УДК 004.052, 519.179.2

A Verification Method for a Family of Multi-agent Systems of Ambiguity Resolution»

Natalia Garanina (A.P. Ershov Institute of Informatics Systems),

Elena Sidorova (A.P. Ershov Institute of Informatics Systems)

In the paper we describe a verification method for families of distributed systems generated by context-sensitive network grammar of a special kind. The method is based on model checking technique and abstraction. A representative model depends on a specification grammar for family of systems. This model simulates a behavior of the systems in such a way that properties which hold for the representative model are satisfied for all these systems. We show using this method for verification of some properties of multiagent system for resolution of context-dependent ambiguities in ontology population.

Keywords: model checking, context-sensitive network grammar, multi-agent systems, abstraction

1. Introduction

The motivation of our work is the ambiguity resolution problem in the frame of ontology population from natural language texts. In [6] we describe text analysis algorithms producing a system of information agents. But features of natural language cause ontology population ambiguities, which these agents have to resolve. We proposed to evaluate the cardinality of agents' contexts, i.e. how much an agent is related with the other agents of the resulting system via the information contained in it, and to mark the agents the most integrated in the text. We developed an ambiguity resolution algorithm [5], removing the less integrated agents from the system.

All agents in parallel perform rather complicate protocols with periodic local synchronizations. Hence, it is reasonable to use formal verification methods for proving correctness of the algorithm. We choose model checking technique for a particular multi-agent system. We verify rather specific multi-agent system of conflict resolution. The works on multi-agent systems usually focus on the behavior of agents, methods of communication between agents, knowledge and belief of an agent about environment and other agents, etc [4, 9]. Works about conflict resolution process usually consider the process in terms of the behavior of the agent depending on its internal state, reasoning and argumentation methods etc. [8]. The dynamics of the agents connections is not a subject of these researches. There are papers related to the dynamics of weighted connections, but they are not the typed and their changes does not affect the internals of the agent [7]. On the other hand there are works on the study of social networks, in which the agents are connected by the typed connections, but their weight does not matter [1]. To the best of our knowledge, there are no works on model checking for a conflict resolution algorithm of the suggested type.

Model checking technique is widely used for verification of distributed and multiagent systems [2]. In our case we would like to verify not a particular agent network, but infinite family of such systems. For verification of infinite network families the model checking method was suggested in [3]. This method is based on using a context-free network grammar generating families of distributed systems, and on abstraction by finite automata. The idea of the method is to construct an invariant network based on a given grammar. This invariant simulates behavior of all systems in the family and is consistent with abstract functions associated with properties to be verified which are expressed by branching time logic $\forall CTL$. Due to consistent simulation, properties holding for the representative invariant also holds for all systems in the family. But authors studied context-free grammars only, while our model of the multiagent system is generated by a context-sensitive grammar of a special kind. In the paper we define such network grammar by adding notions of a quasi-terminal and a merging operator to the standard definition. We show that this verification method still can be used for network families generated by the new grammar.

The rest of the paper is organized as follows. The next section 2 gives base definitions. Section 3 presents results on a new merging operator, used in our context-sensitive network grammar. Section 4 describes using our method for the multiagent system of ambiguity resolution. We conclude in the last section 5 with a discussion of further research.

Acknowledgments. The research has been supported by Russian Foundation for Basic Research (grant 15-07-04144) and Siberian Branch of Russian Academy of Science (Integration Grant n.15/10 “Mathematical and Methodological Aspects of Intellectual Information Systems”).

2. Base Definitions

Let us give necessary definitions from [3] in a modified form. Modification concerns a merging operator and quasi-terminals in a network grammar.

Definition 1.

A Labeled Transition System (LTS) is a structure $M = (S, R, ACT, S_0)$, where

- S is the set of states,
- $S_0 \subseteq S$ is the set of initial states,
- ACT is the set of actions, and
- $R \subseteq S \times ACT \times S$ is the total transition relation, such that for every $s \in S$ there is action a and state s' for which $(s, a, s') \in R$ (denote as $s \xrightarrow{a} s'$).

Let L_{ACT} be the class of LTSs whose set of actions is a subset of ACT. Let $L_{(S,ACT)}$ be the class of LTSs whose state set is a subset of S and whose action set is the subset of ACT. Let $ACT_1, ACT_2 \subseteq ACT$. Let we are given two LTSs $M_1 = (S_1, R_1, ACT_1, S_0^1)$ and $M_2 = (S_2, R_2, ACT_2, S_0^2)$ in the class L_{ACT}.

Definition 2.

A function $\|$: $L_{ACT} \times L_{ACT} \mapsto L_{ACT}$ is called a *composition function* iff $M_1 \parallel M_2$ has the form $(S_1 \times S_2, R', ACT_1 \cup ACT_2, S_0^1 \times S_0^2)$.

A function $\cup : L_{ACT} \times L_{ACT} \mapsto L_{ACT}$ is called a *merging function* iff $M_1 \cup M_2$ has the form $(S_1 \cup S_2, R', ACT_1 \cup ACT_2, S_0^1 \cup S_0^2)$.

The definition of R' depends upon the exact semantics of the composition and merging function. Let S^i be words of length i with S as the alphabet.

Definition 3.

Given a state set S and a set of actions ACT, any subset of $\bigcup_{i=1}^{\infty} L_{(S^i,ACT)}$ is called a network on the tuple (S, ACT).

We give a definition of a *context-sensitive network grammar with quasi-terminals (CSNQ-grammar)* to describe networks, which is the modified definition of a context-free network grammar from [3]. The set of all LTSs derived by a network grammar forms a network which is an LTS also. Let S be a state set and ACT be a set of actions. *CSNQ-grammar* $G = (T, Qt, t, N, P, S)$ is a grammar, where

- T is a set of terminals, each of which is an LTS in $L_{(S,ACT)}$, these LTSs are sometimes referred to as basic processes,

- Qt is a set of quasi-terminals, each of which is an LTS in $L_{(S,ACT)}$, their merging gives an LTS,
- a mapping $t : Qt \mapsto T$ associates quasi-terminals to terminals,
- N is a set of nonterminals, each nonterminal defines a network,
- P is a set of production rules of the following forms:
  - $A \longrightarrow B \parallel_i C$, where $A \in N$, and $B, C \in T \cup Qt \cup N$, and $\parallel_i$ is a composition function.
  - $\{A_1, ...A_n\} \longrightarrow t(A_1) \cup_i ... \cup_i t(A_n)$, where $A_j \in Qt$, and $\cup_i$ is a merging function.
- $S \in N$ represents the network generated by the grammar.

Note, that this grammar is context-free with respect to composition functions and context-sensitive with respect to merging functions.

In order to express properties of a model composed from finite, but unspecified number of LTSs, we define a finite automaton on alphabet S.

Definition 4.

$D = (Q, q_0, \delta, F)$ is a *deterministic automaton* over S, where

- Q is the set of automaton states,
- $q_0 \in Q$ is the initial state,
- $\delta \subseteq Q \times S \times Q$ is the transition relation,
- $F \subseteq Q$ is the set of accepting states, and
- $L(D) \subseteq S^*$ is the set of words accepted by D.

We use finite automata over S for specification of atomic state properties. Let D be an automaton over S. State s satisfies D ($s \models D$) iff $s \in L(D)$. A specification language is a universal branching temporal logic $\forall CTL$ [3] with finite automata over S as the atomic formulas. Syntax of $\forall CTL$ consists of formulas that are composed of Boolean constants, atomic formulas, connectives $\neg$, $\vee$, $\wedge$, and branching time modalities $\mathbf{AX}\varphi$, $\mathbf{AG}\varphi$, and $\varphi\mathbf{AU}\psi$ with standard semantics.

Recall definitions for abstract LTS from [3]. For the simplicity, here the specification language contains a single atomic formula D. Given an automaton $D = (Q, q_0, \delta, F)$ and a word $w \in S^*$ the function induced by w on Q, $f_w : Q \mapsto Q$, is defined by $f_w(q) = q'$ iff $q \xrightarrow{w} q'$. Note that $w \in L(D)$ if and only if $f_w(q_0) \in F$. Two states s and s' are *equivalent* $s \equiv s'$ iff $f_s = f'_s$. The function f_s is called the *abstraction* of s and is denoted by $h(s)$. Relation $\models$ is extended to abstract states: $h(s) \models D$ iff $f_s(q_0) \in F$. Hence $s \models D$ iff $h(s) \models D$.

Let F_D be the set of functions corresponding to the deterministic automaton D. The abstraction function h extended to F_D is defined by $h(f) = f$ for $f \in F_D$ and extension the function h to $(S \cup F_D)$ is $h((a_1, a_2, ..., a_n)) = h(a_1) \circ ... \circ h(a_n)$. From now on we consider LTSs in the network N on the tuple $(S \cup F_D, ACT)$.

Definition 5. *(of abstract LTS)*
Given an LTS $M = (S^i, R, ACT, S_0)$ in the network N, the corresponding abstract LTS is defined by $h(M) = (S^h, R^h, ACT, S_0^h)$, where

- $S^h = \{h(s) | s \in S^i\}$ is the set of abstract states,
- $S_0^h = \{h(s) | s \in S_0\}$, and
- the relation R^h is defined as follows. For any $h_1, h_2 \in S^h$, and $a \in ACT$:
  $(h_1, a, h_2) \in R^h \Leftrightarrow \exists s_1, s_2 [h_1 = h(s_1) \wedge h_2 = h(s_2) \wedge (s_1, a, s_2) \in R]$.

M' simulates M (denoted $M \preceq M'$) iff there is a simulation preorder $E \subseteq S \times S'$ ($(s, s') \in E$ denoted $s \preceq s'$) that satisfies the following conditions: for every $s_0 \in S_0$ there is $s_0' \in S_0'$ such that $s_0 \preceq s_0'$. For every s, s', if $s \preceq s'$ then

- $h(s) = h(s')$, and
- for every s_1 such that $s \xrightarrow{a} s_1$ there is s_1' such that $s' \xrightarrow{a} s_1'$ and $s_1 \preceq s_1'$.

3. The Merging Operator in the Verification Framework

The first two propositions of the following lemma were proved in [3], the last is proved below:

Lemma 1.

1. $M \preceq h(M)$, i.e., $h(M)$ simulates M.
2. If $M \preceq M'$, then $h(M) \preceq h(M')$.
3. $M \cup M' \preceq h(M) \cup h(M')$

Proof of (3) is obvious: $M \cup M' \preceq h(M \cup M')$ due to (1), and $h(M \cup M') = h(M) \cup h(M')$.$\square$

The following theorem about satisfiability of properties in an LTS and its simulator was proved in [3] and holds for our new framework.

Theorem 1.
Let φ be a formula in $\forall CTL$ over the atomic formula D. Let M and M' be two LTSs such that $M \preceq M'$. Let $s \preceq s'$. Then $s' \models \varphi$ implies $s \models \varphi$.

Definition 6.
A merging or composition operator $\bullet \in \{\cup, \|\}$ is called *monotonic* with respect to a simulation preorder $\preceq$ if and only if given LTSs such that $M_1 \preceq M_2$ and $M_1' \preceq M_2'$, we have that $M_1 \bullet M_1' \preceq M_2 \bullet M_2'$. A network grammar G is called monotonic if and only if all rules in the grammar use only monotonic composition and merging operators.

We modify a synchronous framework from [3] with results for the merging operator. Let models be a form of LTSs, Moore machines $M = (S, R, I, O, S_0)$ such that inputs I and outputs O must be disjoint. In addition, they have a special internal action denoted by τ. The set of actions is $ACT = \{\tau\} \cup 2^{I \cup O}$, where each noninternal action is a set of inputs and outputs. A transition $s \xrightarrow{a} t$ from s in a machine M with $a = i \cup o$ such that $i \subseteq I$ and $o \subseteq O$ occurs only if the environment supplies inputs i and the machine M produces the outputs o.

Naturally, for the merging operator inputs and outputs of merging machines must be disjoint also. Let $I \cap O' = \emptyset$ and $O \cap I' = \emptyset$. The merging of M and M', $M'' = M \cup M'$ is defined by

- $S'' = S \cup S'$,
- $S_0'' = S_0 \cup S_0'$,
- $I'' = I \cup I'$ and $O'' = O \cup O'$, and
- $s'' \xrightarrow{a''} s_1''$ is a transition in R'' iff the following holds: $s'' \xrightarrow{a} s_1''$ is a transition in R and $s'' \xrightarrow{a'} s_1''$ is a transition in R' for some a, a' such that $a'' = a$ or $a'' = a'$.

Lemma 2.
The merging $\cup$ is monotonic with respect to $\preceq$.

Proof. Let $M = (S, R, I, O, S_0)$, $M_1 = (S_1, R_1, I_1, O_1, S_{1,0})$, $M' = (S', R', I', O', S_0')$, $M_1' = (S_1', R_1', I_1', O_1', S_{1,0}')$ be four Moore machines. Assume that $M \preceq M_1$ and $M' \preceq M_1'$. Let $E \subseteq S \times S_1$ and $E' \subseteq S' \times S_1'$ be the corresponding simulation relations. We prove that $M \cup M' \preceq M_1 \cup M_1'$.

We say that $(s'', s_1'') \in E''$ iff $(s'', s_1'') \in E$ or $(s'', s_1'') \in E'$. We show that E'' has the required properties. It is clear from the definition that given state $s_0 \in S_0 \cup S_0'$, there exists $s_{0,1} \in S_{0,1} \cup S_{1,0}'$ such that $(s_0, s_{0,1}) \in E \cup E'$.

Assume that $(s, s_1) \in E \cup E'$.

(1) By assumption, we have that $h(s) = h(s_1)$.

(2) Let $s \xrightarrow{a''} t$ be a transition in $M \cup M'$. This means that there exists transition $s \xrightarrow{a} t$ in M or transition $s \xrightarrow{a'} t$ in M' such that $a'' = a$ or $a'' = a'$. By definition there exists $t_1 \in S_1 \cup S_1'$

such that $s_1 \xrightarrow{a} t_1$ or $s_1 \xrightarrow{a'} t_1$, where $(t, t_1) \in E$ or $(t, t_1) \in E'$. Therefore, $s_1 \xrightarrow{a''} t_1$ and $(t, t_1) \in E''$. The proof is thus complete. □

The notion of a representative give us a way to construct a simulation invariant. Given a CSNQ-grammar G, we associate with each symbol A of the grammar *a representative process* $rep(A)$. Let us adopt the definition of a monotonicity property for a set of representative processes of CSNQ-grammar:

- for every terminal and quasi-terminal A: $h(rep(A)) \succeq h(A)$, and
- for every rule $A \longrightarrow B \parallel C$: $h(rep(A)) \succeq h(h(rep(B)) \parallel h(rep(C)))$.

We extend the proof of the following theorem on context-free network grammar from [3] to CSNQ-grammars:

Theorem 2.

Let G be a monotonic grammar and suppose we can find representatives for the symbols of G that satisfy the monotonicity property. Let A be a symbol of the grammar G, and let a be an LTS derived from A using the rules of the grammar G. Then, $h(rep(A)) \succeq a$.

Proof. We prove that $h(rep(A)) \succeq h(a)$. Since $h(a) \succeq a$, the result follows by transitivity. Let $A \Rightarrow^k a$, i.e., A derives a in k steps. Induction on k.

($k = 0$) Proved in [3].

($k = 1$) In the case A, B are quasi-terminals in a rule $A, B \longrightarrow t(A) \cup t(B)$ and $a = t(A) \cup t(B)$. The result follows from the monotonicity property and Lemma 1.

($k \geq 1$) Proved in [3]. □

Verification method is exactly the same as in [3]. Assume that we are given monotonic grammar G and $\forall CTL$ formula φ with atomic formulas $D_1, ..., D_k$. To check that every LTS derived by the grammar G satisfies φ we perform the following steps:

1. For every symbol A in G choose representative process $rep(A)$ and construct the abstract LTS $h(rep(A))$ with respect to the formulas $D_1, ..., D_k$.
2. Check that the set of representatives satisfies the monotonicity property. Theorem 2 implies that for every a derived by the grammar G, $h(rep(S)) \succeq a$.
3. Perform model checking on $h(rep(S))$ with specification φ. By Theorem 1, if $h(rep(S)) \models \varphi$, then for all LTSs M derived by the grammar G, $M \models \varphi$.

For finding monotonic representatives we could use an algorithm from [3] setting $\{t(A)\}$ as an initial representative association set of every quasi-terminal A.

4. Verification of Multiagent Ambiguity Resolution

A detailed description of the multiagent algorithm for ambiguity resolution in ontology population is given in [5]. In this paper we sketch a communication structure without considering agents' actions on message processing.

Let a set of *information agents* be given. Some of agents are *in a conflict* corresponding to some ambiguity. An *agent-master* constructs a conflict-free set of information agents taking into account integration of conflict agents in the system. This integration is evaluated by computing weights and conflict weights of the agents. A conflict is resolved by removing a weak agent from the system. The agent-master performs the main protocol of constructing the conflict-free set, while the information agents perform protocols of computing their weights.

Every information agent is connected with the master by two-way channel. Information agents are linked with others by labeled connections of two types corresponding their conflict reaction: removing (*rem*-type) and updating (*upd*-type). Every labeled connection is acyclic. Processing of every conflict reaction induced by specified connection is considered to be certain base process. An information agent can be union of such processes. This fact specifies a form of a grammar generating family of our multiagent systems for various number of agents connected in various ways. Agents are connected by two-way channels corresponding to these labeled connections. This structure of multiagent network is generated by the following context-sensitive grammar with quasi-terminals $G = (T, Qt, t, N, P, S)$. Let a set of connections be $C = \{c_1, ..., c_n\}$ and c_i^k be a connection having conflict type $k \in \{rem, del\}$.

- terminals $T = \{master\} \cup \bigcup_{i=1}^{n} \{root_i, inter_i, leaf_i\} \cup vrtxs^i$, and $vrtxs^i = \{vrtx \mid vrtx = inter_j$ or $vrtx = leaf_j, j \in [1..n]\}$ and $|vrtxs^i| = i$,
- quasi-terminals $Qt = \bigcup_{i=1}^{n} \{INTER_i, LEAF_i\}$,
- associate mapping $t : Qt \mapsto T$ is defined by $t(INTER_i) = inter_i$, and $t(LEAF_i) = leaf_i$ for every $i \in [1..n]$,
- nonterminals $N = \{S\} \bigcup_{i=1}^{n} \{ROOT_i, SUB_i\}$;
- set of production rules P for every $i \in [1..n]$:
  1. $S \longrightarrow master \parallel_m ROOT_1 \parallel_m \ldots \parallel_m ROOT_n$
  2. $ROOT_i \longrightarrow (ROOT_i \parallel_{c_i^k} SUB_i) \bigvee (root_i \parallel_{c_i^k} SUB_i)$
  3. $SUB_i \longrightarrow (SUB_i \parallel_{c_i^k} SUB_i) \bigvee (INTER_i \parallel_{c_i^k} SUB_i) \bigvee$
     $(SUB_i \parallel_{c_i^k} LEAF_i) \bigvee (INTER_i \parallel_{c_i^k} LEAF_i) \bigvee$
     $(inter_i \parallel_{c_i^k} SUB_i) \bigvee (SUB_i \parallel_{c_i^k} leaf_i) \bigvee$

$(inter_i \parallel_{c_i^k} LEAF_i) \bigvee (INTER_i \parallel_{c_i^k} leaf_i) \bigvee (inter_i \parallel_{c_i^k} leaf_i)$

4. $\{V_1, ..., V_m\} \longrightarrow t(V_1) \cup ... \cup t(V_m) = vrtx^m$, where for every $j \in [1..m]$ $V_j \in \{INTER_i, LEAF_i\}$, and if $V_j = INTER_i$ then for every $l \in [1..m]$ holds $V_l \neq LEAF_i$ ($i \in [1..n]$).

Parallel composition of agent-processes is synchronous. Protocols for computing weights and conflict weights are highly parallel. Hence it is very important to prove that they terminate and are synchronized properly. Satisfiability of these properties is necessary for correctness of weight computing. Launch of these computing could be modeled by sending tokens.

Every base process is defined by the following state variables:

- $Name: int$ is a name of the process;
- $Channel$: set of $\{name: int; c\_type: bool; dir: bool; agn: int; rmvd: bool\}$, where $name$ is a label of a connection, c_type is its type, dir is a direction: a child ($dir = 0$) or a parent ($dir = 1$) named agn, and $rmvd$ is an absence status;
- $Rmvd: bool$ is an absence status;
- $Active: bool$ is an activity status;
- $WasActive: bool$ is a previous activity status.

In synchronous composition of base processes with different names the corresponding channels of the same name must connect. In merging of processes with the same $Name$ sets of channels and sets of $Channel$ join. Processes with different names cannot be merged and processes with the same $Name$ cannot be composed in parallel. Values of above variables define states of a base process. Its input and output channels correspond to names, types and directions of $Channel$. Transitions are defined by sending and receiving tokens through the channels. The initial state is $(Channel, 0, 0, 0)$, where $Channel$ is a nonempty set of channels with $Channel.rmvd = 0$, and a number of channels with $dir = 1$ does not exceed 1 and a number of channels with $dir = 0$ can be equal to 0.

We would like to verify the following properties expressed by $\forall CTL$. For the protocol of parallel weight computing: $\mathbf{AF}(\{wasActive\}^* \wedge \mathbf{AXAF}\{\neg Active\}^*)$ (every agent was active, and then all computation will be terminated). For the protocol of conflict weight computing: $\mathbf{AF}\{\neg Active\}^*$ (all computation will be terminated); $\mathbf{AG}\{Not2Rmvd\}^*$ (Channels and agents cannot be removed twice). For every atomic formula we construct a finite deterministic automaton. They are a base for abstract functions for states of our systems. Then we should construct a set of consistent representatives for symbols of our grammar. This technique is not present here.

5. Conclusion

In the paper we present the verification method for families of distributed systems specified by a context-sensitive grammar with quasi-terminals. This method can be used for verification of the multi-agent system of ambiguity resolution in ontology population. Properties of the system are expressed by $\forall CTL$-formulas.

