\ abn\ (c,\ c :: \{*\})\ abn$

```
  $where\ ((c :: \{*\}\ is\ structure)\ and\ (fi\ is\ (field\ c :: \{*\})))$
  $then\ ((str\ .\{content\})\ .\{fi\})).$
```

The structure field assignment statement is defined by the rule

```
(rule (c1 \. fi := c2) var (c1, fi, c2) val (c1, c2) abn (c1, c2, c1::{*}, c2::{*}) abn
 where ((c1::{*} is structure) and (fi is (field c1::{*})))
 then (let::{und, seq} t1, t2 be (type fi c1::{*}), (type c2::{*})
   in (if (subtype t2 t1)
     then (c1::{*} . {content} ≔ ((c1::{*} . {content}) . {fi} ≔ c2::{*}::{q}))
     else und))).
```

8. Conclusion

In the paper the notion of the conceptual operational semantics of a programming language has been proposed. The conceptual operational semantics of a programming language is an operational semantics of the programming language in terms of its conceptual model [3]. The special kind of CTSs, operational CTSs, oriented to specification of conceptual operational semantics of programming languages has been proposed, the language CTSL has been extended to this kind of CTSs, and the technique of the use of the extended CTSL as a domain-specific language for specification of conceptual operational semantics has been presented. We have conducted the incremental development of the conceptual operational semantics for the family of sample programming languages to illustrate this technique.

There is only one more approach which, like our approach, can specify both the structural and dynamic parts of the operational semantics of a programming language in quite general unified way. This approach is based on abstract state machines (ASMs) [4]. ASMs are the special kind of transition systems in which states are algebraic systems.

The key features of our approach in comparison with the approach based on ASMs are as follows.

The instantiation semantics and, in particular, states are directly described in CTSs in ontological terms whereas its conceptual structure can be only modelled by the appropriate choice of symbols of the signature of an algebraic system.

The transition relation in ASMs is built with the finite set of algebraic operations [5]. The transition relation in operational CTSs is based on the pattern matching on the conceptual structure of states.

The set of predefined executable elements of the CTSL language have analogues for the algebraic operations used in sequential ASMs, and also includes the elements for parsing the conceptual state structure.

The languages of executable specifications of abstract state machines AsmL [6] and XasM [7] are general-purpose languages of specification of discrete dynamic systems. They are not domain-specific languages oriented to development of operational semantics of programming languages in contrast to the CTSL language.

At present, our technique is applied to only the sequential fragments of programming languages. We plan to extend it to the concurrent fragments of programming languages.

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. Anureev I.S., Promsky A.V. Conceptual transition systems and their application to development of conceptual models of programming languages // System Informatics. 2017. Vol. 9. P. 133–154.
4. 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.
5. Borger E., Stark R.F. Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Secaucus. 2003.
6. AsmL: The Abstract State Machine Language. Reference Manual, 2002. http://research.microsoft.com/en-us/projects/asml/
7. Matthias Anlauff. XasM — An Extensible, Component-Based Abstract State Machines Language. http://xasm.sourceforge.net/XasmAnl00/XasmAnl00.html