In the near future we plan to implement the suggested method using model checking tool SPIN and give formal proofs of correctness of the ambiguity resolution algorithm. But some properties concerning agent interaction cannot be expressed easily in this framework. This fact is a reason for trying other more expressive formalisms for properties. Other research direction is to extend the method for other types of context-sensitive grammars.

Список литературы

1. Bergenti F., Franchi E., Poggi A. Selected models for agent-based simulation of social networks // In: Procs. 3rd Symposium on Social Networks and Multiagent Systems (SNAMAS 2011) 2011, pp. 27-32.
2. Clarke E.M., Grumberg O., Peled D. Model Checking. MIT Press, 1999.
3. Clarke E.M., Grumberg O., Jha S. Verifying Parameterized Networks // In: ACM Transactions on Programming Languages and Systems, Vol. 19, No. 5, September 1997. Pages 726-750.
4. Fagin R., Halpern J.Y., Moses Y., Vardi M.Y. Reasoning about Knowledge. MIT Press, 1995.
5. Garanina N., Sidorova E. An Approach to Ambiguity Resolution for Ontology Population // Proc. of the 24th International Workshop on CS&P. Rzeszow, Poland, Sep. 28-30, 2015. – University of Rzeszow, 2015, Vol. 1, pp 134-145.
6. Garanina N. O., Sidorova E. A. Ontology Population as Algebraic Information System Processing Based on Multi-agent Natural Language Text Analysis Algorithms//Programming and Computer Software, 2015, V. 41, n.3, pp. 140–148.
7. De Gennaro M.C., Jadbabaie, A. Decentralized Control of Connectivity for Multi-Agent Systems // In: Proc. of 45th IEEE Conference on Decision and Control, pp. 3628 - 3633.
8. Huhns M. N., Stephens L. M. Multiagent Systems and Societies of Agents // In: Multiagent Systems, MIT Press, 1999 pp. 79–120.
9. Wooldridge, M. An Introduction to Multiagent Systems. Willey&Sons Ltd, 2002.

УДК 004.415.53

Верификация промышленных алгоритмов управления методом Model checking в сочетании с концепцией виртуальных объектов управления

Лях Т.В. (Институт автоматики и электрометрии СО РАН, Новосибирский государственный университет),

Зюбин В.Е. (Институт автоматики и электрометрии СО РАН, Новосибирский государственный университет)

На сегодняшний день текущая практика промышленной автоматизации такова, что тестирование управляющих алгоритмов в подавляющем большинстве случаев начинается только при запуске ПО на реальном объекте. В результате проверка алгоритма откладывается до этапа пуско-наладочных работ на объекте автоматизации. В статье предложен подход к тестированию алгоритмов управления на основе концепции виртуальных объектов управления. Для гарантии, что алгоритм управления удовлетворяет полностью накладываемым на него требованиями, используется метод верификации Model checking.

Ключевые слова: *Алгоритмы управления, промышленная автоматизация, процесс-ориентированное программирование, язык Reflex, верификация, Model checking..*

1. Введение

На сегодняшний день текущая практика промышленной автоматизации предполагает, что автоматизированные системы управления создаются исключительно на базе цифровой техники в виде программно-аппаратных комплексов. При этом на современном этапе наблюдается четкая тенденция к усложнению программной составляющей таких систем, повышению ее функциональности и общей трудоемкости ее реализации. Рост значимости программного обеспечения в области промышленной автоматизации, высокая стоимость логических ошибок в программах давно уже находятся в противоречии с текущей практикой разработки управляющих программ, которая ведется в рамках водопадной модели. Тестирование управляющих алгоритмов в подавляющем большинстве случаев начинается только при запуске ПО на реальном объекте. В результате проверка алгоритма откладывается до этапа пуско-наладочных работ на объекте автоматизации. Такая практика чревата высокими рисками, нештатными ситуациями или даже авариями на объекте.

Для решения проблемы тестирования управляющих алгоритмов в Институте автоматики и электрометрии СО РАН была предложена концепция виртуальных объектов управления (ВОУ) – программных имитаторов автоматизируемого технического процесса, со свойствами, схожими со свойствами моделируемого объекта [1]. Код ВОУ (Рис. 1) исполняется независимо от алгоритма управления (АУ), создаваемого разработчиком. Унифицированный обмен данными между ВОУ и алгоритмом управления обеспечивает сохранение связей при изменении алгоритма. Такой подход позволил использовать итерационную модель разработки и отлаживать код алгоритма управления до этапа пуска-наладки.

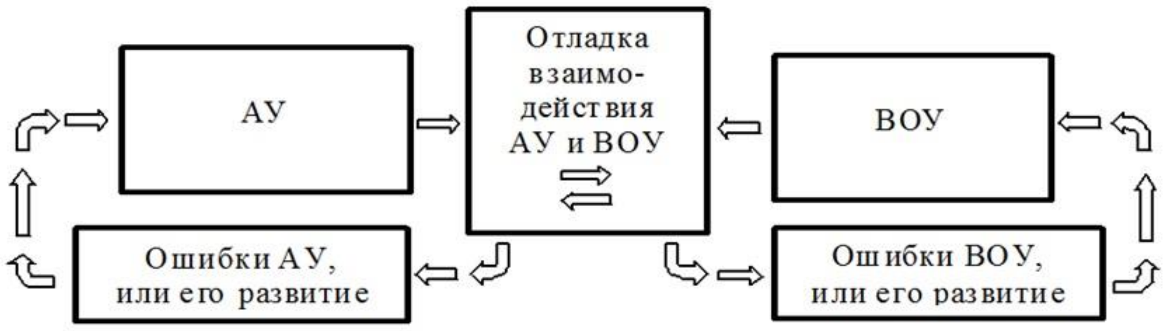
АУ
Отладка взаимо-действия АУ и ВОУ
ВОУ
Ошибки АУ, или его развитие
Ошибки ВОУ, или его развитие

Рис. 1. Итерационная модель разработки алгоритма управления (АУ) с использованием виртуального объекта управления (ВОУ)

Было создано ПО на базе среды LabVIEW, которое позволило запускать одновременно и тестировать ВОУ и АУ, имитируя обмен данными и внешние события [2], и с помощью него был отлажен алгоритм управления Большим солнечным вакуумным телескопом.

Однако в процессе работы был выявлен ряд недостатков такого подхода. Эти недостатки связаны с тем, что тестирование не обеспечивает полноту покрытия тестами всей функциональности алгоритма. Также при таком подходе множество операций приходилось выполнять вручную оператору.

Это растягивало процесс тестирование и увеличивало возможность пропущенных ошибок и неисследованных поведений алгоритма управления. Поэтому для автоматизации проверки алгоритма управления было предложено воспользоваться подходом Model chescking. Этот подход хорошо себя зарекомендовал при верификаии реаигирующих систем, т.е. систем, взаимодействующих с окружением. На данный момент подход Model checking активно развивается.

В статье рассматриваются особенности разработки управляющих алгоритмов промышленного уровня, исследуются проблемы тестирования алгоритмов управления в приборостроении, описывается подход к тестированию алгоритмов управления на основе концепции виртуальных объектов управления. Предложен способ автоматизации тестирования АУ с использованием подхода Model checking [3].

2. Специфика разработки алгоритмов управления и преимущества языка Reflex

Задачи автоматизации имеют ряд характерных особенностей, и потому на ПО и языки программирования, которые используются в разработке АУ, накладывается ряд особых требований. Поскольку система управления воздействует на объект управления через органы управления и реагирует события на объекте так, как это определено в алгоритме управления, от управляющего алгоритма требуется цикличность: он считывает входные сигналы, обрабатывает и формирует выходные сигналы. От алгоритма также требуется адекватность реакции по времени событиям на объекте, т.е. синхронизация исполнения алгоритма с физическими процессами во внешней среде. Поскольку на объекте управления зачастую множество процессов возникают и протекают одновременно, требуется, чтобы средства разработки предоставляли возможность обеспечить логический параллелизм алгоритма.

Благодаря этим особенностям стратегии создания управляющих алгоритмов в промышленной автоматизации и используемые языки программирования отличаются от практики, применяемой при создании ПО, не взаимодействующего с реальными объектами.

Для создания промышленных алгоритмов управления используется множество подходов: языки стандарта МЭК 61131-3, языки общего назначения (такие, как C, C++ или Delphi), языково-ориентированное программирование с использованием предметно-ориентированных языков и прочее [4]. Каждый из таких подходов имеет определенные преимущества и недостатки, и, не вдаваясь глубоко в детали, следует упомянуть, что в конечном счете выбор делается в пользу того решения, которое наилучшим образом соответствует особенностям автоматизируемого объекта. Однако в последнее время в связи с недостатками стандарта МЭК [5] наблюдаемая тенденция такова, что при разработке промышленных алгоритмов управления все чаще отказываются от языков МЭК в пользу либо языков общего назначения, либо новых, специализированных для узкого применения, формализмов.

Процесс-ориентированный язык Reflex был создан для описания алгоритмов управления при решении задач промышленной автоматизации [6]. Язык Reflex отличается рядом достоинств:

1) Адекватность задачам промышленной автоматизации;
2) Легкость в изучении;
3) Язык Reflex – высокоуровневый язык программирования, разработчику не требуется работать в терминах низкоуровневых операций с оборудованием;
4) Алгоритмы, созданные на языке Reflex, не зависят от среды исполнения;
5) Язык Reflex допускает вызовы функций, написанных на других языках программирования.

3. Концепция виртуальных объектов управления на базе LabVIEW с использованием языка Reflex

Концепция ВОУ для итерационной разработки АУ была реализована с использованием механизма DLL, пакета LabVIEW [7] и транслятора языка Reflex. Интерфейс был создан средствами пакета прикладных программ технических вычислений LabVIEW, который широко используется для имитационного моделирования. Алгоритм управления и описание ВОУ создается на языке Reflex.

При итерационной разработке алгоритма управления на основе концепции ВОУ работа происходит по схеме, изображенной на рис. 2:

1) На языке Рефлекс создается описание логически обособленной части АУ (например, часть алгоритма, отвечающая за определенную функциональность);
2) На языке Рефлекс создается описание элемента ВОУ, соответствующего функционированию этого АУ;
3) ВОУ и АУ транслируются в DLL, которые встраиваются в отладочное ПО. Дополнительно транслятор создает конфигурационные файлы, которые автоматически интегрируются в отладочное ПО;
4) Оператор за отладочным интерфейсом проводит тестирование блока АУ: запускает одновременно и тестирует пошагово АУ и ВОУ, имитирует передачу данных от оператора интерфейса управления для АУ, имитирует передачу данных с датчиков объекта управления. Это позволяет имитировать нештатные ситуации на объекте управления: аварии, поломки оборудования и отсутствие связи с оборудованием – и оценивать реакцию АУ на них;

5) Если было выявлено несоответствие поведения АУ требованиям спецификации, или же найдены ошибки в описании АУ или ВОУ на языке Reflex, вносятся изменения в код АУ или ВОУ, и трансляция и тестирование происходят заново;
6) Если тестирование прошло успешно, не было выявлено ошибок в описаниях ВОУ и АУ, и было установлено, что АУ удовлетворяет накладываемым на него признакам, описывается следующий логический модуль АУ и создается соответствующий блок ВОУ.

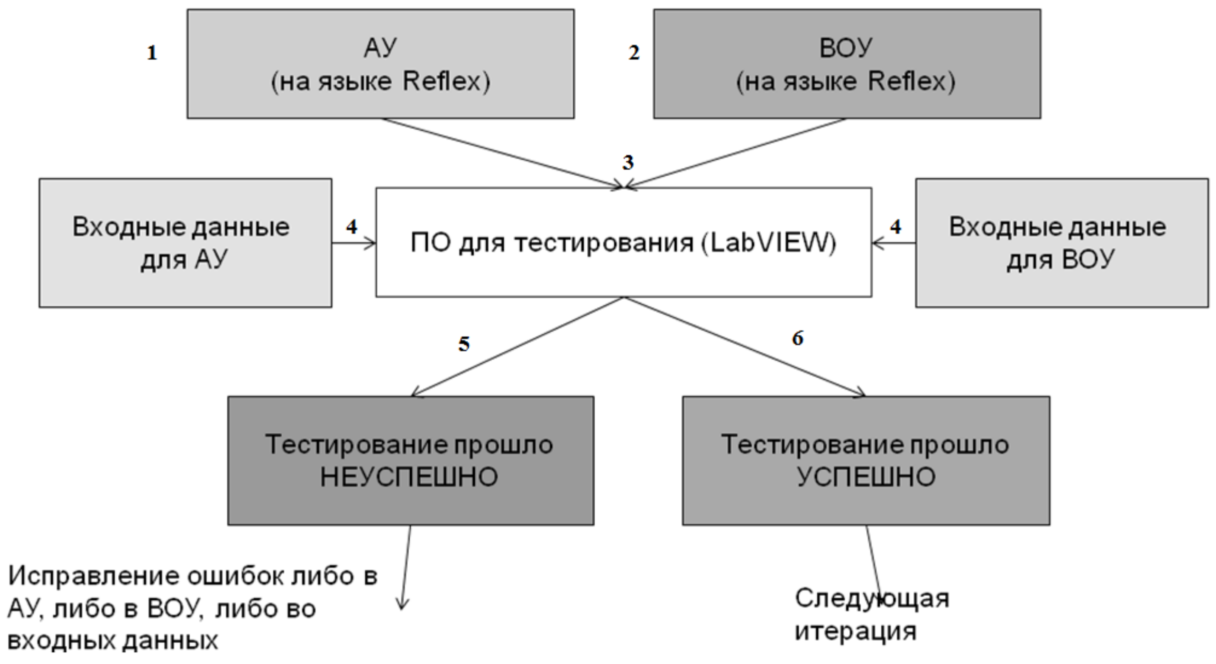
1
АУ (на языке Reflex)
2
ВОУ (на языке Reflex)
3
Входные данные для АУ
4
ПО для тестирования (LabVIEW)
4
Входные данные для ВОУ
5
6
Тестирование прошло НЕУСПЕШНО
Тестирование прошло УСПЕШНО
Исправление ошибок либо в АУ, либо в ВОУ, либо во входных данных
Следующая итерация

Рис. 2. Итерационная схема разработки алгоритмов управления на основе концепции ВОУ

Рассмотренная выше схема разработки была опробована при разработке алгоритма управления системы вакуумирования Большим солнечным вакуумный телескопом (БСВТ, г Иркутск, поселок Листвянка).

3.1. Недостатки разработанного метода проверки безопасности АУ

Однако при работе над системой вакуумирования БСВТ было обращено внимание на ряд неудобств данного подхода к разработке АУ:

- Большое количество работы «вручную». Оператору приходится вручную запускать ВОУ и АУ, тестировать их пошагово, имитировать передачу данных от оператора интерфейса управления для АУ и передачу данных с датчиков объекта управления;
- Полнота покрытия алгоритма тестами неизвестна. Тестирование не способно дать точный ответ, насколько полно тесты покрывают функциональность АУ, и насколько

точно выполняются требования, накладываемые на АУ. К тому же, тестирование чаще всего выявляет частые ошибки, в то время как редкие, но критические ошибки, могут ускользнуть от внимания. С помощью тестирование методом «черного ящика» невозможно доказать отсутствие ошибок в программе.

Для точного доказательства, что АУ удовлетворяет накладываемым на него требованиям, необходим строгий и непротиворечивый математический аппарат.

4. Использование метода Model checking для тестирования АУ

Для проверки корректности АУ в идеале необходимо перебрать все возможные пути его вычисления, однако для систем, взаимодействующих с окружающей средой, эта задача невыполнима, так как таких путей может оказаться бесконечное множество. В настоящий момент для того, чтобы показать, что алгоритм соответствует накладываемым на него требованиям, используются методы верификации.

Среди существующих методов верификации для автоматизации тестирования АУ был выбран метод Model Checking. При верификации алгоритма подходом Model Checking проверяется, что некоторое свойство поведения алгоритма управления, выраженное формулой темпоральной логики, выполняется для модели системы с конечным числом состояний [3].

Так как язык Reflex базируется на модели конечного гиперавтомата, предоставляет логический параллелизм, средства взаимодействия между процессами и дает возможность контролировать время нахождения процесса в текущем состоянии, это делает Reflex удобным для описания верифицируемой модели системы, которая при верификации методом Model Checking представляется в виде модифицированного конечного автомата.

Итерационная схема разработки АУ была изменена (рис.3)

1) На языке Reflex создается описание части АУ, а также на формальном языке темпоральной логики описываются требования к АУ. Требования, описываемые на этом не этапе, не учитывают взаимодействие АУ с ВОУ – это те требования, которые рассматривают внутреннюю логику функционирования АУ, не зависящую от обмена данными с внешней средой.
2) Автоматическая верификация АУ (выполняется верифицирующим ПО). Если верфикатор выносит решение, что требования не выполняются, происходит поиск ошибок (или в описании АУ, или в формализации требований), после чего вносятся необходимые исправления и верификация повторяется

3) На языке Reflex создается описание части ВОУ, а также на формальном языке темпоральной логики описываются требования к ВОУ. Требования к ВОУ, описываемые на этом этапе, проверяют лишь то, что поведение ВОУ соответствует поведению реального объекта управления.
4) Автоматическая верификация ВОУ (выполняется верифицирующим ПО). Если верификатор выносит решение, что требования не выполняются, происходит поиск ошибок (или в описании АУ, или в формализации требований), после чего вносятся необходимые исправления и верификация повторяется
5) Формулировка требований на языке темпоральной логики, истинность которых зависит от взаимодействия АУ и ВОУ
6) Автоматическое построение общей модели АУ и ВОУ (выполняется верифицирующим ПО)
7) Автоматическая верификация общей модели АУ и ВОУ(выполняется верифицирующим ПО)
8) Если по результатам верификации было вынесено решение, что накладываемые требования не выполняются, происходит поиск ошибок (или в описании ВОУ, или в описании ВОУ, или в описании требований), после чего повторяется верификация общей модели. Если верификация прошла успешно, то АУ удовлетворяет накладываем требованиям. Происходит возврат на п. 1. и описание следующего логического блока алгоритма.

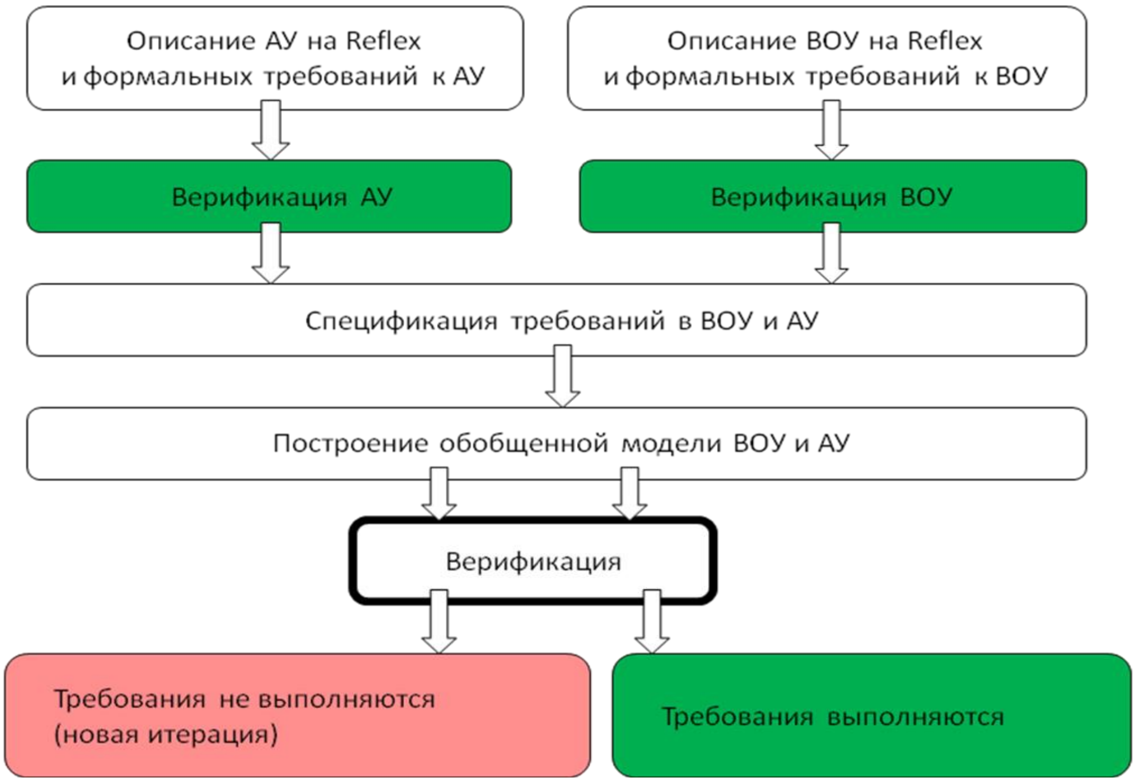
Описание АУ на Reflex
и формальных требований к АУ
Описание ВОУ на Reflex
и формальных требований к ВОУ
Верификация АУ
Верификация ВОУ
Спецификация требований в ВОУ и АУ
Построение обобщенной модели ВОУ и АУ
Верификация
Требования не выполняются
(новая итерация)
Требования выполняются

Рис. 3. Схема тестирования АУ и ВОУметодом Model checking

4.1. Схема верификации кода, созданного на языке Reflex, методом Model Checking с помощью верификатора SPIN

Для реализации описанной выше схемы требовалось создание верификатора кода на языке Reflex. Однако создание верификатора для метода Model Checking с нуля – задача крайне объемная, и требующая серьезных затрат. Поэтому было решено воспользоваться уже существующим верификатором Spin.

Верификация кода на языке Reflex проходит по следующей схеме:

1) Создается описание алгоритмического блока на языке Reflex
2) Транслятор автоматически преобразует код на языке Reflex в код на языке Promela – стандартном языке верификатора SPIN
3) Описание формальных требований методами верификатора SPIN
4) Трансляция требований в язык Promela с помощью верификатора SPIN
5) Автоматическая верификация верификатором SPIN
6) Требования выполняются – задача выполнена. Требования не выполняются – поиск ошибок или в описании алгоритма на языке Reflex, или в описании требований.

Таким образом, в предложенной схеме автоматически проходит верификация алгоритма, и для реализации необходимо было только реализовать транслятор из языка Reflex в язык Promela.

Такой подход позволяет избежать одного из самых значимых недостатков подхода Model checking: необходимости дальнейшего тестирования результирующего кода. Это связано с тем, что при верификации методом Model checking верифицируется не результирующий код, а построенная модель алгоритма. Однако верифицируемая модель, созданная на языке Reflex, транслируется в исполняемый код алгоритма на языке Си, который уже не требует дополнительного тестирования.

5. Заключение

Таким образом, в работе был предложен вариант реализации концепции итерационной разработки управляющих алгоритмов на основе виртуального объекта управления (ВОУ). Была разработана схема автоматической верификации алгоритма управления с помощью метода Model Checking. Так как описанный на языке Reflex алгоритм транслируется в исполняемый код алгоритма, это значит, что использование предложенной схемы позволит уйти от самого значительного недостатка верификации методом Model Checking: необходимости повторного тестирования, так как в классическом подходе верифицируется не сама конечная система, а только ее модель. Используя подход Model Checking можно проверять как корректное функционирование АУ, так и поведение ВОУ.

Концепция итерационной разработки управляющих алгоритмов на основе ВОУ эффективна для задач снижения рисков при вводе систем управления в эксплуатацию. Использование метода в реальных проектах по автоматизации позволяет:

1. тестировать создаваемые алгоритмы, начиная с самых ранних стадий разработки, внедрить итерационную модель разработки для случая промышленной автоматизации;
2. обеспечить контроль процесса создания управляющих алгоритмов и снизить психологическую нагрузку на коллектив разработчиков;
3. сократить время выполнения проекта и имеющиеся риски этапа пуско-наладки;
4. гибко расширять круг лиц, участвующих в процессе разработки, в частности, чтобы своевременно выявлять и устранять ошибки в техническом задании.

Разработанный подход был использован для отладки алгоритма управления вакуумной системой БСВТ.

Список литературы

1. Зюбин В. Е. Итерационная разработка управляющих алгоритмов на основе имитационного моделирования объекта управления // Автоматизация в промышленности. 2010. № 11. С. 43-48
2. Лях Т. В., Зюбин В. Е. Применение концепции виртуальных объектов управления для решения задач промышленной автоматизации // Материалы Девятой международной Ершовской конференции PSI-2014 (г. Санкт-Петербург, Россия, июнь 2014г). С. 57-64.
3. C. Baier, J.P. Katoen, Principles of Model Checking. The MIT Press. Massachusetts Institute of Technology, 2007.
4. Горячкин А. А., Зюбин В. Е., Лубков А. А. Разработка графического формализма для описания алгоритмов в процесс-ориентированном стиле // Вестн. Новосиб. гос. ун-та. Серия: Информационные технологии. 2013. Т. 11, вып. 2. С. 44–54.
5. Зюбин В. Е. К пятилетию стандарта IEC 1131-3. Итоги и прогнозы // Приборы и системы. Управление, контроль, диагностика. 1999. № 1.
6. В. Е. Зюбин. «Си с процессами» - язык программирования логических контроллеров // Мехатроника, автоматизация, управление. 2006. № 12 С. 31-35
7. Дж. Тревис. LabVIEW для всех. М.: ДМК Пресс, 2011. 912 с.

УДК 004.414.38

Developing formal temporal requirements to distributed program systems

Shoshmina I. V. (Peter the Great Saint-Petersburg Polytechnic University)

Developing temporal requirements to distributed program systems an engineer should determine and systemize event sequences caused by system processes interleaving. A number of such sequences grow exponentially that makes the requirement development procedure nontrivial. This is why engineers prefer not to construct or construct elementary formal requirements. As result powerful formal verification methods become unavailable or some important properties of distributed systems leaved unexpressed. While it is well-known, that development of formal requirement even without verification improves an quality of a distributed system structure and functions.

In this paper we suggest a method for formal temporal systems development which is easy-to-use. The method is based on scalable patterns of linear temporal logic formulas.

Using this method we developed formal temporal requirements to a practical program control system (a vehicle power supply control system). Verifying the requirements with the model checking method we found 3 critical errors that were missed by developers of the vehicle power supply control system during design and testing.

Keywords: software requirement specification, requirement patterns, model checking, linear temporal logic

1. Introduction

Developing temporal requirements to distributed asynchronous program system is complicated in practice. Because an engineer should systemize an exponential number of system behaviour sequences resulting as process interleaving.

The wide-spread approach to solve this problem is to use formulas patterns for requirements: an engineer tries to find a requirement close to a pattern. Dwyer et al. in [1] developed the *specification pattern system* (SPS). They analyzed 500 temporal requirements to systems from different application fields and suggested patterns for the most typical ones. The main SPS drawback it is too strict: patterns aren't scalable to different events number.

De-facto SPS has become the standard [2], [8], [9], [10]. Later it was modified by different way. In [2], [3] patterns were extended by real time and probabilistic requirements. In [4], [5] there were suggested nested patterns for interval logic, in [6] — nested patterns for linear temporal logic. In [2], [7] authors described patterns in limited English.

In spite of these modifications the strict structure of SPS formulas has left unchanged. In this research we develop patterns that, on one hand, could be scalable and, on another hand, we give an easy way how to use these patterns to develop significant temporal requirements. As a base for our patterns we use the temporal relation “leads-to” [11] where after an environment stimulus somewhen in the future there should follow a system reaction. Our patterns are formalised in the linear temporal logic (LTL).

Using our patterns we developed and verified formal temporal requirements to a power vessel supply control system (PVSS). The PVSS were developed and provided to us by a Russian ship control systems manufacturer. The original PVSS code was written on C++ and contained 20000 code strings (not counting external libraries). One of the most complicated PVSS characteristics was an asynchronous work of its modules. Verification allows us to find 3 critical errors that developers did not find neither during design nor during bench and program testing.

2. Patterns of events sequences

Temporal requirements of program systems are often some event sequences. The most suitable and concise temporal logic for describing event sequences is the linear temporal logic (LTL). LTL–formulas consist of atomic propositions $p \in AP$, Boolean operations and temporal operations: Until – $\mathcal{U}$ and the Next time – $\mathcal{X}$ (NextTime). This is grammar for a LTL–formula φ:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathcal{X}\varphi \mid \varphi\,\mathcal{U}\,\varphi \tag{1}$$

To short fomulas we will use some extra operations, Boolean ($\Rightarrow$, $\wedge$, etc.) and temporal ones: Future – $\mathcal{F}$, Globally – $\mathcal{G}$, Release – $\mathcal{R}$, Weak Until – $\mathcal{W}$, where $\mathcal{F}\,\varphi = \top U \varphi$; $\mathcal{G}\,\varphi = \neg\mathcal{F}\,\neg\varphi$, $\varphi\,\mathcal{R}\,\psi = \neg(\neg\varphi\,\mathcal{U}\,\neg\psi)$; $\varphi\,\mathcal{W}\;\;\psi = \mathcal{G}\,\varphi \vee \varphi\,\mathcal{U}\,\psi$, and the true constant: $\top = p \vee \neg p$. We use a common formal semantics of LTL formulas defined on infinite sequences (i. e. [12]).

A lot of formulas decribing different temporal requirements could be constructed with LTL. We consider one that has very practical application, when a system environment gives a stimulus by an event s and then the system guarantees an event–reaction p somewhen in the future (p *is after* s). We call the relation as the “unconditional response”. It’s a LTL formalization $Resp(s, p)$:

$$Resp(s, p) = s \Rightarrow \mathcal{F}\,p. \tag{2}$$

The requirement *"If someone from a floor calls an elevator then in the future the elevator will stop at that floor"* is an example of a requirement with the unconditional response. The response relation in the form (2) is well–known as "leads–to" [11].

Now we require that a system should remember receiving a stimulus s until emitting a reaction p by setting a condition t. So we get a "conditional response" relation denoted it as $Resp(s, p, t)$:

$$Resp(s, p, t) = s \Rightarrow t\,\mathcal{U}\,p. \tag{3}$$

Similarly we define a conditional precedence relation (*before an event–reaction p should be an event–stimulus s which sets a condition t*), denoted as $Prec(s, p, t)$:

$$Prec(s, p, t) = \neg p \Rightarrow (t\,\mathcal{U}\,p \Rightarrow \neg p\,\mathcal{U}\,s). \tag{4}$$

The requirement *"If a fire fighting system switched on then before that a duty officer gave its a corresponding command"* is an example of a requirement with the precedence relation.

Formulas (3) and (4) describe local temporal relations between a stimulus and a reaction in sense that a temporal relation is satisfied in a current state of a system behaviour. To develop requirements we should consider temporal relations (3) or (4) in all states of a system behaviour. Let's consider 4 typical systems work phases: *start*, *global*, *regular*, *final*. In a *global* phase a temporal relation should be satisfied in all system states. Other phases define a scope where a temporal relation is satisfied. In a *final* phase a temporal relation should be true after the final phase started; in a *start* phase — before the phase finished; in a *regular* phase a temporal relation should be satisfied during the phase. Defining phases bounds by events we get following LTL formulas for temporal requirements:

$$\begin{aligned}
global(s, p, t; \varphi) &= \mathcal{G}\,\varphi(s, p, t),\\
fin(s, p, t; q; \varphi) &= \mathcal{F}\,\mathcal{G}\,q \Rightarrow \mathcal{F}\,global(s, p, t; \varphi),\\
start(s, p, t; r; \varphi) &= \neg r \Rightarrow \varphi(s, p, t)\,\mathcal{W}\,r,\\
reg(s, p, t; q, r; \varphi) &= \mathcal{G}\,(q \Rightarrow start(s, p, t; r; \varphi)),
\end{aligned} \tag{5}$$

when φ is substituted by a formula *Resp* or *Prec* from (3)–(4), the formula *global* defines a requirement in a global phase, formulas fin, $start$ and reg — for final, start and regular phases respectively. The variable q defines an event of starting a final phase in fin, r — an event ending a start phase in $start$, and variables q and r – events starting and ending a regular phase respectively.

The suggested temporal relations (3)–(4) are so that they are easily scalable to stimuli and reactions consisting from event sequences (not from one event): $\vec{s} = \{s_1, s_2, \ldots, s_m\}$, $\vec{p} = \{p_1, p_2, \ldots, p_n\}$ with sequences of conditions $\vec{v} = \{v_1, v_2, \ldots, v_{m-1}\}$, $\vec{t} = \{t_1, t_2, \ldots, t_n\}$, restricting stimuli and reactions respectively:

$$\begin{aligned}
\mu(\vec{s},\vec{v}) &= s_1 \wedge v_2 \mathcal{U} (\ldots s_{m-1} \wedge (v_m \mathcal{U}\, s_m) \ldots), \\
\chi(\vec{p},\vec{t}) &= t_1 \mathcal{U} (p_1 \wedge \ldots t_{n-1} \mathcal{U} (p_{n-1} \wedge (t_n \mathcal{U}\, p_n)) \ldots), \\
Resp(\vec{s},\vec{p},\vec{v},\vec{t}) &= \mu(\vec{s},\vec{v}) \Rightarrow v_2 \mathcal{U} (s_2 \wedge \ldots (v_m \mathcal{U} (s_m \wedge \chi(\vec{p},\vec{t}))) \ldots), \\
Prec(\vec{s},\vec{p},\vec{v},\vec{t}) &= \neg p_1 \Rightarrow (\chi(\vec{p},\vec{t}) \Rightarrow \neg p_1 \mathcal{U}\, \mu(\vec{s},\vec{v})) \qquad (6)
\end{aligned}$$

In this case requirements expressed in LTL formulas (5) aren't changed except substituting φ by formulas $Resp$ or $Prec$ from (6) and adding $\vec{v}$.

If requirements depended on an infinite behaviour of environment we describe them by the following LTL formula:

$$\psi \Rightarrow \xi, \qquad (7)$$

when ξ is a formula from (5), ψ — a formula defining an infinite environment behavior. Common fairness requirements is a particular case of the formula (7).

Comparing patterns of [1] with ours ones by temporal relations structure we could resume that 83 formulas from 217 LTL-formulas of [1] have a response relation $Resp$, 13 formulas — a precedence relation $Prec$, while other 121 don't contain relations between a stimulus and a reaction. So our LTL–pattern coincide with existing practical requirements and even allow scaling them to represent wider temporal dependencies.

3. Developing requirements to the power vessel supply control system

The considered power vessel supply system (PVSS) consists of two power supply stations (PS) while a power supply station contains a diesel, a generator, a generator cutout switch (GCS). The power vessel supply control system coordinate the work of these PSs. Its structure is shown at the fig. 1 inside the bold frame. We will use the abbreviation PVSS for the power vessel supply system and for its control system. All PVSS controllers have independent asynchronous behaviors coordinated by passing messages. Environment modules/devices the PVSS works with are drawn outside the frame. The PVSS monitors and controls these devices

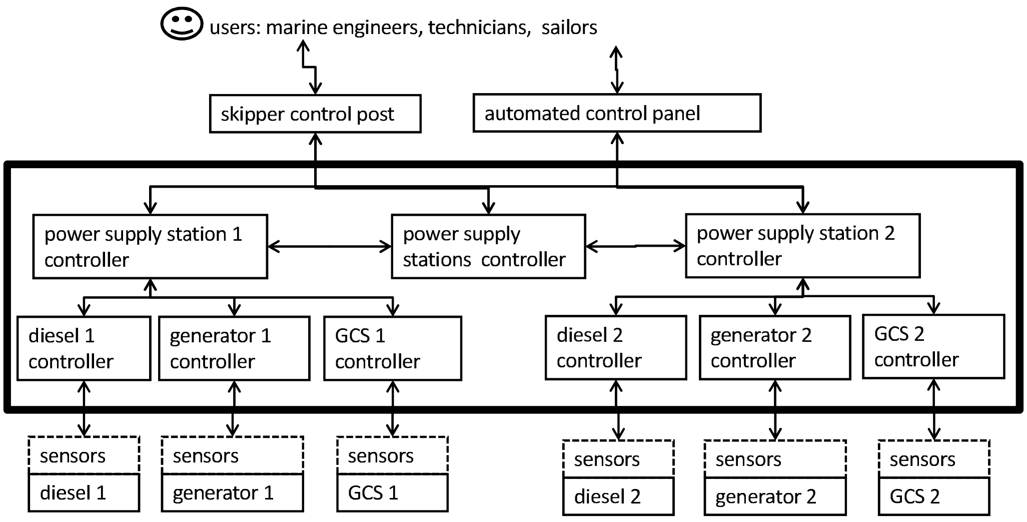

Рис. 1. The PVSS Structure

by reading sensors values and setting signals. A diesel has the utmost number of sensors (12 pieces).

The PVSS provides electricity power to all vessel consumers. For that it dynamically switches on/off power stations depending on loading. The PVSS activity could be quite complicated. For example, to start a power station the PVSS starts a diesel at first. When the diesel rotation becomes stable, PVSS starts a generator, after that it starts a generator cutout switch. And only after that consumers get the electricity power. Moreover, procedures of switching power stations on/off depend on PVSS modes and could be different.

“PSSV requirement specification” provided us by PVSS developers was written quite poor and did not contain enough information about PVSS to develop formal temporal requirements. So we used mainly “Bench testing program and technique”. The test from this manual is cited at the fig. 2. To resolve ambiguity and uncertainty we used “Operating guide’, the PVSS code too, and sometimes consulted with experts developed the PSSV.

At first we identified input/output events from tests in natural language (like at the fig. 2). We will use some of them in requirements below.

To combine events into temporal requirements sequences we will use the patterns (5)–(6). If someone would like to avoid the direct usage of formulas he/she could use the modified problem frame approach and translate requirements to formulas from graphical problem frames [13]. In general modified problem frames allow to construct temporal requirements unlike the original one developed by M. Jackson [14].

The test at the fig. 2 describes the PVSS transition to a remote automated mode. Analysing

D.1.2.1 Before start:

- DG1 and DG2 stopped (banners “DG1 is ready to start” and “DG2 is ready to start” lighten in the ACP window “Power Supply Station”);
- the SCP switch “PS mode” is in the position AUT;

D.1.2.2 Testing transition to the PS remote automated control mode.

D.1.2.2.1 Switch on the test bench “Testing PS control algorithms” buttons “DG1: ready to start”, “DG2: ready to start”, “DG1: remote control on”, “DG2: remote control on”, “SCP control”.

D.1.2.2.2 That should have the following effects:

- on the ACP display the message “PS control mode — remote” received and indicators “Control mode — remote”, “DG1 is ready to start”, “DG2 is ready to start” changed to yellow;
- on SCP lamps “DG1: SCP control”, “DG2: SCP control”, “DG1: ready to start”, “DG2: ready to start” lighted up.

D.1.2.3 Testing results are accepted when all effects described above happened.

Рис. 2. Testing transition to a PVSS remote automated mode. Abbreviations: DG – diesel generator, ACP – automated control panel, SCP – skipper control post, AUT – automated.

other tests of “Bench testing program and technique” we found out that the PVSS could transfer to the remote automated mode independently of diesels state. This is why the test at the fig. 2 splits to few temporal requirements, in particular: “Transition to a remote automated mode”, “Diesel 1 activation in the remote automated mode”, “Absence of a diesel 1 misactivation in the remote automated mode” and symmetrical for the diesel 2.

Transition to a remote automated mode. *Always when the PVSS is not in the remote automated mode and it would be in this mode in the future then before that an operator gives commands “DG1: remote control on”, “DG2: remote control on” on the ACP and changes the switch “PS mode” in the position AUT on the SCP.*

The requirement is written in LTL so:

$$\mathcal{G}\,(\neg dist \wedge \mathcal{F}\, dist \Rightarrow \neg dist\, \mathcal{U}\, autosig), \tag{8}$$

where *autosig* — the signal to set the remote automated mode (commands “DG1: remote control on”, “DG2: remote control on” and the switch “PS mode” in the position AUT), *dist* — the signal that the remote automated mode is set.

The other temporal requirement describes an absence of an unwanted diesel 1 activation.

Absence of a diesel 1 misactivation in the the remote automated mode. *Always in the remote automated mode the lamp “DG1: ready to start” wouldn’t light up until the diesel 1 is ready to start.*

The temporal relation in this requirement corresponds to the precedence pattern (6), so as result we get:

$$\begin{aligned}\mathcal{G}\,(dist \quad &\Rightarrow \quad ((\neg readylamp \wedge \neg hand\,\mathcal{U}\,readylamp) \Rightarrow \\ &\qquad \neg readylamp\,\mathcal{U}\,ready)\,\mathcal{W}\,hand),\end{aligned} \tag{9}$$

where *ready* — the signal from sensors that the diesel 1 is ready to start (simulated at the fig. 2 as the ACP banner "DG1 is ready to start"), *readylamp* — the lamp "DG1: ready to start" lights, $hand = \neg dist$ — manual or local modes is set, *dist* — as in (8).

The diesel 1 in the test at the fig. 2 activated (becomes ready to start) if the remote automated mode is set enough long. This is modelled in LTL as "somewhen forever".

The diesel 1 activation in the the remote automated mode. *If somewhen forever the remote automated mode is set up then somewhen forever the lamp "DG1: ready to start" would light up to the sensors signal that the diesel 1 is ready to start.*

Formally:

$$\mathcal{F}\,\mathcal{G}\,dist \Rightarrow \mathcal{F}\,\mathcal{G}\,(ready \Rightarrow \neg hand\,\mathcal{U}\,readylamp), \tag{10}$$

where $dist, hand, readylamp, ready$ — the same are in (9).

At the tab. 1 we compare our formal temporal requirements development to PVSS and "Bench testing program and technique". As result, we described more events explicitly than it was in an events table of "Bench testing program and technique". We found out requirements that unnecessary repeated in different tests. We defined requirements that were formulated implicitly, for example, the requirement (9) is implicit in the test at the fig. 2. So we resume that developing formal temporal requirements with the patterns (5)–(6) gives a better structure of requirement specification than informal procedures. But some requirements described by quite complicated LTL formulas containing 10-15 events.

4. Verifying the power vessel supply control system

We claim that our patterns allow to describe important requirements to distributed programs. To approve that we verified the PVSS with respect to developed formal temporal requirements using SPIN [15]. At first we constructed a PVSS model in Promela, the input language of SPIN. A PVSS module algorithm was modeled as an independent asynchronous process. Processes coordinated their work transferring messages by asynchronous channels.

Table 1

Comparing formal requirement development method and bench testing program on the PVSS

	Formal requirement development method	Testing program
Number of explicitly enumerated events	71	20
Number of requirements or tests	36 requirements	23 tests
Average size of a requirement or a test	10-15 subformulas (precedence relation), 30-36 subformulas (response relation)	2 pages (A4)
Development time	2 weeks	unknown

Because the PVSS model was large we reduced it manually. To check our temporal requirements we used 4 reduced models of the PVSS model. Correctness of reduced models is proved by correspondence of counterexamples traces in Promela with traces in the original C++ code.

Let's consider one of the critical error found out in the PVSS verification. Because this error obviously shows problems that developers of program systems meet with, and such errors are quite difficult to analyze and understand without verification.

Starting a reserve diesel–generator while another one crashed *If the power station 2 hardware failures infinitely often, and the power station 1 hardware works properly infinitely long, and always in case of failure of the power station 1 hardware the protection would be reset and the remote automated mode with the power station 2 priority is set, then somewhen in the future for every reserve response the diesel–generator 1 would start.*

Formally the requirement is so:

$$\bigwedge_i \mathcal{G}\,\mathcal{F}\,\tilde{b}_i \wedge \bigwedge \mathcal{F}\,\mathcal{G}\,\neg b_i \wedge \mathcal{F}\,\mathcal{G}\,\neg reset \wedge \bigwedge_j \mathcal{G}\,(b_j \Rightarrow \mathcal{F}\,reset) \wedge \mathcal{F}\,\mathcal{G}\,prior21 \quad \Rightarrow \\ \mathcal{F}\,\mathcal{G}\,(reserv \Rightarrow prior21\,\mathcal{U}\,lampon), \tag{11}$$

when $\tilde{b}_i$ – sensors data of the diesel–generator 2, b_i – sensors data of the diesel–generator 1, *reset* – reset a protection, *prior*21 – the remote automated mode with the power station 2 priority is set, *reserv* – the signal to starting a reserve diesel–generator, *lampon* – the lamp signalling the diesel–generator 1 started lights up. The requirement part "infinitely often" allow to model cases when hardware failures happen regular, but messages about these failures come with some delays.

SPIN found out a counterexample violated the formula (11) at the depth 29915. The requirement violated because the message which the generator cutout switch 1 controller sent to the power station 1 controller came with delay and blocked starting a reserve diesel–generator.

This error is impossible to detect while bench testing, because it's impossible to produce unknown quantity of hardware failures with unknown delay. And it's difficult to detect while program testing because it happens in a very seldom set of circumstances. But because of this error a vessel loses the electrical power control at all.

Interesting that developers observed such a behaviour in vessel sea trials, but they were sure that the error was caused by hardware (not by controllers coordination). So they tried to solve it by adding checks of the generator cutout switch data. And this obviously didn't help. Developers were not beginners: they specialize in power vessel supply control systems development. Except 23 bench tests they checked the PVSS with 841 program tests. But they didn't determine the error reason without the requirement formalization and verification.

During verification we detected about 141 errors. Most of them were minor and could be easily fixed, but 3 of them were critical. One of them were discussed above. Second one was about an uncontrollable start of a power station. As result of third critical error hardware could be under the electrical voltage in a PVSS protection mode.

To solve these critical problems it's required to change controllers algorithms for some modes and add few new modes more. This solution is time consuming, and takes about 80% of the PVSS time design. So we get the well-known consequence that using formal verification methods at first stages of a control program design could allow to avoid subtle, expensive errors at late design stages.

5. Conclusion

We suggested scalable LTL formulas patterns which describe many practical temporal requirements. We show that developing formal temporal requirements with them gives a well-structured requirement specification. The development allows to avoid redundant repeating of temporal requirements and to find out implicit requirements by organizing input-output events and their temporal relations.

Verifying the power vessel supply control system with developed requirements we found out three critical errors. These errors were not found developers by testing. The result of one critical error were observed by developers but they could not determine errors reasons correctly without the requirement formalization and verification. Fixing such critical and subtle errors at late stages of a control program design sometimes could be compared starting a program development from the scratch.

Список литературы

1. Dwyer M. B., Avrunin G. S., Corbett J. C. Patterns in property specifications for finite-state verification // ICSE '99: Proceedings of the 21st international conference on Software engineering. ACM. 1999. P. 411-420
2. Konrad S., Cheng B. H. C. Real-time specification patterns // Proceedings of the 27th international conference on Software engineering. ACM. 2005. P. 372-381
3. Grunske L. Specification patterns for probabilistic quality properties // Proceedings of the 30th international conference on Software engineering. ACM. 2008. P. 31-40
4. Mondragon O. A., Gates A. Q., Roach, S. M. Composite Propositions: Toward Support for Formal Specification of System Properties // Software Engineering Workshop. IEEE. 2002. P. 67-74
5. Mondragon O., Gates A. Q., Roach S. Prospec: Support for Elicitation and Formal Specification of Software Properties // Runtime Verification Workshop. ENTCS. Elsevier. 2004. V. 89. P. 67-88
6. Salamah S., Gates A. Q., Kreinovich V. Validated templates for specification of complex LTL formulas // Journal of Systems and Software. 2012. V. 85. P. 1915-1929
7. Smith R. L., Avrunin G. S., Clarke L. A., Osterweil L. J. Propel: an approach supporting property elucidation // 24th Intl. Conf. on Software Engineering. ACM Press. 2002. P. 11-21
8. Ramezani E., Fahland D., van Dongen B. F., van der Aalst W. M. P. Diagnostic Information for Compliance Checking of Temporal Compliance Requirements // CAiSE. 2013. P. 304-320
9. Post A., Menzel I., Podelski A. Applying restricted english grammar on automotive requirements: does it work? A case study // 17th international working conference on Requirements engineering: foundation for software quality. Springer-Verlag. 2011. V. 6606. P. 166-180
10. Yu J., Manh T. P., Han J., Jin Y., Han Y., Wang J. Pattern Based Property Specification and Verification for Service Composition // 7th International Conference on Web Information Systems Engineering (WISE). Springer-Verlag. 2006. V. 4255. P. 156-168
11. Pnueli A. The temporal logic of programs // 18th Annyv. Symp. on Foundation of Computer Science. IEEE Computer Society. 1977. P. 46-57
12. Karpov Yu.G. Model Checking. Parallel and distributed program systems verification // SPb:BHV-Petersburg. 2010. 560 p. (in Russian)
13. Shoshmina I.V. A method eliciting context requirements to logical control program systems // Information and Control Systems. 2014. №3, P. 68–77 (in Russian)

14. Jackson M. A. Problem Analysis Using Small Problem Frames // South African Computer Journal. 1999. V. 22. P. 47-60
15. Holzmann G. The Spin Model Checker // Primer and Reference Manual Addison Wesley. 2003

UDC 519.713

Using BALM-II for deriving parallel composition of timed finite state machines with outputs delays and timeouts: work-in-progress

Shabaldina N. (Tomsk State University),

Gromov M.(Tomsk State University)

In this paper we consider a procedure of parallel composition construction of Timed Finite State Machines (TFSMs) using BALM-II and suggest different ways of getting linear functions that describe a set of output delays. Our research consists of three steps: at first step we consider composition of TFSMs when an output delay may be a natural number or zero; at second – we add transitions under timeouts; at third we consider composition of TFSMs in general case (when output delays are described as sets of linear functions). This paper is devoted only to the first step of the research.

Keywords: *Timed finite state machines, parallel composition, BALM-II.*

1. Introduction

Most modern applications, such as web-services, telecommunication protocols, are oriented on interaction with each other. The classical model for a discrete system is Finite State Machine (FSM). If the behavior of each system is described by an FSM, then their common work can be described by their composition (that also will be an FSM under appropriate assumptions) [1,2]. In this work we are interested in so-called parallel composition [1], when the interacting systems work asynchronously in as-sumption of slow environment, and for deriving such FSM composition there is a tool named BALM-II (Berkeley Automata and Language Manipulation)[2].

Sometimes it is necessary to take into account time aspects of a discrete system. Probably the most general way to describe such a system is Timed Automaton [3]. However, in this work we are interested in input-output reactive systems, when every input action is necessary followed by output action, probably after some time. The class of such systems has been already mentioned, it includes telecommunication protocols, sequential circuits, web-services etc. In this case we can use Timed Finite State Machine (TFSM) as a model. There exist different ways to introduce Timed FSM, for example, with timed guards on transitions [4]. In this work we consider TFSM with output delays and timeouts [5,6]. We got inspiration for our research from the work [5], in which authors describe how to build parallel binary composition of two timed FSMs with output delays and timeouts. In order to derive the composition of timed FSMs the corresponding automaton should be built [5]. First we transform both TFSMs into automaton, then we compose them, and then we need to go back to the TFSM model. In [5] it is shown that composition of two Timed FSMs can have infinite number of output delays for a given transition and those delays can be de-scribed by a finite set of linear functions $\{ b + k\cdot t \mid b, k \in \{0\} \cup \mathbb{N} \}$.

There are several tools dealing with timed automata, their composition and verification. One of the most popular is UPPAAL [7]. It allows to describe timed system using Extended Timed Automata, as well as composition of such systems. One of the key feature of UPPAAL is built-in verifier. Unfortunately, UPPAAL does not build composition explicitly and one of the objectives for this work is to get composition explicitly for further processing (for example, test generation). For that reason in this work we decided to use BALM-II since it was designed to build parallel binary com-position of two FSMs. To be able to use this tool for Timed FSMs we use well-known transformation of TFSM into FSM and then into common automaton by introducing new (tick) action. Also we suggest two approaches for extracting functions $f(t) = b + k\cdot t$ from the composition of corresponding automata in order to derive TFSM. First approach is based on using BALM-II once again. And the second one is to find corresponding loops in the transition graph of the automaton composition.

This work is partially supported by the basic part of the State Assignment of the Ministry of Education and Science of the Russian Federation (Project code No. 1975) and by the grant of Russian Fund for Basic Research No. 15-58-46013 CT_a.

2. Preliminaries

An automaton S is a 5-tuple $(S, X, s_0, F, \lambda_S)$, where S is a finite nonempty set of states with s_0 as the initial state and $F \subseteq S$ as a set of final (accepting) states; X is an alphabet of actions; and $\lambda_S \subseteq S\times X\times S$ is a transition relation. The transition relation defines all possible transitions of the automaton. The language L_S of automaton S is the set of all sequences α in alphabet X, such that in automaton S there is a sequence of transitions (marked by α) from the initial state to some final state. An FSM S is a 5-tuple $(S, I, O, s_0, \lambda_S)$, where S is a finite nonempty set of states with s_0 as the initial state; I and O are input and output alphabets; and $\lambda_S \subseteq S\times I\times O\times S$ is a transition relation. In FSM all states are final.

Let $\mathbb{N}$ be the set of natural numbers. TFSM [5] is an FSM with timeouts and output delays $S = (S, I, O, s_0, \lambda_S, \Delta_S, \sigma_S)$, where 5-tuple $(S, I, O, s_0, \lambda_S,)$ is underlying FSM, Δ_S: $S \rightarrow S \times (\mathbb{N} \cup\{\infty\})$

is a timeout function that determine maximal time of waiting for input symbol, σ_S: $\lambda_S \rightarrow (\{0\} \cup \mathbb{N})$ is an output delay function that determine for each transition time delay for producing output (output timeout).

Parallel composition describes a dialog between two components. The structure of the composition is represented in Figure 1.

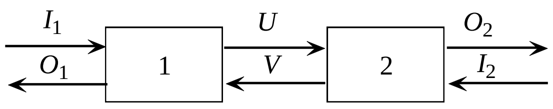

Fig. 1. Structure of binary parallel composition

We suppose that we have "slow environment" (it means that the next input can be applied to the composition only after it produces external output to the previous input), the alphabets of different channels don't intersect and there are no infinite dialogs under internal inputs (it means no livelocks). We also suppose that each component and the whole composition have timed variables. The values of these variables are increasing synchronically, and they reset when the system gets an input or when the state is changed.

3. Deriving an automata based on the given TFSMs

In order to derive the composition of timed FSMs we can use the corresponding automaton [5]. For deriving an automaton that corresponds to the classical FSM we need to do the following steps [2]:

1. Derive the set of states that contains all FSM states (final, or accepting states) and a number of intermediate (not-final) states (one new state for each transition in FSM). The initial state of the automaton is the same as the initial state of the FSM.
2. Derive the set of actions $X = I \cup O$.
3. Derive the set of transitions: for each FSM transition we add two transitions in automaton, i.e. (s, i, o, s') generates $\{(s, i, s''), (s'', o, s')\}$, where s'' is one of the intermediate states we added at the first step which corresponds to the transition under consideration.

So, in order to construct an FSM from the given automaton, we need to split alphabet of actions into input alphabet and output alphabet, merge transitions and delete intermediate states.

In order to derive an automaton for the timed FSM with timeouts and output delays we first apply steps that are described above. Then we need to add into the set of actions a new special symbol $1 \notin I \cup O$ that corresponds to tick count and represents an action "to wait for one time unit"[5]. We add in each final state a loop under 1 (in order to describe the situation that the current component is waiting for input and time variable of the other component is increasing). Then, we replace the transition under timeouts by a chain of transitions under 1 (in order to model the time delay), the length of the chain corresponds to the value of time delay. And we do almost the same by adding the chain of transitions under 1 between an input and output symbols (if there is an output delay for this transition in TFSM) [5].

In Figures 2 and 3 one can see TFSMs that describe the behavior of left and right components of the composition, correspondingly. We take this example from work [5]. In Figures 4 and 5 we show automata for these TFSMs. In this example the structure of the composition is simpler then in Figure 1 (left component has external input Request and external output Deliver; right component has no external inputs and external outputs) and also TFSMs are simpler: they have output delays, however, they have no transitions under timeouts.

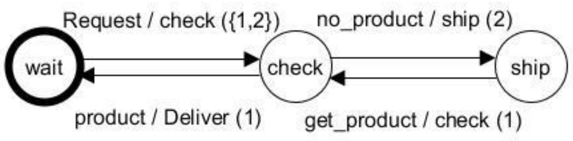

Fig. 2. Left-part component (TFSM)

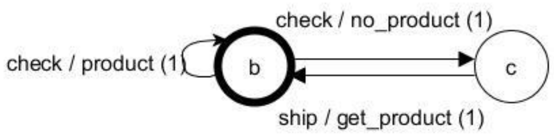

Fig. 3. Right-part component (TFSM)

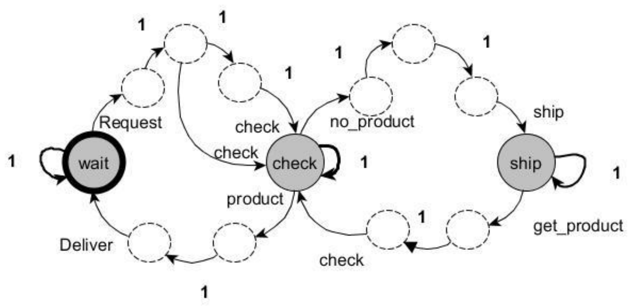

Fig. 4. Left-part component (automaton)

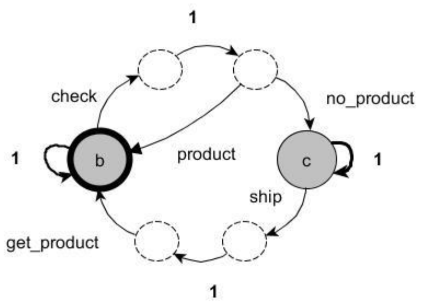

Fig. 5. Right-part component (automaton)

4. Deriving automata parallel composition using BALM-II

In this section we describe how to derive a binary parallel composition of two automata using BALM-II, and we illustrate this procedure using our example from previous Section.

BALM-II supports `AUT` file format for describing automata [2]. This format is a restricted form of `BLIF_MV` format. Due to the restriction of space we just mention the most important things. First of all, we need to determine our channels, in AUT format it will be like this for the left component:

```
.inputs x v u y t E
```

We underline that in addition to the channels of the left component that you can see in Figure 1, we need to mention the special time channel (channel *t*) that correspond to the timed variable (or to our special action 1). As for the channel *E*, this is also a special channel that determine which one from the channels *x*, *v*, *u*, *y* and *t* is active now (while the other channels are inactive).

For the time channel *t* we need to introduce in addition to the input 1 one more input (due to the fact that we need at least two values for the channel alphabet in BALM-II):

```
.mv t 2 1 none
```

When we have our automata in `AUT` format, the first thing we need to do is synchronizing channels of the composing automata:

```
chan_sync x|v|u|y|t|E u|v|t|E left_timed.aut right_timed.aut left_t_sync.aut right_t_sync.aut
```

Then, according to the algorithm of deriving the composition of two automata [1,2], we need to extend the alphabet of the right-component automaton to the channels *X* and *Y*:

```
expansion E0,E3 right_t_sync.aut right_t_exp_aut
support x,v(3),u,y,t,E(5) right_t_exp.aut right_t_support.aut
```

The next step is deriving an intersection of two automata:

```
product left_t_sync.aut right_t_support.aut product_timed.aut
```

Now we have an automaton that describes common behavior of left and right components, but its behavior does not always correspond to our “slow environment” restriction, and in this case we need to intersect derived automaton with the automaton that represent the language $(X(UT^*V)^*T^*Y)^*$. In our example we don’t need to do this. So the next step is to restrict the automaton to external channels and special timed channel:

```
restriction E0,E3,E4 product_timed.aut restriction_timed.aut
support x,y,t,E(3) restriction_timed.aut comp_timed.aut
```

The result is shown in Figure 6 (a). One can see that after Request there can be output Deliver after $3 + 5t$ or $4 + 5t$ tick counts, where *t* is arbitrary non-negative integer number. So in Figure 6 (b) you can see corresponding TFSM.

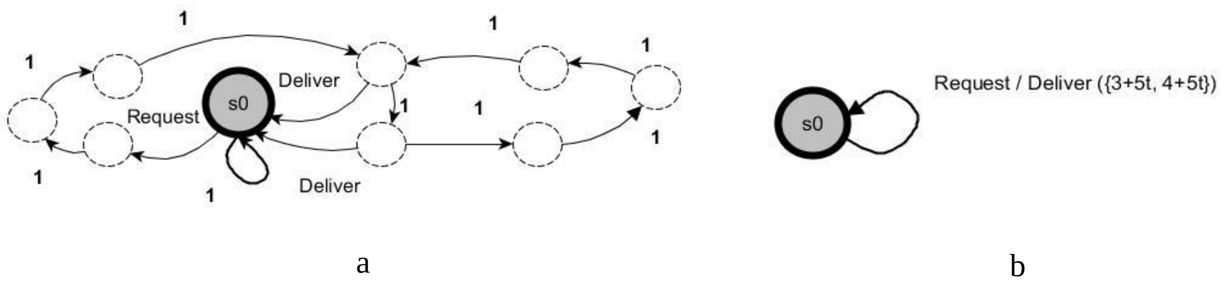

Fig. 5. The composed automaton (a) and Timed FSM (b)

5. Deriving parallel composition of two TFSMs with output delays: two approaches for extracting output delays functions

In this section we propose two approaches for extracting a set of linear functions from the derived automaton.

5.1. Using BALM-II for extracting output delays functions

The idea of this approach is to intersect consequently the resulting automaton with the automata that correspond to the languages $X1^b(1^k)^*Y$, i.e., the languages with the following property: they contain sequences that start with any external input symbol, the end of the sequences is any external output symbol, and between these input and output there is subsequence that corresponds to the function $\{b + k\,t \mid b, k \in \{0,1,..,n\}$.

We need to mention that in this case we need to intersect not only the composition automaton, but also its modifications that can be derived by making each accepting state as an initial state (one by one). So we fix b and k and intersect automaton with the language $X1^{b+k}(1^k)^*Y$ with the composition, fixing in the composition automaton an initial state (we consider the automaton with the language $X1^{b+k}(1^k)^*Y$ instead of the language $X1^b(1^k)^*Y$ in order to avoid the case when in the composition automaton there is a chain that corresponds to 1^b and then no loop, i.e. the case when t can only be equal to zero). Then we test the intersection using `check_nb` BALM-II command. This command allows answering the question: whether we can extract output delays function for the fixed b and k or not. If in the composition automata between input and output there is a subsequence that corresponds to the function $b + k\cdot t$, then the corresponding intersection will be nonblocking, it means it has no deadlocks; otherwise, it will be blocking, so, the intersection will contain no external output after some sequence under 1. For our example the intersection (product) of composition automaton and the automaton with the language $X1^3(1^5)^*Y$ will be nonblocking, the intersection with the automaton with the language, for example, $X1^2(1^5)^*Y$ will be blocking.

5.2. Getting output delays procedure based on analyzing cycles in automaton

Let us notice some properties of automata, derived from TFSMs:

1. Every transition, marked with input symbol, starts at final state and ends at non-final state.
2. Every transition, marked with output symbol, starts at non-final state and ends at final state.
3. Every transition, marked with 1 (a tick count), starts at non-final state and ends at non-final state.
4. If there are several non-final states s_1, …, s_k, such that $(s_i, 1, s_{i+1}) \in \lambda_S$, $i = 1, \ldots, k-1$ (continuous non-final chain of transitions marked by 1), then $s_i \neq s_j$, for every i and j, $i \neq j$ (there are no time loops, Figure 7 (a)).

However, as it was shown with the example in previous Sections, when we have parallel composition of two TFSMs, the resulting automaton may have continuous non-final time chain with a loop (Figure 7 (b)). Nevertheless, there cannot be intermediate time loops, i.e. loops with outgoing edge that is marked by 1 (Figure 7(d)) or several (Figure 7(e)) time loops. We shall prove this by the following proposition.

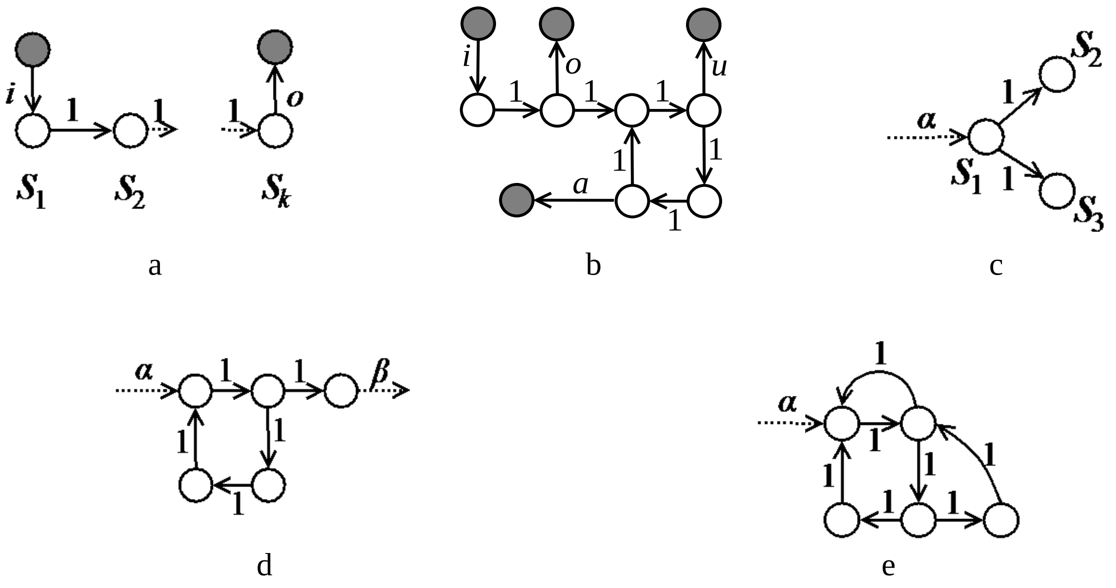

Fig. 7. Time chains. Here *i* – input symbol, *o*, *u*, *a* – output symbols, final states marked gray and non-final are blank

Proposition 1. Given automaton *A*, describing parallel compositions of two TFSMs *P* and *Q*. There are no states with more than one outgoing transition, marked by 1.

Proof. *Indeed, suppose there is such a state (Figure 7(c)), reachable by sequence* α*. It means, that by construction in automaton* A_P *there is state p reachable by* α *and automaton* A_Q *there is state q reachable by* α *as well, such that either p has two different outgoing transitions marked by*

1, or q has two different outgoing transitions marked by 1, or both of them have such transitions. Neither of listed is possible. □

Corollary 1.1. There cannot be intermediate time loop in any continuous time chain of an automaton, describing TFSM parallel composition.

Corollary 1.2. There cannot be more than one time loop in in any continuous time chain of an automaton, describing TFSM parallel composition.

Corollary 1.3. There cannot be more than one state in continuous time chain with more than one ingoing transitions, and the number of ingoing transitions is not more than two (Figure 7(b)).

Now we describe a procedure for counting output delays. In this procedure we shall use sets Q_{siop}. Each set Q_{siop} contains functions (constant or linear) of output delays for transition (*s*, *i*, *o*, *p*). We notice that the estimation of the procedure is ~*N*, where *N* is the number of states in automaton *A*, describing parallel composition of two TFSMs.

Procedure 1. Getting output delays.

Input. Automaton *A*, describing parallel composition of two TFSMs.

Output. Set of sets of output delays for every input-output pair possible in composition.

```
1. Get next final state s of automaton A. IF they are over, THEN
   END.
2. Get next outgoing transitions of s. IF they are over, THEN
   GOTO Step 1. Let outgoing transition be marked with input
   symbol i, and the next state of the transition be s1.
3. scurr := s1; b := 0; k := 0.
4. IF scurr has more than one ingoing transition THEN
   k := countLoopLength(A, scurr) (Procedure 2).
5. FOR every transition (scurr, o, p) ∈ $\lambda_S$, where $\lambda_S$ is
   transition relation of A, o is output symbol, and p is final
   state of A, DO put function b + k*t in Qsiop.
6. IF there is transition (scurr, 1, s′) ∈ $\lambda_S$, THEN scur := s′.
7. GOTO Step 2.
```

Procedure 2. countLoopLength

Input. Automaton *A* and non-final state *s* of *A*.

Output. Length of a loop, containing *s*, or *N* + 1, if there is no such loop, where *N* – number of all states in *A*.

```
1. $s_{curr}$ := $s$; $k$ := 0.
2. IF there exists transition ($s_{curr}$, 1, $s'$) ∈ $\lambda_S$, THEN $s_{curr}$ := $s'$,
   $k$ := $k$ + 1.
   ELSE RETURN $N$ + 1. END.
3. IF $s_{curr}$ == $s$, THEN RETURN $k$, END.
```

6. Conclusion and Future Research Work

In this paper, we consider the procedure of parallel composition construction of TFSMs using BALM-II and investigate different ways of extraction the set of linear functions (that describe an infinite set of output delays) from the composition of corresponding automata. This is work in progress, so we represent here just the first step of our investigation, considering only the case of deriving the composition of TFSMs with output delays that are natural number or zero. We suggest two approaches for getting output delays from the composition of corresponding automata: first deals with BALM-II once again, and the second is based on analyzing time loops in automaton. In our future work we'll consider the composition of TFSMs with transitions under timeouts and the composition of TFSMs when the output delays are infinite and represented by the set of linear functions; this can happen in cascade composition.

References

1. N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli. Solution of parallel language equations for logic synthesis // In The Proceedings of the International Conference on Computer-Aided Design. 2001. P. 103–110.
2. G. Castagnetti, M. Piccolo, T. Villa, N. Yevtushenko, A. Mishchenko, Robert K. Brayton. Solving Parallel Equations with BALM-II // Technical Report No. UCB/EECS-2012-181, Electrical Engineering and Computer Sciences University of California at Berkeley. 2012. [Electronic resource] http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-181.pdf (date of access: 21.04.2016).
3. R. Alur and D. L. Dill. A theory of timed automata // Theoretical computer science. 1994. Vol.126, Iss. 2. P. 183–235.
4. K. El-Fakih, M. Gromov, N. Shabaldina, N. Yevtushenko. Distinguishing Experiments for Timed Non-Deterministic Finite State Machines // Acta Cybernetica. 2013. Vol. 21, № 2. P. 205–222.
5. O. Kondratyeva, N. Yevtushenko, and A. Cavalli. Parallel composition of nondeterministic finite state machines with timeouts // Journal of Control and Computer Science. Tomsk State University, Russia. 2014. Vol. 2(27). P. 73–81.
6. O. Kondratyeva, N. Yevtushenko, A. Cavalli. Solving parallel equations for Finite State Machines with Timeouts // Trudy ISP RAN [The Proceedings of ISP RAS]. 2014. Vol. 26, Iss. 6. P. 85–98.

7. http://www.uppaal.com/

UDC 004.423.4+004.415.5

The formalism for semantics specification of software libraries

V. Itsykson (Peter the Great St. Petersburg Polytechnic University)

The paper is dedicated to the specification of the structure and the behavior of software libraries. It describes the existing problems of libraries specifications. A brief overview of the research field concerned with formalizing the specification of libraries and library functions is presented. The requirements imposed on the formalism designed are established; the formalism based on these requirements allows specifying all the properties of the libraries needed for automating several classes of problems: detection of defects in the software, migration of applications into a new environment, generation of software documentation. The conclusion defines potential directions for further research.

Keywords: formal specification, software library, behavioral description, software defect.

1. Introduction

Software libraries have become the de facto standard for implementing the component-oriented approach in which the software maker encapsulates specific functionality as a set of functions, data types and an application user interface. Modern libraries are extremely complex objects whose functionality is often considerably more sophisticated than that of the applications using them.

The key difference between libraries and standard applications is the manner in which they are used. Applications are used by users who follow instructions, operating manuals and built-in help systems, and have no need for formal specifications describing the applications. Libraries, on the other hand, are mainly used by other programmers who, in order to integrate the functionality of applications and libraries, need to clearly understand how a library works, how it can be used, how it affects the application, which changes are introduced to it from version to version, etc.

How does the library developer typically specify the library? One or several of the following methods are commonly used:

- headers with comments;
- verbal description of the library interface;
- verbal description of the behavior of individual functions;

- verbal description of some allowed sequences of function calls;
- examples provided by the developer.

However, none of these methods solves the problems of the formal specification of the library semantics. The library semantics consists of two components: the *semantics of individual functions* and of the allowed ways of joint use of library functions. The semantics of individual functions is determined by the function call conditions, the obtained results, the side effects, and the impact on the environment. Typically, the semantics of functions is described informally in the form of text descriptions. The *allowed ways* of joint use of library functions are at best described by the authors informally in the documentation accompanying the library,

In other words, the software engineering industry is currently lacking a set of tools for formalized description of the semantics of software libraries.

Since there is no formal specification for the libraries, it is, at present, impossible to satisfactorily solve several classes of problems:

- automatic verification of whether an application is correctly using a library. Here the term ‘correctly’ implies that the application accesses the library with satisfy a protocol specified by the designer[1]
- detection of programming errors in multi-file projects using third-party libraries when the source code is unavailable
- analysis of the compatibility between the applications and the new version of the library
- porting applications into a new library environment

Thus, the goal of this paper is to develop a formalism allowing to rigorously describe all the necessary aspects of libraries.

2. State of the Art

The specification of libraries and services has been long studied; a sufficient number of publications offer different approaches to describing the specifications. The first studies were related primarily to providing interoperability, with the main goal of the specifications created in designing the self-contained description of the interfaces of libraries and services that could be then used in different programming languages and operating systems. Examples of such specifications include the IDL language [1], as well as many of expansions, such as MIDL [2], and OMG IDL [3]. The main limitation of these languages for library specification is in the detailed API description

[1] Currently, this problem is typically solved dynamically at runtime by analyzing the return codes of library functions, or by exception handling.

without focusing on valid options for using the libraries. This means that the emphasis is on describing the signatures of functions and data types, while not enough attention is paid to the semantics specification of the entire library.

One of the first studies in the field of component interface specification is the work by Allen and Garlan [4], in which the authors reduce the problem of the interaction between the components of a software system to the specification of interaction protocols similar to computer networking protocols. The theory on communicating sequential processes (CSP), developed by Hoare [5], was taken as the basis for the formalism, and then altered in an appropriate manner. The introduction of special elements, such as ports, connectors and roles, into the formalism allowed separately specifying various aspects of the potential interaction between the components. Using the formalism can partially solve the problem of component compatibility with the help of the FDR model checker [6].

Alfaro and Henzinger describe in [7] their own version of the formalism for describing the interacting components, called the interface automata. The study uses an optimistic definition of component compatibility, based on the use of the environment model. The authors propose formal methods for verifying the optimistic compatibility of two interface automata.

Some studies are focused on the mechanisms of automated construction of specifications of interfaces and libraries based on analyzing the existing software. For example, the authors of [8] propose an approach to library specification inference based on static predicate mining. The authors use data flow and control flow analyzes for collecting predicates characterizing the interface functions. Another approach is described in [9], where the authors offer using dynamic output of library specifications based on unit testing. For this purpose, library functions, interfaces, data types and transactions are defined in terms of the Datalog formalism. Valid sequences of function calls are specified through special predicates. Specifications inference is based on analyzing and generalizing the results of random unit testing of the library’s functions.

One of the most interesting approaches to describing library APIs and their application rules is the SLAM approach proposed by Microsoft Research for driver verification. SLAM uses the SLIC language [10] for specifying the libraries and the rules of interaction between the programs and the API. The SLIC specification is used for the instrumentation of the program and/or the library for further dynamic or static compliance control. The lack of semantic descriptions for the library back-ends prevents SLAM from being used for automated migration.

In his paper ‘The future of library specification’ [11], Leavens describes several indirect approaches in addition to the known ones associated with informal documentation and formal specification; these are specification through example uses, specification through library source codes and specification through unit tests. The main conclusion reached by the author is that library specification must combine all of these approaches.

In our previous studies [12, 13], we also proposed a formalism for library specification and a language supporting the description of such specifications. However, the options for using the libraries (i.e., the behavior) are described implicitly within this approach, and the language does not allow defining function contracts and the influence of the functions on the environment to the full extent.

Thus, at present, there is no universal approach to library specification that would allow to:

- describe the external interface of the library in detail;
- define the potential protocols for using the library;
- specify the side effects of the library, i.e., its influence on the environment;
- explicitly introduce semantic descriptions of library behavior.

3. Library Organization Specifics

The specifics of using libraries is that a library is not just a purely functional object; it can possess an internal state and various side effects that significantly affect the opportunities for calling individual functions.

Let us introduce a classification of function libraries in terms of their internal state.

1. Libraries without an internal state
2. Libraries with the internal state of the library
3. Library with the internal state of the object created
4. Combined libraries

The first class comprises libraries containing pure functions without side effects. These include, for example, libraries of the mathematical functions of the standard C language library (math.h).

The second class includes libraries that preserve their state, that is to say, the behavior of individual functions depends on the state of the library. An example of such a library is the part of the stdlib library providing random number generation. Calling srand() sets the initial value of the generator, while rand() returns the next random number in the sequence constructed on the basis of the initial value.

The third class consists of libraries that preserve context within an object created by the library's functions. Such an object may be, for example, a newly created socket or a file descriptor. In the first case, the context contains the parameters of the socket (IP-addresses, port numbers, state), while in the second case, the context contains the file parameters, the opening mode and the next read data pointer.

The fourth class is the most general, containing libraries that combine the features of the second and the third classes.

4. Formal Specification of Libraries

A full formal specification of libraries should describe:

- a signature of all functions making up the library;
- a contract for each library function (preconditions, postconditions, the influence on the environment, etc);
- a behavioral model of the library taking into account all possible options for using the library’s functions and specifying, in particular, the behavior of the library in case of invalid use;

Based on the above, let us define the full specification of libraries as <F, L>, where

- F = {F_i} is the set of library functions;
- L is the behavioral description of the library.

An individual library function, Fi, is defined as <Name, Arg, Res, Pre, Post, A, CondA, D, CondD>, where

- Name is the name of the function;
- Arg is the set of the formal arguments of the function;
- Res is the result of the function;
- Pre are the preconditions of the function expressed by the formula in the first-order logic of the arguments Arg and Res;
- Post are the postconditions of the function expressed by the formula in the first-order logic of the arguments Arg and Res;
- A is the set of semantic actions[2] performed by the function;
- CondA is the set of conditions for semantic actions to be performed. An action Ai is performed during the execution of a function if the expression $CondA_i$ is true.
- D is the set of launched child state machines;
- CondD is the set of launch conditions for child state machines. A machine Di is launched during the execution of a function if the expression $CondD_i$ is true.

[2] Semantic actions are an abstraction for describing significant behavioral elements [16]

Let us represent the behavioral description of the library by a of set of parameterized extended finite-state machines (EFSM): L = {L, S1(q,P), …, Sn(q,P) ()}, where

- L is the main extended finite-state machine describing the behavior of the entire library;
- Si is an ith child EFSM launched if certain conditions are fulfilled;
- the parameter q is the initial state of the child finite-state machine;
- P is the optional parameter of the child finite-state machine

The state of the main state machine corresponds to the state of the library, and the state of the child ones corresponds to the state of the objects created. The stimuli forcing the machine to pass from one state to another are the calls of library's API functions.

An individual machine is defined as a modified EFSM <Q, Q_0, X, V, C, T>, where

- Q is the set of control states of the machine (the states of the library objects);
- Q_0 is the non-empty set of initial states of the machine. Several initial states can exist for child state machines, since initial conditions may be different when an instance of the machine is created;
- X is the set of finish states. Child machines are destroyed after reaching these states;
- V is the set of internal variables of the machine;
- C is the set of function calls acting as stimuli, C_i is the call of an ith function; $C_i \in F$;
- C_i^A is the set of semantic actions initiated by the function launch when C_i^{CondA} is true;
- C_i^D is the set of child state machines launched by the function when that C_i^{CondD} is true;
- T is the transition relation.

Due to limitations of space, the formalism is presented without going into too much detail. Such issues as the specification of invalid behavior, the default actions, the data types, etc., have been left outside the scope of our investigation. These issues will be discussed in more depth in other studies.

Actually, from a developer's perspective, the behavioral description of libraries is better represented in graphical form rather than from the standpoint of set theory.

Fig. 1 shows an example of graphically describing the client side of the TCP-socket library. The solid line indicates the transitions of the machine, and the dashed line indicates the launches of the child machines; the finish states are highlighted in red. The machine "L" describes the overall behavior of the bsd-socket library, which, in contrast to WinSock, does not require initialization. A side effect of calling socket() is the creation of a new machine "P", corresponding to the newly created socket, with its own life cycle. It should be noted that several machines can be created, differing only in the launch parameter of the child machine (an element corresponding to calling socket()).

Fig. 2 presents a more complex example, showing a graphic model of the server side of the TCP protocol of the bsd-socket library. In addition to the top-level machine corresponding to the library ("L"), the figure shows two families of machines: the first ("P") encapsulates the properties of the listening sockets, and the second one ("S") those of the server sockets created.

Both examples demonstrate only the behavioral description of libraries, without specifying a set of functions.

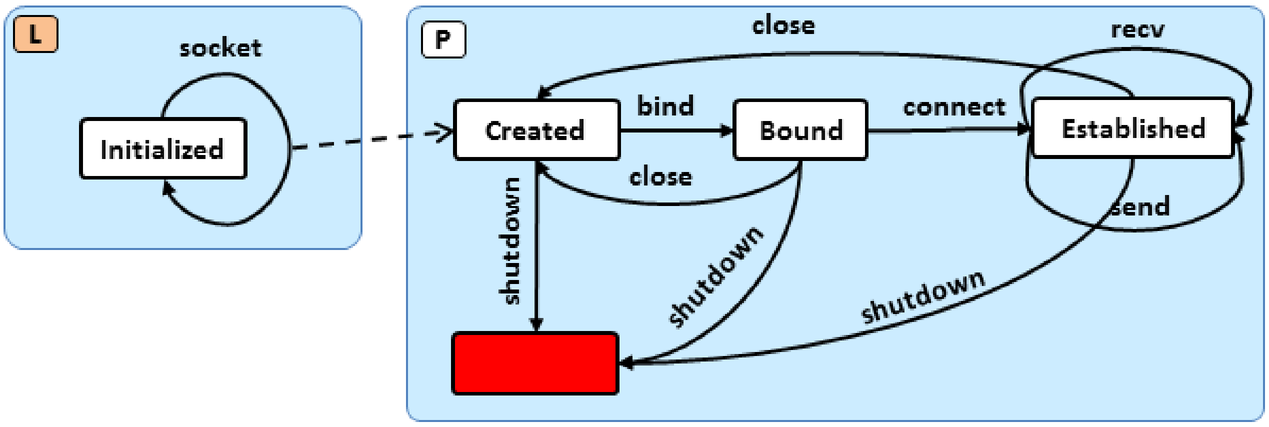

Рис. 1. An example of a simple machine corresponding to the client side of the TCP protocol of the bsd-socket library

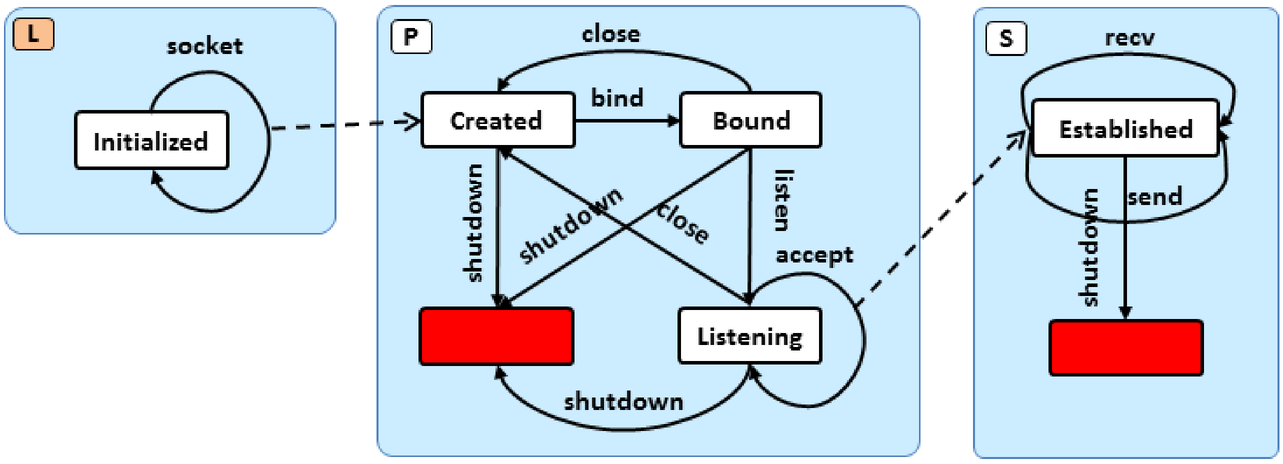

Рис. 2. 2. Example of a machine describing the server side of the TCP protocol of the bsd-socket library

Obviously, a library developer requires convenient tools for defining the formalisms introduced. We propose using special language for describing the sets of library functions, and an object-oriented graphical editor for the behavioral description of the library.

5. Prospects of Using the Developed Formalism

5.1 Constructing Specifications

Formal specifications could be constructed in one of two ways: with the help of library designers and of developer communities.

In the first case, the library specification is created by its developer. A language for describing library specifications (similar to the previously developed PanLang language [13]) is formed for this purpose; all the properties of the library expressed by the formalism developed can be defined by means of that language.

In the second case, the extraction methods will be based on exploiting the international programming experience (Empirical Software Engineering), with the structural and the behavioral components of the specifications stemming from the analysis of software repositories (Mining Software Repositories). In this case, only a skeleton of the specification is formed, with the remaining part to be refined manually.

The language for describing specifications corresponding to the formalism presented, and the methods for analyzing software repositories with the purpose of obtaining specification skeletons are currently being developed by the author’s research team; describing them is beyond the scope of this paper.

5.2 Using Formal Library Specifications

The formalism developed and described in this paper can be used in the future for solving a wide range of research and engineering problems, including automated defect detection in complex multi-component software projects, automated porting of applications to new libraries and automated generation of software documentation.

Library specifications are used as part of the solution for the problem of detecting software defects with the purpose of reducing the dimension of the detection problem. This is achieved by approximating the behavior of libraries and library functions by integrated visible behavior set in the specification. In this case, the library function is replaced by a system of predicates based on contracts and error states defined in the specification. This approach is used for the BMC analyzer Borealis, developed in the Program Analysis and Verification Laboratory of the Peter the Great St. Petersburg Polytechnic University [14]. A similar approach is used for the Aegis tool based on abstract interpretation, being developed in the same laboratory [15].

The task of automated migration of software to new libraries requires not only the external specification of the library’s behavior, but also a partial description of the internal semantics of the library. A semantic domain of the library is built based on the description of the internal semantics, and can be then used for checking library compatibility and automatically constructing the migration procedure. [16]

6. Conclusion

The study presented the results on creating formalism for software library specification. The formalism was built taking into account the entire range of problems that could be solved through it. The main idea was in using the same formal specification as a basis for several methods of software engineering: detection of software defects, automated software migration and software documentation generation. Due to limited space, the formalism was presented without going into details.

A direction for the future research is developing language support for the proposed formalism and implementing converters of language descriptions for the existing tools of error detection and software migration.

References

1. D. Lamb. IDL: sharing intermediate representations. ACM Trans. Program. Lang. Syst. 9, 3 (July 1987), 297-318. DOI=http://dx.doi.org/10.1145/24039.24040
2. https://msdn.microsoft.com/en-us/library/aa367091
3. http://www.omg.org/gettingstarted/omg_idl.htm
4. R. Allen and D. Garlan. Formalizing architectural connection. In Proceedings of the 16th international conference on Software engineering (ICSE '94). IEEE Computer Society Press, Los Alamitos, CA, USA, 71-80.
5. Хоар Ч. Взаимодействующие последовательные процессы. — М.: Мир, 1989. — 264 с.
6. A.W. Roscoe, Modelling and verifying key-exchange protocols using CSP and FDR, Proceedings of 1995 IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 1995.
7. L. de Alfaro and T. Henzinger. Interface automata. In Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-9). ACM, New York, NY, USA, 2001, 109-120. DOI=http://dx.doi.org/10.1145/503209.503226
8. M. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07). ACM, New York, NY, USA, 123-134. DOI=http://dx.doi.org/10.1145/1250734.1250749
9. S. Sankaranarayanan, F. Ivančić, and A. Gupta. Mining library specifications using inductive logic programming. In Proceedings of the 30th international conference on Software engineering (ICSE '08). ACM, New York, NY, USA, 131-140. DOI=http://dx.doi.org/10.1145/1368088.1368107

10. Thomas Ball and Sriram K. Rajamani. SLIC: a Specication Language for Interface Checking (of C). Microsoft Research, Technical Report, MSR-TR-2001-21. 2002
11. Gary T. Leavens. The future of library specification. In Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10). ACM, New York, NY, USA, 211-216. DOI=10.1145/1882362.1882407
12. Itsykson V. M., Zozulya A.V. The formalism for description of the partial specifications of program envinroment components. St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunication and Control Systems. N 4, 2011. – SPb: Publishing of Polytechnic University - pp. 81-90.
13. Itsykson V.M., Glukhikh M.I. A program component behavior specification language. St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunication and Control Systems. N 3, 2010, SPb: Publishing of Polytechnic University cc. 63-71.
14. M.Kh. Akhin, M.A. Belyaev, V.M. Itsykson. Software defect detection by combining bounded model checking and approximations of functions / Automatic Control and Computer Sciences, December 2014, Volume 48, Issue 7, pp 389-397
15. V. Itsykson, M. Moiseev ; V. Tsesko ; A. Zakharov. Automatic defects detection in industrial C/C++ software. In proceeding of Software Engineering Conference in Russia (CEE-SECR), 2009 5th Central and Eastern European. IEEE, Moscow, 07 DOI=10.1109/CEE-SECR.2009.5501189, pp 50-55
16. Itsykson V.M., Zozulya A.V. Automated Program Transformation for Migration to New Libraries. Software Engineering. 2012. N 6. pp. 8-14

УДК 004.8

Formalisms for conceptual design of information systems*

Anureev I.S. (Institute of Informatics Systems)

A class of information systems considered in this paper is defined as follows: a system belongs to the class if its change can be caused by both its environment and factors inside the system, and there is an information transfer from it to its environment and from its environment to it. Two formalisms (information transition systems and conceptual transition systems) for abstract unified modelling of the artifacts (concept sketches and models) of the conceptual design of information systems of the class, early phase of information systems design process, are proposed. Information transition defines the abstract unified information model for the artifacts, based on such general concepts as state, information query, answer and transition. Conceptual transition systems are a formalism for conceptual modelling of information transition systems. They defines the abstract unified conceptual model for the artifacts. The basic definitions of the theory of conceptual transition systems are given. A language of conceptual transition systems is defined.

Keywords: information system, information transition system, conceptual structure, ontology, ontological element, conceptual, conceptual state, conceptual configuration, conceptual transition system, conceptual information transition model, transition system, CTSL

1. Introduction

The conceptual models play an important role in the overall system development life cycle [1]. Numerous conceptual modelling techniques have been created, but all of them have a limited number of kinds of ontological elements and therefore can only represent ontological elements of fixed conceptual granularity. For example, entity-relationship modelling technique [2] uses two kinds of ontological elements: entities and relationships.

The purpose of the paper is propose formalisms for abstract unified modelling of the artifacts (concept sketches and models) of the conceptual design of information systems (IS for short) by ontological elements of arbitrary conceptual granularity. In our two stage approach the informational and conceptual aspects of the system that the conceptual model represents are described by two separate formalisms. The first formalism describes the informational model of the system, and the second formalism describes the conceptual model of the informational

*Partially supported by RFBR under grants 15-01-05974 and 15-07-04144 and SB RAS interdisciplinary integration project No.15/10.

model.

An information transition system (ITS for short) is an extension of an information query system (IQS for short) characterized additionally by the exogenous and endogenous transition relations specifying transitions on states. The exogenous transition relation models change of an information system caused by its environment. It associates queries with binary relations on states called transition relations and answers returning by state pairs from these transition relations called transitions. The endogenous transition relation models change of an information system caused by factors inside the system. It is defined as a transition relation with answers returning by transitions of the transition relation.

A wide variety of information systems is modelled by ITSs in the information aspect, including database management systems with transitions initiated by queries, expert systems with transitions initiated by operations with facts and rules, social networks with transitions initiated by actions of users in accordance with certain communications protocols, abstract machines specifying operational semantics of programming languages with transitions initiated by instructions of abstract machines, verification condition generators specifying axiomatic semantics of programming languages with transitions initiated by inference rules and so on.

We consider that the second formalism used for for conceptual modelling of ITSs must meet the following general requirements (in relation to modelling of a ITS):

1. It must model the conceptual structure of states and state objects of the ITS.
2. It must model the content of the conceptual structure.
3. It must model information queries, information query objects, answers and answer objects of the IQS.
4. It must model the interpretation function of the ITS.
5. It must be quite universal to model typical ontological elements (concepts, attributes, concept instances, relations, relation instances, individuals, types, domains, and so on.).
6. It must provide a quite complete classification of ontological elements, including the determination of their new kinds and subkinds with arbitrary conceptual granularity.
7. The model of the interpretation function must be extensible.
8. It must have language support. The language associated with the formalism must define syntactic representations of models of states, state objects, queries, query objects, answers and answer objects and includes the set of predefined basic query models.
9. It must model the change of the conceptual structure of states and state objects of the

ITS.

10. It must model the change of the content of the conceptual structure.
11. It must model the transition relations of the ITS.
12. The model of the exogenous transition relation must be extensible.

As is shown in [3], conceptual configuration systems (CCSs for short) meet the seven requirements in relation to IQSs. Comparison of CCSs with the abstract state machines [4, 5] which partially meet these requirements was made in [3]. In this paper we present an extension of CCSs, conceptual transition systems (CTSs for short) as the formalism satisfying the all above requirements.

The paper has the following structure. The preliminary concepts and notation are given in section 2. The basic definitions of the theory of CTSs are given in section 3. The language CTSL of CTSs is described in section 4. Semantics of executable elements in CTSL is defined in 5. We establish that CTSs meet the above requirements in section 6.

2. Preliminaries

2.1. Sets, sequences, multisets

Let O_b be the set of objects considered in this paper. Let S_t be a set of sets. Let I_{nt}, N_t, N_{t0} and B_l be sets of integers, natural numbers, natural numbers with zero and boolean values $true$ and $false$, respectively.

Let the names of sets be represented by capital letters possibly with subscripts and the elements of sets be represented by the corresponding small letters possibly with extended subscripts. For example, i_{nt} and $i_{nt.1}$ are elements of I_{nt}.

Let S_q be a set of sequences. Let $s_{t.(*)}$, $s_{t.\{*\}}$, and $s_{t.*}$ denote sets of sequences of the forms $(o_{b.1}, \ldots, o_{b.n_{t0}})$, $\{o_{b.1}, \ldots, o_{b.n_{t0}}\}$, and $o_{b.1}, \ldots, o_{b.n_{t0}}$ from elements of s_t. For example, $I_{nt.(*)}$ is a set of sequences of the form $(i_{nt.1}, \ldots, i_{nt.n_{t0}})$, and $i_{nt.*}$ is a sequence of the form $i_{nt.1}, \ldots, i_{nt.n_{t0}}$. Let $o_{b.1}, \ldots, o_{b.n_{t0}},$ denote $o_{b.1}, \ldots, o_{b.n_{t0}}$. Let $s_{t.(*n_{t0})}$, $s_{t.\{*n_{t0}\}}$, and $s_{t.*n_{t0}}$ denote sets of the corresponding sequences of the length n_{t0}.

Let $o_{b.1} \prec_{[\![s_q]\!]} o_{b.2}$ denote the fact that there exist $o_{b.*.1}$, $o_{b.*.2}$ and $o_{b.*.3}$ such that $s_q = o_{b.*.1},\ o_{b.1},\ o_{b.*.2},\ o_{b.2},\ o_{b.*.3}$, or $s_q = (o_{b.*.1},\ o_{b.1},\ o_{b.*.2},\ o_{b.2},\ o_{b.*.3})$.

Let $[o_b\ o_{b.1} \hookleftarrow o_{b.2}]$ denote the result of replacement of all occurrences of $o_{b.1}$ in o_b by $o_{b.2}$. Let $[s_q\ o_b \hookleftarrow_* o_{b.1}]$ denote the result of replacement of each element $o_{b.2}$ in s_q by $[o_{b.1}\ o_b \hookleftarrow o_{b.2}]$. For example, $[(a, b)\ x \hookleftarrow_* (f\ x)]$ denotes $((f\ a), (f\ b))$.

Let $[len\ s_q]$ denote the length of s_q. Let und denote the undefined value. Let $[s_q\ .\ n_t]$ denote the n_t-th element of s_q. If $[len\ s_q] < n_t$, then $[s_q\ .\ n_t] = und$. Let $[s_q\ +\ s_{q.1}]$, $[o_b\ .+\ s_q]$ and $[s_q\ +.\ o_b]$ denote $o_{b.*}, o_{b.*.1}$, $o_b, o_{b.*}$ and $o_{b.*}, o_b$, where $s_q = o_{b.*}$ and $s_{q.1} = o_{b.*.1}$.

Let $[and\ s_q]$ denote $(c_{nd.1}\ and\ \dots\ and\ c_{nd.n_t})$, where $s_q = c_{nd.1}$, ..., $c_{nd.n_t}$, and $[and]$ denote $true$. In the case of $n_t = 1$, the brackets can be omitted.

Let $o_{b.1}, o_{b.2} \in S_t \cup S_q$. Then $o_{b.1} =_{st} o_{b.2}$ denote that the sets of elements of $o_{b.1}$ and $o_{b.2}$ coincide, and $o_{b.1} =_{ml} o_{b.2}$ denote that the multisets of elements of $o_{b.1}$ and $o_{b.2}$ coincide.

2.2. Contexts

The terms used in the paper are context-dependent.

Let L_b be a set of objects called labels. Contexts have the form $[\![o_{b.*}]\!]$, where the elements of $o_{b.*}$ called embedded contexts have the form: l_b:o_b, l_b: or o_b.

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 $o_b[\![o_{b.*}]\!]$ denote the object o_b in the context $[\![o_{b.*}]\!]$.

The object 'in $[\![o_b, o_{b.*}]\!]$' can be reduced to 'in $[\![o_b]\!]$ in $[\![o_{b.*}]\!]$' if this does not lead to ambiguity.

2.3. Functions

Let F_n be a set of functions. Let A_{rg} and V_l be sets of objects called arguments and values. Let $[f_n\ a_{rg.*}]$ denote the application of f_n to $a_{rg.*}$.

Let $[support\ f_n]$ denote the support in $[\![f_n]\!]$, i. e. $[support\ f_n] = \{a_{rg} : [f_n\ a_{rg}] \neq und\}$. Let $[image\ f_n\ s_t]$ denote the image in $[\![f_n, s_t]\!]$, i. e. $[image\ f_n\ s_t] = \{[f_n\ a_{rg}] : a_{rg} \in s_t\}$. Let $[image\ f_n]$ denote the image in $[\![f_n, [support\ f_n]]\!]$. Let $[narrow\ f_n\ s_t]$ denote the function $f_{n.1}$ such that $[support\ f_{n.1}] = [support\ f_{n.1}] \cap s_t$, and $[f_{n.1}\ a_{rg}] = [f_n\ a_{rg}]$ for each $a_{rg} \in [support\ f_{n.1}]$. The function $f_{n.1}$ is called a narrowing of f_n to s_t. Let $[support\ f_{n.1}] \cap [support\ f_{n.2}] = \emptyset$. Let $f_{n.1} \cup f_{n.2}$ denote the union f_n of $f_{n.1}$ and $f_{n.2}$ such that $[f_n\ a_{rg}] = [f_{n.1}\ a_{rg}]$ for each $a_{rg} \in [support\ f_{n.1}]$, and $[f_n\ a_{rg}] = [f_{n.2}\ a_{rg}]$ for each $a_{rg} \in [support\ f_{n.2}]$. Let $f_{n.1} \subseteq f_{n.2}$ denote the fact that $[support\ f_{n.1}] \subseteq [support\ f_{n.2}]$, and $[f_{n.1}\ a_{rg}] = [f_{n.2}\ a_{rg}]$ for each $a_{rg} \in [support\ f_{n.1}]$.

An object u_p of the form $a_{rg} : v_l$ is called an update. Let U_p be a set of updates. The objects a_{rg} and v_l are called an argument and value in $[\![u_p]\!]$.

Let $[f_n\ u_p]$ denote the function $f_{n.1}$ such that $[f_{n.1}\ a_{rg}] = [f_n\ a_{rg}]$ if $a_{rg} \neq a_{rg}[\![u_p]\!]$, and

$[f_{n.1}\ a_{rg}[\![u_p]\!]] = v_l[\![u_p]\!]$. Let $[f_n\ u_p, u_{p.*n_t}]$ be a shortcut for $[[f_n\ u_p]\ u_{p.*n_t}]$. Let $[f_n\ a_{rg}.a_{rg.1}.\ \ldots\ .a_{rg.n_t} : v_l]$ be a shortcut for $[f_n\ a_{rg} : [[f_n\ a_{rg}]\ a_{rg.1}.\ \ldots\ .a_{rg.n_t} : v_l]]$. Let $[u_{p.*}]$ be a shortcut for $[f_n\ u_{p.*}]$, where $[support\ f_n] = \emptyset$.

Let C_{nd} be a set of objects called conditions. Let $[if\ c_{nd}\ then\ o_{b.1}\ else\ o_{b.2}]$ denote the object o_b such that

- if $c_{nd} = true$, then $o_b = o_{b.1}$;
- if $c_{nd} = false$, then $o_b = o_{b.2}$.

2.4. Attributes and multi-attributes

An object $o_{b.ma}$ of the form $(u_{p.*})$ is called a multi-attribute object. Let $O_{b.ma}$ be a set of multi-attribute objects. The elements of $[o_{b.ma}\ w \hookleftarrow_* a_{rg}[\![w]\!]]$ are called multi-attributes in $[\![o_{b.ma}]\!]$. Let $O_{b.ma}$ be a set of multi-attributes. The elements of $[o_{b.ma}\ w \hookleftarrow_* v_l[\![w]\!]]$ are called values in $[\![o_{b.ma}]\!]$. The sequence $u_{p.*}$ is called a sequence in $[\![o_{b.ma}]\!]$ and denoted by $[sequence\ in\ o_{b.ma}]$. An object v_l is a value in $[\![a_{tt.m}, o_{b.ma}]\!]$ if $o_{b.ma} = (u_{p.*.1}, a_{tt.m} : v_l, u_{p.*.2})$ for some $u_{p.*.1}$ and $u_{p.*.2}$.

An object $o_{b.ma}$ is an attribute object if the elements of $[o_{b.ma}\ w \hookleftarrow_* a_{rg}[\![w]\!]]$ are pairwise distinct. Let $O_{b.a}$ be a set of attribute objects. The multi-attributes in $[\![o_{b.a}]\!]$ are called attributes in $[\![o_{b.a}]\!]$. Let A_{tt} be a set of objects called attributes.

Let $[function\ o_{b.a}]$, $[o_{b.a}\ a_{tt}]$, and $[support\ o_{b.a}]$ denote $[[sequence\ in\ o_{b.a}]]$, $[[function\ o_{b.a}]\ a_{tt}]$, and $[support\ [function\ o_{b.a}]]$.

Let $[seq{-}to{-}att{-}obj\ s_q]$ denote $(1 : [s_q\ .\ 1], ..., [len\ s_q] : [s_q\ .\ [len\ s_q]])$. Let $o_{b.a} =_{st} (1 : v_{l.1}, ..., n_t : v_{l.n_t})$. Then $[att{-}obj{-}to{-}seq\ o_{b.a}]$ denote $(v_{l.1}, ..., v_{l.n_t})$.

3. Basic definitions of the theory of conceptual transition systems

Conceptual transition systems (CTSs) are transition systems in which states are conceptual configurations, and transition relations are binary relations on conceptual configurations. In this section the basic definitions of the theory of conceptual transition systems are presented. The defined structures of CTSs are constructed from atoms and, thus, defined implicitly in $[\![A_{tm}]\!]$.

3.1. Information transition systems

Let S_{tt} be a set of objects called states. An element t_{rn} of the form $(s_{tt.1}, s_{tt.2})$ is called a transition. Let T_{rn} be a set of transitions. The states $s_{tt.1}$ and $s_{tt.2}$ are called input and output states in $[\![t_{rn}]\!]$.

Let $S_{s.q}$ be a set of query systems. An object $s_{s.t.i}$ of the form $(s_{s.q}, t_{rn.rlt.ex}, t_{rn.rlt.en})$ is an information transition system if $t_{rn.rlt.ex} \in Q_r \times A_{ns} \times S_{tt} \times S_{tt} \to B_l$, $t_{rn.rlt.en} \in A_{ns} \times S_{tt} \times S_{tt} \to B_l$, and for all $q_r \in Q_r$ there exists $s_{tt} \in S_{tt}$ such that $[value\ q_r\ s_{tt}] \neq und$, or there exist $s_{tt.1} \in S_{tt}$, $s_{tt.2} \in S_{tt}$ and $a_{ns} \in A_{ns}$ such that $[t_{rn.rlt.ex}\ q_r\ a_{ns}\ s_{tt.1}\ s_{tt.2}] = true$. Let $S_{s.t.i}$ be a set of information transition systems.

The system $s_{s.q}$ is called a query system in $[\![s_{s.t.i}]\!]$. The function $t_{rn.rlt.ex}$ is called an exogenous transition relation in $[\![s_{s.t.i}]\!]$. The function $t_{rn.rlt.en}$ is called an endogenous transition relation in $[\![s_{s.t.i}]\!]$. Let $s_{tt.1} \to_{q_r,a_{ns}} s_{tt.2}$ and $s_{tt.1} \to_{a_{ns}} s_{tt.2}$ be shortcuts for $[t_{rn.rlt.ex}\ q_r\ a_{ns}\ s_{tt.1}\ s_{tt.2}] = true$ and $[t_{rn.rlt.en}\ a_{ns}\ s_{tt.1}\ s_{tt.2}] = true$, respectively.

The elements of $S_{tt}[\![s_{s.q}]\!]$, $O_{b.s}[\![s_{s.q}]\!]$, $Q_r[\![s_{s.q}]\!]$, $O_{b.q}[\![s_{s.q}]\!]$, $A_{ns}[\![s_{s.q}]\!]$ and $O_{b.a}[\![s_{s.q}]\!]$ are called states, state objects, queries, query objects, answers and answer objects in $[\![s_{s.t.i}]\!]$, respectively. The function $value[\![s_{s.q}]\!]$ is called a query interpretation in $[\![s_{s.t.i}]\!]$.

A query q_r is an information query in $[\![s_{s.t.i}]\!]$ if $[value\ q_r\ s_{tt}] \neq und$ for some s_{tt}. A query q_r is a change query in $[\![s_{s.t.i}]\!]$ if $[t_{rn.rlt.ex}\ q_r\ a_{ns}\ s_{tt.1}\ s_{tt.2}] = true$ for some $s_{tt.1}$, $s_{tt.2}$ and a_{ns}.

A system $s_{s.t.i}$ executes t_{rn} if $s_{tt.1}[\![t_{rn}]\!] \to_{q_r,a_{ns}} s_{tt.2}[\![t_{rn}]\!]$ for some q_r and a_{ns}, or $s_{tt.1}[\![t_{rn}]\!] \to_{a_{ns}} s_{tt.2}[\![t_{rn}]\!]$ for some a_{ns}. A system $s_{s.t.i}$ transits from $s_{tt.1}$ to $s_{tt.2}$ if $s_{s.t.i}$ executes $(s_{tt.1}, s_{tt.2})$.

3.2. Substitutions, patterns, pattern specifications, instances

A function $s_b \in E_l \to E_{l.*}$ is called a substitution. Let S_b be a set of substitutions. A function $subst \in S_b \times E_{l.*} \to E_{l.*}$ is a substitution function if it is defined as follows (the first proper rule is applied):

- if $e_l \in [support\ s_b]$, then $[subst\ s_b\ e_l] = [s_b\ e_l]$;
- $[subst\ s_b\ a_{tm}] = a_{tm}$;
- $[subst\ s_b\ l_b : e_l] = [subst\ s_b\ l_b] : [subst\ s_b\ e_l]$;
- $[subst\ s_b\ e_l :: nosubst] = e_l$;
- $[subst\ s_b\ e_l :: (nosubstexcept\ e_{l.*})] = [subst\ [narrow\ s_b\ \{e_{l.*}\}]\ e_l]$;
- $[subst\ s_b\ e_l :: s_{rt}] = [subst\ s_b\ e_l] :: [subst\ s_b\ s_{rt}]$;
- $[subst\ s_b\ (e_{l.*})] = ([e_{l.*}\ w \hookleftarrow_*\ [subst\ s_b\ w]])$;
- $[subst\ s_b\ e_{l.*}] = [e_{l.*}\ w \hookleftarrow_*\ [subst\ s_b\ w]]$.

The sort *nosubst* specifies the elements to which the substitution s_b is not applied. The sort $(\mathit{nosubstexcept}\ e_{l.*})$ specifies the elements to which the narrowing of the substitution s_b to the set $e_{l.*}$ is applied. An element p_t is a pattern in $[\![e_l, s_b]\!]$ if $[\mathit{subst}\ s_b\ p_t] = e_l$. Let P_t be a set of patterns. An element i_{nst} is an instance in $[\![p_t, s_b]\!]$ if $[\mathit{subst}\ s_b\ p_t] = i_{nst}$. Let I_{nst} be a set of instances.

Let V_r and $V_{r.s}$ be sets of objects called element variables and sequence variables, respectively. An element $p_{t.s}$ of the form $(p_t, (v_{r.*}), (v_{r.s.*}))$ is a pattern specification if $\{v_{r.s.*}\} \cap \{v_{r.*}\} = \emptyset$, and the elements of $\{v_{r.*}\} \cup \{v_{r.s.*}\}$ are pairwise distinct. Let $P_{t.s}$ be a set of pattern specifications.

The objects p_t, $(v_{r.*})$, and $(v_{r.s.*})$ are called a pattern, element variable specification, and sequence variable specification in $[\![p_{t.s}]\!]$. The elements of $v_{r.*}$ and $v_{r.s.*}$ are called element pattern variables and sequence pattern variables in $[\![p_{t.s}]\!]$, respectively.

An element i_{nst} is an instance in $[\![p_{t.s}, s_b]\!]$ if $[\mathit{support}\ s_b] = \{v_{r.*}\}$, $[s_b\ v_r] \in E_l$ for $v_r \in \{v_{r.*}\} \setminus \{v_{r.s.*}\}$, $[s_b\ v_r] \in E_{l.*}$ for $v_r \in \{v_{r.s.*}\}$, and i_{nst} is an instance in $[\![p_t, s_b]\!]$. An element i_{nst} is an instance in $[\![p_{t.s}]\!]$ if there exists s_b such that i_{nst} is an instance in $[\![p_{t.s}, s_b]\!]$.

A function $m_t \in E_l \times P_{t.s} \to S_b$ is a match if the following property holds:

- if $[m_t\ e_l\ p_{t.s}] = s_b$, then e_l is an instance in $[\![p_{t.s}, s_b]\!]$.

An element i_{nst} is an instance in $[\![p_{t.s}, m_t, s_b]\!]$ if $[m_t\ i_{nst}\ p_{t.s}] = s_b$. An element i_{nst} is an instance in $[\![p_{t.s}, m_t]\!]$ if there exists s_b such that i_{nst} is an instance in $[\![p_{t.s}, m_t, s_b]\!]$.

3.3. The transition relation

Let $S_{s.c.c}$ be a set of conceptual configuration systems. Let C_{nf} be a set of conceptual configurations. An element t_{rn} of the form $(c_{nf.1}, c_{nf.2})$ is called a transition. Let T_{rn} be a set of transitions. The configurations $c_{nf.1}$ and $c_{nf.2}$ are called input and output configurations in $[\![t_{rn}]\!]$.

The transition relations of a IQS is modelled by the transition relation $t_{rn.rlt} \in T_{rn} \to B_l$ based on atomic exogenous transition relations, transition rules, atomic endogenous transition relations, the exogenous transition order and the endogenous transition order. The exogenous transition relation of the IQS is modelled by atomic exogenous transition relations and transition rules. The endogenous transition relation of the IQS is modelled by atomic endogenous transition relations.

Transitions from a configuration c_{nf} in $[\![t_{rn.rlt}]\!]$ are executed by a program in $[\![c_{nf}]\!]$. An element sequence p_{rg} is a program in $[\![c_{nf}]\!]$ if $[c_{nf}\ (0 : ())\ ::\ \mathit{state}\ ::\ \mathit{program}] = (p_{rg})$. Let

P_{rg} be a set of programs. Thus, programs in configurations are specified by the conceptual $(0 : ()) :: state :: program$ from the substate *program* of the configurations. A program in $[\![c_{nf}]\!]$ is empty if $[c_{nf}\ (0 : ()) :: state :: program] = ()$. Atomic exogenous transition relations and transition rules define transitions executed by the first element of the program. Atomic endogenous transition relations define transitions executed in the case of the empty program.

Let $c_{nf.1} \to c_{nf.2}$ be a shortcut for $[t_{rn.rlt}\ c_{nf.1}\ c_{nf.2}] = true$. Transitions can return values. An element v_l is a value in $[\![c_{nf}]\!]$ if $v_l = [c_{nf}\ (0 : ()) :: state :: value]$. An element v_l is a value in $[\![t_{rn}]\!]$ if $c_{nf.1}[\![t_{rn}]\!] \to c_{nf.2}[\![t_{rn}]\!]$, and v_l is a value in $[\![c_{nf.2}[\![t_{rn}]\!]]\!]$. Thus, the returned values in transitions are specified by the conceptual $(0 : ()) :: state :: value$ from the substate *value* of output configurations of the transitions. A transition t_{rn} returns a value v_l if v_l is a value in $[\![t_{rn}]\!]$. A transition t_{rn} returns (or generates) an exception e_{xc} if e_{xc} is a value in $[\![t_{rn}]\!]$. A transition t_{rn} is normally executed if t_{rn} returns no exception.

The special variables $conf :: in$ and $val :: in$ reference to the current configuration and the value in the current configuration, respectively, in the definitions below.

An object $t_{rn.rlt.ex}$ of the form $(p_t, (v_{r.*}), (v_{r.s.*}), f_n)$ is an atomic exogenous transition relation if $(p_t, (v_{r.*}), (v_{r.s.*}))$ is a pattern specification, $conf :: in \notin \{v_{r.*}\} \cup \{v_{r.s.*}\}$, $val :: in \notin \{v_{r.*}\} \cup \{v_{r.s.*}\}$, $f_n \in S_b \to (T_{rn} \to B_l)$, $[support\ f_n] = \{s_b : [support\ s_b] = \{v_{r.*}\} \cup \{v_{r.s.*}\} \cup \{conf :: in, val : in\}, [s_b\ v_r] \in E_l$ for $v_r \in \{v_{r.*}\}$ and $[s_b\ v_r] \in E_{l.*}$ for $v_r \in \{v_{r.s.*}\}\}$. Let $T_{rn.rlt.ex}$ be a set of atomic exogenous transition relations. Let $c_{nf.1} \to_{f_n, s_b} c_{nf.2}$ be a shortcut for $[[f_n\ s_b]\ c_{nf.1}\ c_{nf.2}] = true$.

The objects p_t, $(v_{r.*})$, $(v_{r.s.*})$, and f_n are called a pattern, element variable specification, sequence variable specification, and value in $[\![t_{rn.rlt.ex}]\!]$. The elements of $v_{r.*}$ and $v_{r.s.*}$ are called element pattern variables and sequence pattern variables in $[\![t_{rn.rlt.ex}]\!]$, respectively.

A function $t_{rn.rlt.ex.s} \in E_l \to T_{rn.rlt.ex}$ is called an atomic exogenous transition specification if $[support\ t_{rn.rlt.ex.s}]$ is finite. A relation $t_{rn.rlt.ex}$ is an atomic exogenous transition relation in $[\![t_{rn.rlt.ex.s}]\!]$ if $[t_{rn.rlt.ex.s}\ n_m] = t_{rn.rlt.ex}$ for some $n_m \in E_l$. An element n_m is a name in $[\![t_{rn.rlt.ex}, t_{rn.rlt.ex.s}]\!]$ if $[t_{rn.rlt.ex.s}\ n_m] = t_{rn.rlt.ex}$. An element n_m a name in $[\![t_{rn.rlt.ex.s}]\!]$ if n_m is a name in $[\![t_{rn.rlt.ex}, t_{rn.rlt.ex.s}]\!]$ for some $t_{rn.rlt.ex}$. Let $c_{nf.1} \to_{n_m, s_b} c_{nf.2}$ be a shortcut for $c_{nf.1} \to_{f_n[\![[t_{rn.rlt.ex.s}\ n_m]]\!], s_b} c_{nf.2}$.

An element r_l of the form $(p_t, (v_{r.*}), (v_{r.s.*}), (b_d))$ is a transition rule if $b_d \in E_{l.*}$, $(p_t, (v_{r.*}), (v_{r.s.*}))$ is a pattern specification, $conf :: in \notin \{v_{r.*}\} \cup \{v_{r.s.*}\}$, and $val :: in \notin \{v_{r.*}\} \cup \{v_{r.s.*}\}$. Let R_l be a set of transition rules.

The objects p_t, $(v_{r.*})$, $(v_{r.s.*})$ and b_d are called a pattern, element variable specification, sequence variable specification and body in $[\![r_l]\!]$. The elements of $v_{r.*}$ and $v_{r.s.*}$ are called element pattern variables and sequence pattern variables in $[\![r_l]\!]$, respectively.

An attribute element $r_{l.s}$ is called a transition rule specification if $[support\ r_{l.s}] \subseteq E_l$, and $[image\ r_{l.s}] \subseteq E_l$. A rule r_l is a rule in $[\![r_{l.s}]\!]$ if $[r_{l.s}\ n_m] = r_l$ for some $n_m \in E_l$. An element n_m is a name in $[\![r_l, r_{l.s}]\!]$ if $[r_{l.s}\ n_m] = r_l$. An element n_m a name in $[\![r_{l.s}]\!]$ if n_m is a name in $[\![r_l, r_{l.s}]\!]$ for some r_l.

A function $t_{rn.rlt.en} \in \{c_{nf} : [c_{nf}\ (0 : ()) :: state :: program] = ()\} \times C_{nf} \to B_l$ is called an atomic endogenous transition relation. Let $T_{rn.rlt.en}$ be a set of atomic endogenous transition relations.

A function $t_{rn.rlt.en.s} \in E_l \to T_{rn.rlt.en}$ is called an atomic endogenous transition specification if $[support\ t_{rn.rlt.en.s}]$ is finite. A relation $t_{rn.rlt.en}$ is an atomic endogenous transition relation in $[\![t_{rn.rlt.en.s}]\!]$ if $[t_{rn.rlt.en.s}\ n_m] = t_{rn.rlt.en}$ for some $n_m \in E_l$. An element n_m is a name in $[\![t_{rn.rlt.en}, t_{rn.rlt.en.s}]\!]$ if $[t_{rn.rlt.en.s}\ n_m] = t_{rn.rlt.en}$. An element n_m a name in $[\![t_{rn.rlt.en.s}]\!]$ if n_m is a name in $[\![t_{rn.rlt.en}, t_{rn.rlt.en.s}]\!]$ for some $t_{rn.rlt.en}$. Let $c_{nf} \to_{n_m} c_{nf}$ be a shortcut for $[[t_{rn.rlt.en.s}\ n_m]\ c_{nf}\ c_{nf.1}] = true$.

Let $[support\ t_{rn.rlt.ex.s}]$, $[support\ t_{rn.rlt.en.s}]$ and $[support\ r_{l.s}]$ be pairwise disjoint.

An element $o_{rd.trn.ex}$ of the form $(n_{m.*})$ is called an exogenous transition order in $[\![t_{rn.rlt.ex.s}, r_{l.s}]\!]$ if $\{n_{m.*}\} \subseteq [support\ t_{rn.rlt.ex.s}] \cup [support\ r_{l.s}]$, and the elements of $n_{m.*}$ are pairwise distinct. It specifies the order of application of atomic exogenous transition relations and transition rules.

An element $o_{rd.trn.en}$ of the form $(n_{m.*})$ is called an endogenous transition order in $[\![t_{rn.rlt.en.s}]\!]$ if $\{n_{m.*}\} \subseteq [support\ t_{rn.rlt.en.s}]$, and the elements of $n_{m.*}$ are pairwise distinct. It specifies the order of application of atomic endogenous transition relations.

The information about the transition rule specification and the transition orders is stored in the substate *transition* of the configurations. The conceptuals $(0 : rules) :: state :: transition$, $(-1 : exogenous, 0 : order) :: state :: transition$ and $(-1 : endogenous, 0 : order) :: state :: transition$ define the transition rule specification, exogenous transition order and endogenous transition order. The conceptual $(0 : history) :: state :: transition$ defines the substates that store the information about transitions preceding the transition to the current configuration.

An element c_{nf} is consistent with $(t_{rn.rlt.ex.s}, r_{l.s}, t_{rn.rlt.en.s}, o_{rd.trn.ex}, o_{rd.trn.en})$ if the following properties hold:

- if $[support\ t_{rn.rlt.ex.s}] \cap [support\ [c_{nf}\ (0 : rules) :: state :: transition]] = \emptyset$;

- if $[support\ o_{rd.trn.en}] \cap [support\ [c_{nf}\ (0 : rules) :: state :: transition]] = \emptyset$;
- if $r_{l.s} \subseteq [c_{nf}\ (0 : rules) :: state :: transition]$;
- if $n_{m.1} \prec_{[\![o_{rd.trn.ex}]\!]} n_{m.2}$, and $n_{m.1},\ n_{m.2} \in [c_{nf}\ (-1 : exogenous, 0 : order) :: state :: transition]$, then $n_{m.1} \prec_{[\![[c_{nf}\ (-1: exogenous, 0: order)::state::transition]]\!]} n_{m.2}$;
- if $n_{m.1} \prec_{[\![o_{rd.trn.en}]\!]} n_{m.2}$, and $n_{m.1},\ n_{m.2} \in [c_{nf}\ (-1 : endogenous, 0 : order) :: state :: transition]$, then $n_{m.1} \prec_{[\![[c_{nf}\ (-1: endogenous, 0: order)::state::transition]]\!]} n_{m.2}$.

Let $e_{l.*}\ \#\ c_{nf}$ be a shortcut for $[c_{nf}\ program.(0 : ()) : (e_{l.*})]$. Let $e_{l.*}\ \#\ v_l\ \#\ c_{nf}$ be a shortcut for $[c_{nf}\ program.(0 : ()) : (e_{l.*}), value.(0 : ()) : v_l]$.

Let $[add{-}history\ c_{nf.1}\ to\ c_{nf.2}]$ denote $[narrow\ c_{nf.1}\ [support\ c_{nf.1}] \setminus \{[c_{nf.1}\ (0 : history) :: state :: transition]\}] \cup [narrow\ c_{nf.1}\ \{[c_{nf.1}\ (0 : history) :: state :: transition]\}]$. A function $t_{rn.rlt} \in C_{nf.c} \times C_{nf} \to B_l$ is a transition relation in $[\![t_{rn.rlt.ex.s}, r_{l.s}, t_{rn.rlt.en.s},\ o_{rd.trn.ex}, o_{rd.trn.en}]\!]$ if it is defined by the following definition rules (the first proper rule is applied):

- if c_{nf} is not consistent with $(t_{rn.rlt.ex.s}, r_{l.s}, t_{rn.rlt.en.s}, o_{rd.trn.ex}, o_{rd.trn.en})$, then $[t_{rn.rlt}\ c_{nf}\ c_{nf.1}] = false$;
- if $t_{rn.rlt.ex} = [t_{rn.rlt.ex.s}\ n_m]$, e_l is an instance in $[\![p_{t.s}[\![t_{rn.rlt.ex}]\!], m_t, s_b]\!]$, $e_{l.*}\ \#\ c_{nf} \to_{n_m, s_b \cup (conf::in:c_{nf}, val::in:v_l[\![c_{nf}]\!])} e_{l.*.1}\ \#\ v_l\ \#\ c_{nf.1}$, and $v_l \neq und$, then $(execute{-}exogenous{-}transition,\ e_l,\ (n_m\ n_{m.*})),\ e_{l.*}\ \#\ c_{nf} \to e_{l.*.1}\ \#\ v_l\ \#\ c_{nf.1}$;
- if $t_{rn.rlt.ex} = [t_{rn.rlt.ex.s}\ n_m]$, e_l is an instance in $[\![p_{t.s}[\![t_{rn.rlt.ex}]\!], m_t, s_b]\!]$, $e_{l.*}\ \#\ c_{nf} \to_{n_m, s_b \cup (conf::in:c_{nf}, val::in:v_l[\![c_{nf}]\!])} e_{l.*.1}\ \#\ u_{nd}\ \#\ c_{nf.1}$, then $(execute{-}exogenous{-}transition,\ e_l,\ (n_m\ n_{m.*})),\ e_{l.*}\ \#\ c_{nf} \to (execute{-}exogenous{-}transition,\ e_l,\ (n_{m.*})),\ e_{l.*}\ \#\ [add{-}history\ c_{nf.1}\ to\ c_{nf}]$;
- if $t_{rn.rlt.ex} = [t_{rn.rlt.ex.s}\ n_m]$, and e_l is not an instance in $[\![p_{t.s}[\![t_{rn.rlt.ex}]\!], m_t]\!]$, then $(execute{-}exogenous{-}transition,\ e_l,\ (n_m,\ n_{m.*})),\ e_{l.*}\ \#\ c_{nf} \to (execute{-}exogenous{-}transition,\ e_l,\ (n_{m.*})),\ e_{l.*}\ \#\ c_{nf}$;
- if $r_l = [[c_{nf}\ (0 : rules) :: state :: transition]\ n_m]$, and e_l is an instance in $[\![p_{t.s}[\![r_l]\!], m_t, s_b]\!]$, then $(execute{-}exogenous{-}transition,\ e_l,\ (n_m\ n_{m.*})),\ e_{l.*}\ \#\ c_{nf} \to ([subst\ s_b \cup (conf :: in : c_{nf}, val :: in : v_l[\![c_{nf}]\!])\ b_d[\![r_l]\!]],\ (execute{-}exogenous{-}transition,\ e_l,\ (n_m\ n_{m.*}),\ (e_{l.*}),\ c_{nf}),\ e_{l.*}\ \#\ c_{nf}$;
- if $v_l \neq und$, then $(execute{-}exogenous{-}transition,\ e_l,\ (n_m\ n_{m.*}),\ (e_{l.*.1}),\ c_{nf.1}),\ e_{l.*}\ \#\ v_l\ \#\ c_{nf} \to e_{l.*}\ \#\ v_l\ \#\ c_{nf}$;
- $(execute{-}exogenous{-}transition,\ (n_m\ n_{m.*}),\ e_l,\ (e_{l.*.1}),\ c_{nf.1}),\ e_{l.*}\ \#\ und\ \#\ c_{nf} \to (execute{-}exogenous{-}transition,\ e_l,\ (n_{m.*})),\ e_{l.*.1}\ \#\ [add{-}history\ c_{nf}\ to\ c_{nf.1}]$;

- if $r_l = [[c_{nf}\ (0 : rules) :: state :: transition]\ n_m]$, and e_l is not an instance in $[\![p_{t.s}[\![r_l]\!], m_t]\!]$, then $(execute{-}exogenous{-}transition,\ e_l,\ (n_m\ n_{m.*})),\ e_{l.*}\ \#\ c_{nf}\ \rightarrow\ (execute{-}exogenous{-}transition,\ e_l,\ (n_{m.*})),\ e_{l.*}\ \#\ c_{nf}$;
- $(execute{-}exogenous{-}transition,\ e_l,\ ()),\ e_{l.*}\ \#\ c_{nf} \rightarrow e_{l.*}\ \#\ und\ \#\ c_{nf}$;
- if $t_{rn.rlt.en} = [t_{rn.rlt.en.s}\ n_m]$, $c_{nf} \rightarrow_{n_m} e_{l.*}\ \#\ v_l\ \#\ c_{nf.1}$, and $v_l \neq und$, then $(execute{-}endogenous{-}transition,\ (n_m\ n_{m.*}))\ \#\ c_{nf} \rightarrow e_{l.*}\ \#\ v_l\ \#\ c_{nf.1}$;
- if $t_{rn.rlt.en} = [t_{rn.rlt.en.s}\ n_m]$, and $c_{nf} \rightarrow_{n_m} e_l\ e_{l.*}\ \#\ u_{nd}\ \#\ c_{nf.1}$, then $(execute{-}endogenous{-}transition,\ (n_m\ n_{m.*}))\ \#\ c_{nf} \rightarrow e_l\ e_{l.*}\ \#\ u_{nd}\ \#\ c_{nf.1}$;
- if $t_{rn.rlt.en} = [t_{rn.rlt.en.s}\ n_m]$, and $c_{nf} \rightarrow_{n_m}\ \#\ u_{nd}\ \#\ c_{nf.1}$, then $(execute{-}endogenous{-}transition,\ (n_m\ n_{m.*}))\ \#\ c_{nf}\ \rightarrow\ (execute{-}endogenous{-}transition,\ (n_{m.*}))\ \#\ [add{-}history\ c_{nf.1}\ to\ c_{nf}]$;
- $e_l,\ e_{l.*}\ \#\ c_{nf} \rightarrow (execute{-}exogenous{-}transition,\ e_l,\ [c_{nf}\ (-1 : exogenous, 0 : order) :: state :: transition]),\ e_{l.*}\ \#\ c_{nf}$;
- $\#\ c_{nf} \rightarrow (execute{-}endogenous{-}transition,\ [c_{nf}\ (-1 : endogenous, 0 : order) :: state :: transition]),\ \#\ c_{nf}$.

3.4. Conceptual transition systems

An object $s_{s.t.c}$ of the form $(s_{s.c.c}, t_{rn.rlt.ex.s}, r_{l.s}, t_{rn.rlt.en.s}, o_{rd.trn.ex}, o_{rd.trn.en})$ is a conceptual transition system if $s_{s.c.c}$ is a conceptual configuration system, $t_{rn.rlt.ex.s}$, $r_{l.s}$, $t_{rn.rlt.en.s}$, $o_{rd.trn.ex}$ and $o_{rd.trn.en}$ are an atomic exogenous transition specification, transition rule specification, atomic endogenous transition specification, exogenous transition order and endogenous transition order in $[\![A_{tm}[\![s_{s.c.c}]\!]]\!]$, and the sets $[support\ t_{rn.rlt.ex.s}]$, $[support\ t_{rn.rlt.en.s}]$ and $[support\ r_{l.s}]$ are pairwise disjoint. It specifies the transition system $(C_{nf}[\![s_{s.c.c}]\!], t_{rn.rlt}[\![t_{rn.rlt.ex.s}, r_{l.s}, t_{rn.rlt.en.s}, o_{rd.trn.ex}, o_{rd.trn.en}, m_t[\![s_{s.c.c}]\!]]\!])$. Let $S_{s.t.c}$ be a set of conceptual transition systems.

The elements of $A_{tm}[\![s_{s.c.c}]\!]$, $E_l[\![s_{s.c.c}]\!]$, $C_{ncpl}[\![s_{s.c.c}]\!]$, $S_{tt}[\![s_{s.c.c}]\!]$, $C_{nf}[\![s_{s.c.c}]\!]$ and $T_{rn}[\![A_{tm}[\![s_{s.c.c}]\!]]\!]$ are called atoms, elements, conceptuals, states, configurations and transitions in $[\![s_{s.t.c}]\!]$.

The objects $t_{rn.rlt.ex.s}$, $r_{l.s}$, $t_{rn.rlt.en.s}$, $o_{rd.trn.ex}$, $o_{rd.trn.en}$, $i_{ntr.a.s}[\![s_{s.c.c}]\!]$, $d_{f.s}[\![s_{s.c.c}]\!]$, $o_{rd.intr}[\![s_{s.c.c}]\!]$ and $m_t[\![s_{s.c.c}]\!]$ are called an atomic exogenous transition specification, transition rule specification, atomic endogenous transition specification, exogenous transition order, endogenous transition order, atomic element interpretation specification, element definition specification, element intepretation order and match in $[\![s_{s.t.c}]\!]$.

The function $t_{rn.rlt}[\![t_{rn.rlt.ex.s}, r_{l.s}, t_{rn.rlt.en.s}, o_{rd.trn.ex}, o_{rd.trn.en}, m_t]\!]$ is called a transition relation in $[\![s_{s.t.c}]\!]$. A system $s_{s.t.c}$ executes t_{rn} if $s_{tt.1}[\![t_{rn}]\!] \to s_{tt.2}[\![t_{rn}]\!]$. A system $s_{s.t.c}$ transits from $s_{tt.1}$ to $s_{tt.2}$ if $s_{s.t.c}$ executes $(s_{tt.1}, s_{tt.2})$.

An element e_l is interpretable in $[\![s_{s.t.c}]\!]$ if e_l is interpretable in $[\![s_{s.c.c}[\![s_{s.t.c}]\!]]\!]$.

An element e_l is executable in $[\![s_{s.t.c}]\!]$ if there exist n_m such that e_l is an instance in $[\![p_{t.s}[\![[t_{rn.rlt.ex.s}\ n_m]]\!], m_t]\!]$, or e_l is an instance in $[\![p_{t.s}[\![[r_{l.s}\ n_m]]\!], m_t]\!]$.

3.5. Conceptual information transition models

An object $m_{dl.t.q.c}$ of the form $(s_{s.t.c}, r_{pr.s}, r_{pr.q}, r_{pr.a})$ is a conceptual information transition model in $[\![s_{s.t.i}]\!]$ if $(s_{s.c.c}[\![s_{s.t.c}]\!], r_{pr.s}, r_{pr.q}, r_{pr.a})$ is a conceptual query model in $[\![s_{s.q}[\![s_{s.t.i}]\!]]\!]$, $[t_{rn.rlt.ex}[\![s_{s.t.i}]\!]\ q_r\ a_{ns}\ s_{tt.1}\ s_{tt.2}] = [t_{rn.rlt}[\![s_{s.t.c}]\!]\ [[r_{pr.s}\ s_{tt.1}]\ (0 : ()) :: state :: program : ([r_{pr.q}\ q_r])]$ $[[r_{pr.s}\ s_{tt.2}]\ (0 : ()) :: state :: value : [r_{pr.a}\ a_{ns}]]]$, and $[t_{rn.rlt.en}[\![s_{s.t.i}]\!]\ a_{ns}\ s_{tt.1}\ s_{tt.2}] = [t_{rn.rlt}[\![s_{s.t.c}]\!]$ $[[r_{pr.s}\ s_{tt.1}]\ (0 : ()) :: state :: program : ()]\ [[r_{pr.s}\ s_{tt.2}]\ (0 : ()) :: state :: value : [r_{pr.a}\ a_{ns}]]]$. Let $M_{dl.t.q.c}$ be a set of conceptual query transition models.

The objects $s_{s.c.c}[\![s_{s.t.c}]\!]$ and $s_{s.t.c}$ are called a conceptual configuration system and conceptual transition system in $[\![m_{dl.t.q.c}]\!]$, respectively. The functions $r_{pr.s}$, $r_{pr.q}$ and $r_{pr.a}$ are called a state representation, query representation and answer representation in $[\![m_{dl.t.q.c}]\!]$.

A system $s_{s.t.i}$ is conceptually modelled in $[\![s_{s.t.c}]\!]$ if there exists $m_{dl.t.q.c}$ such that $s_{s.t.c} = s_{s.t.c}[\![m_{dl.t.q.c}]\!]$, and $m_{dl.t.q.c}$ is a conceptual query model in $[\![s_{s.t.i}]\!]$. The set $[image\ r_{pr.s}]$ is called an ontology in $[\![s_{s.t.i}, m_{dl.t.q.c}]\!]$.

3.6. Extensions

A system $s_{s.t.i.1}$ is an extension of $s_{s.t.i.2}$ if $s_{s.q}[\![s_{s.t.i.1}]\!]$ is an extension of $s_{s.q}[\![s_{s.t.i.2}]\!]$, and $s_t[\![s_{s.t.i.1}]\!] \subseteq s_t[\![s_{s.t.i.2}]\!]$ for each $s_t \in \{t_{rn.rlt.ex}, t_{rn.rlt.en}\}$.

A system $s_{s.t.c.1}$ is an extension of $s_{s.t.c.2}$ if $s_{s.c.c}[\![s_{s.t.c.1}]\!]$ is an extension of $s_{s.c.c}[\![s_{s.t.c.2}]\!]$, $s_t[\![s_{s.t.c.1}]\!] \subseteq s_t[\![s_{s.t.c.2}]\!]$ for each $s_t \in \{t_{rn.rlt.ex.s}, r_{l.s}, t_{rn.rlt.en.s}\}$, and the following property hold:

- if $n_{m.1} \prec_{[\![o_{rd.trn.ex}[\![s_{s.t.c.1}]\!]]\!]} n_{m.2}$, and $n_{m.1}, n_{m.2} \in o_{rd.trn.ex}[\![s_{s.t.c.2}]\!]$, then $n_{m.1} \prec_{[\![o_{rd.trn.ex}[\![s_{s.t.c.2}]\!]]\!]} n_{m.2}$;
- if $n_{m.1} \prec_{[\![o_{rd.trn.en}[\![s_{s.t.c.1}]\!]]\!]} n_{m.2}$, and $n_{m.1}, n_{m.2} \in o_{rd.trn.en}[\![s_{s.t.c.2}]\!]$, then $n_{m.1} \prec_{[\![o_{rd.trn.en}[\![s_{s.t.c.2}]\!]]\!]} n_{m.2}$.

A CCS l_n is a language of CTSs if the conceptual structures (atoms, elements, conceptuals and so on) of l_n is syntactically defined.

3.7. Programs

A program p_{rg} is executed in $[\![t_{rn}]\!]$ if p_{rg} is a program in $[\![c_{nf.1}[\![t_{rn}]\!]]\!]$, and $c_{nf.1}[\![t_{rn}]\!] \to c_{nf.2}[\![t_{rn}]\!]$. A program p_{rg} executes (initiates) t_{rn} if p_{rg} is executed in $[\![t_{rn}]\!]$.

An element v_l is a value in $[\![p_{rg}, t_{rn}]\!]$ if p_{rg} executes $[\![t_{rn}]\!]$, and v_l is a value in $[\![t_{rn}]\!]$. A program p_{rg} returns v_l in $[\![t_{rn}]\!]$ if v_l is a value in $[\![p_{rg}, t_{rn}]\!]$. A program p_{rg} returns v_l in $[\![c_{nf}]\!]$ if there exists t_{rn} such that p_{rg} returns v_l in $[\![t_{rn}]\!]$, and $c_{nf} = c_{nf.1}[\![t_{rn}]\!]$.

A program p_{rg} returns (or generates) an exception e_{xc} in $[\![t_{rn}]\!]$ if e_{xc} is a value in $[\![p_{rg}, t_{rn}]\!]$. A program p_{rg} is normally executed in $[\![t_{rn}]\!]$ if p_{rg} is executed in $[\![t_{rn}]\!]$, and t_{rn} is normally executed.

An element e_l is executed in $[\![t_{rn}]\!]$ if there exist p_{rg} such that p_{rg} is executed in $[\![t_{rn}]\!]$, and $e_l = [p_{rg}\ .\ 1]$. An element e_l executes (initiates) t_{rn} if e_l is executed in $[\![t_{rn}]\!]$.

An element v_l is a value in $[\![e_l, t_{rn}]\!]$ if e_l is executed in $[\![t_{rn}]\!]$, and v_l is a value in $[\![t_{rn}]\!]$. An element v_l returns v_l in $[\![t_{rn}]\!]$ if v_l is a value in $[\![v_l, t_{rn}]\!]$. An element v_l returns v_l in $[\![c_{nf}]\!]$ if there exists t_{rn} such that v_l returns v_l in $[\![t_{rn}]\!]$, and $c_{nf} = c_{nf.1}[\![t_{rn}]\!]$.

An element e_l returns (or generates) an exception e_{xc} in $[\![t_{rn}]\!]$ if e_{xc} is a value in $[\![e_l, t_{rn}]\!]$. An element e_l is normally executed in $[\![t_{rn}]\!]$ if e_l is executed in $[\![t_{rn}]\!]$, and t_{rn} is normally executed.

3.8. Safe configurations, transitions, programs and elements

A configuration c_{nf} is locally safe if $v_l[\![c_{nf}]\!] \neq und$.

A transition t_{rn} is safe if $c_{nf.1}[\![t_{rn}]\!]$ and $c_{nf.2}[\![t_{rn}]\!]$ are locally safe.

A configuration c_{nf} is safe if there is no $c_{nf.1}$ such that $c_{nf} \to^* c_{nf.1}$ and $c_{nf.1}$ is not locally safe.

A program p_{rg} is safe in $[\![c_{nf}]\!]$ if p_{rg} is a program in $[\![c_{nf}]\!]$, and c_{nf} is safe. A program p_{rg} is safe if p_{rg} is safe in $[\![c_{nf}]\!]$ for each c_{nf}.

An element e_l is safe in $[\![c_{nf}]\!]$ if $e_l = [p_{rg}[\![c_{nf}]\!]\ .\ 1]$, and p_{rg} is safe in $[\![c_{nf}]\!]$. An element e_l is safe if e_l is safe in $[\![c_{nf}]\!]$ for each c_{nf}.

4. The CTSL language

The CTSL language (Conceptual Transition System Language) is a basic language of CTSs. The CCSL language is a sublanguage of CTSL. Interpretable and executable elements of CTSL are called basic elements of CTSs.

Let $s_b \subseteq (x : x_0,\ y : y_0,\ z : z_0,\ u : u_0,\ v : v_0,\ w : w_0,\ x_1 : x_{1.0},\ ...,\ x_{n_t} : x_{n_t.0},\ conf :: in : c_{nf},\ val :: in : v_l[\![c_{nf}]\!])$.

4.1. Syntax of CCSL

CTSL is an extension of CCSL. Therefore, atoms, elements, conceptual states, conceptual configurations, pattern specifications and element definitions are represented in CTSL as in CCSL.

The element $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ then\ b_d)\ ::\ name\ ::\ n_m$ in CCSL represents the transition rule $(p_t, (v_{r.*}), (v_{r.s.*}), b_d)$ with the name n_m.

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

4.2. The special forms for atomic exogenous transition relations, transition rules and atomic endogenous transition relations

In this section we define the special forms for atomic exogenous transition relations, transition rules and atomic endogenous transition relations used below.

The form $(transition\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ then\ f_n)\ ::\ name\ ::\ n_m$ denotes the atomic exogenous transition relation $(p_t, (v_{r.*}), (v_{r.s.*}), f_n)$ with the name n_m.

The objects $var\ (v_{r.*})$ and $seq\ (v_{r.s.*})$ in the form $(transition\ ...)$ can be omitted. The omitted objects correspond to $var\ ()$ and $seq\ ()$, respectively.

The form $(endogenous{-}transition\ f_n)\ ::\ name\ ::\ n_m$ denotes the atomic endogenous transition relation f_n with the name n_m.

Let $\{v_{r.*}\}$, $\{v_{r.s.*}\}$, $\{v_{r.*.1}\}$ and $\{v_{r.*.2}\}$ are pairwise disjoint, $\{v_{r.*.3}\} \subseteq \{v_{r.*}\} \cup \{v_{r.*.1}\} \cup \{v_{r.*.2}\}$, and $(e_{l.*}) \in \{(), und, abn\}$. The form $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ (v_{r.*.1})\ und\ (v_{r.*.2})\ val\ (v_{r.*.3})\ e_{l.*}\ where\ c_{nd}\ then\ b_d)$ called a rule form is defined as follows:

- $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ val\ (v_{r.*.3})\ e_{l.*}\ where\ c_{nd}\ then\ b_d)$ is a shortcut for $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ (v_{r.*.1})\ und\ (v_{r.*.2})\ val\ (v_{r.*.3})\ e_{l.*}\ then\ (if\ c_{nd}\ then\ b_d\ else\ und))$;
- $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ val\ (v_{r.*.3},\ v_r)\ e_{l.*}\ then\ b_d)$ is a shortcut for $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ val\ (v_{r.*.3})\ e_{l.*}\ then\ (let\ w\ be\ v_r\ in\ [subst\ (v_r :: * : w)\ b_d]))$, where w is a new element that does not occur in this definition;
- $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ val\ ()\ e_{l.*}\ then\ b_d)$ is a shortcut for $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ e_{l.*}\ then\ b_d)$;
- $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1},\ v_r)\ abn\ (v_{r.*.2})\ e_{l.*}\ then\ b_d)$ is a shortcut for $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ e_{l.*}\ then\ (if\ (v_r\ is\ undefined)\ then\ und\ else\ b_d))$;

- $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ ()\ abn\ (v_{r.*.2})\ e_{l.*}\ then\ b_d)$ is a shortcut for $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ (v_{r.*.2})\ e_{l.*}\ then\ b_d)$;
- $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ (v_{r.*.2},\ v_r)\ e_{l.*}\ then\ b_d)$ is a shortcut for $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ (v_{r.*.2})\ e_{l.*}\ then\ (if\ (v_r\ is\ abnormal)\ then\ v_r\ else\ b_d))$;
- $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ ()\ e_{l.*}\ then\ b_d)$ is a shortcut for $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ e_{l.*}\ then\ b_d)$;
- $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ then\ b_d)$ is a shortcut for $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ then\ (if\ (val :: in\ is\ undefined)\ then\ skip\ else\ b_d)$;
- $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ then\ b_d)$ is a shortcut for $(rule\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ then\ (if\ (val :: in\ is\ abnormal)\ then\ skip\ else\ b_d)$.

The element c_{nd} specifies the restriction on the values of the pattern variables. The undefined value is propagated through the variables of $v_{r.*.1}$. Abnormal values are propagated through the variables of $v_{r.*.2}$. The sequence $e_{l.*}$ specifies propagation of abnormal values depending on the value of $val :: in$. The undefined value is propagated when $e_{l.*} = und$. Abnormal values are propagated when $e_{l.*} = abn$. The special element $v_r :: *$ references to the value of element associated with the pattern variable v_r. A pattern variable is evaluated if the element associated with it is evaluated. Thus, the sequence $v_{r.*.3}$ contains evaluated pattern variables. A pattern variable is quoted if the element associated with it is not evaluated. Let $F_{rm.r}$ be a set of rule forms.

The objects $var\ (v_{r.*})$, $seq\ (v_{r.s.*})$, $und\ (v_{r.*.1})$, $abn\ (v_{r.*.2})$, $val\ (v_{r.*.3})$ and $where\ c_{nd}$ in the form $(rule\ ...)$ can be omitted. The omitted objects correspond to $var\ ()$, $seq\ ()$, $und\ ()$, $abn\ ()$, $val\ ()$ and $where\ true$, respectively.

5. Semantics of executable elements in CTSL

5.1. Element interpretation

The element $x :: value$ returning the interpretation of x is defined by the rule

$(rule\ \ x :: value\ \ var\ \ (x)\ \ abn\ \ then\ \ x :: value :: atm)$;

$(transition\ \ x :: value :: atm\ \ var\ \ (x)\ \ then\ \ f_n)$,

where $x_0 :: value :: atm\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [value\ x_0\ c_{nf}]\ \#\ c_{nf}$.

5.2. Abnormal elements operations

The element und is defined by the rule

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

The element e_{xc} is defined by the rule

$(rule\ \ x\ \ var\ \ (x)\ \ abn\ \ where\ \ (x\ \ is\ \ exception)\ \ then\ \ x :: q) :: name :: (\text{"@"},\ \ exception)$.

The rule satisfies the property: $n_m \prec_{[\![o_{rd.trn.ex}]\!]} (\text{"@"}, exception)$ for each n_m such that n_m is a name of an atomic exogenous transition relation or transition rule with the pattern distinct from v_r, where v_r is a variable of this pattern.

The element $e_l :: q$ is defined by the rule

$(rule\ \ x :: q\ \ var\ \ (x)\ \ abn\ \ then\ \ x :: q :: value)$.

The element e_l of the form $(catch :: u\ x\ y)$ called an undefined value handler is defined as follows:

$(transition\ \ (catch :: u\ \ x\ \ y)\ \ var\ \ (x)\ \ seq\ \ (y)\ \ then\ \ f_n)$,

where $(catch\ :: u\ x_0\ y_0),\ e_{l.*}\ \#\ v_l\ \#\ c_{nf} \rightarrow_{f_n,s_b} [subst\ (x_0\ :\ v_l)\ y_0],\ e_{l.*}\ \#\ true\ \#\ c_{nf}$. The elements x and y are called a variable and body in $[\![e_l]\!]$. The element e_l replaces all occurences of x in y by the current value, resets the current value to *true* and executes the modified body.

The element e_l of the form $(catch\ x\ y)$ called an exception handler is defined as follows:

$(rule\ \ (catch\ \ x\ \ y)\ \ var\ \ (x)\ \ seq\ \ (y)\ \ und\ \ then\ \ (catch :: u\ \ x\ \ y))$.

The elements x and y are called a variable and body in $[\![e_l]\!]$.

The element e_l of the form $(throw\ x)$ is defined by the rule

$(rule\ \ (throw\ \ x)\ \ var\ \ (x)\ \ val\ \ (x)\ \ abn\ \ then\ \ (throw\ \ x :: *) :: atm)$;

$(transition\ \ (throw\ \ x) :: atm\ \ var\ \ (x)\ \ then\ \ f_n)$,

where $(throw\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ x_0\ \#\ c_{nf}$. The element x is called a body in $[\![e_l]\!]$.

The deletion $(delete{-}exception\ x)$ of the exception of the type x is defined by the rule

$(rule\ \ (delete{-}exception\ \ x)\ \ var\ \ (x)\ \ und\ \ then\ \ (catch\ \ w$

$\ (if\ \ ((w\ \ is\ \ exception)\ \ and\ \ (((element\ \ in\ \ w)\ \ ..\ \ type)\ \ =\ \ x :: q))$

$\ \ then\ \ (throw\ \ true)\ \ else\ \ (throw\ \ w :: q))))$.

5.3. Statements

The element *skip* is defined as follows:

$(rule\ \ skip\ \ abn\ \ then\ \ skip :: atm)$;

$(transition\ \ skip :: atm\ \ then\ \ f_n)$,

where $skip :: atm, e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ c_{nf}$.

The sequential composition e_l of the form $(seq\ e_{l.*})$ is defined by the rule

$(rule\ \ (seq\ \ x)\ \ var\ \ (x)\ \ seq\ \ (x)\ \ then\ \ x)$

The elements of $e_{l.*}$ are called elements in $[\![e_l]\!]$ and $e_{l.*}$ is called a body in $[\![e_l]\!]$. The element e_l executes its elements sequentially from left to right.

The conditional element $(if\ x\ then\ y\ else\ z)$ is defined as follows:

$(rule\ \ (if\ \ x\ \ then\ \ y\ \ else\ \ z)\ \ var\ \ (x)\ \ seq\ \ (y,\ \ z)\ \ val\ \ (x)\ \ abn$

$then\ \ (if\ \ x :: *\ \ then\ \ y\ \ else\ \ z) :: atm)$;

$(transition\ \ (if\ \ x\ \ then\ \ y\ \ else\ \ z) :: atm\ \ var\ \ (x)\ \ seq\ \ (y,\ \ z)\ \ then\ \ f_n)$,

where $(if\ x_0\ then\ y_0\ else\ z_0) :: atm, e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} [if\ [x_0 \neq und]\ then\ y_0\ else\ z_0], e_{l.*}\ \#\ c_{nf}$.

The element $(if\ x\ then\ y)$ is a shortcut for $(if\ x\ then\ y\ else\ skip)$.

The conditional element $(if\ x\ then\ y\ elseif\ z\ then\ u\ ...\ else\ v)$ is defined as follows:

$(definition\ \ (if\ \ x\ \ then\ \ y\ \ elseif\ \ z)\ \ var\ \ (x)\ \ seq\ \ (y,\ \ z)\ \ abn$

$then\ \ (if\ \ x\ \ then\ \ y\ \ else\ \ (if\ \ z)))$.

The element e_l of the form $(let\ x\ be\ y\ in\ z)$ is defined as follows:

$(rule\ \ (let\ \ x\ \ be\ \ y\ \ in\ \ z)\ \ var\ \ (x)\ \ seq\ \ (y,\ \ z)\ \ abn\ \ then\ \ (let\ \ x\ \ be\ \ y\ \ in\ \ z) :: atm)$;

$(transition\ \ (let\ \ x\ \ be\ \ y\ \ in\ \ z) :: atm\ \ var\ \ (x)\ \ seq\ \ (y,\ \ z)\ \ then\ \ f_n)$,

where $(let\ x_0\ be\ y_0\ in\ z_0) :: atm, e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} y_0,\ (let\ x_0\ be{-}val{-}in\ z_0), e_{l.*}\ \#\ c_{nf}$. The elements x, y and z are called a substitution variable, substitution value and substitution body in $[\![e_l]\!]$.

The auxiliary element $(let\ x\ be{-}val{-}in\ y)$ is defined as follows:

$(transition\ \ (let\ \ x\ \ be{-}val{-}in\ \ y)\ \ var\ \ (x)\ \ seq\ \ (y)\ \ abn\ \ then\ \ f_n)$,

where $(let\ x_0\ be{-}val{-}in\ y_0), e_{l.*}\ \#\ v_l\ \#\ c_{nf} \rightarrow_{f_n,s_b} [subst\ (x_0 : v_l)\ y_0], e_{l.*}\ \#\ c_{nf}$.

The element e_l of the form $(let\ ::\ seq\ x\ be\ y\ in\ z)$, where $x \in E_{l.(*)}$, $y \in E_{l.(*)}$, and $[len\ x] = [len\ y]$, is defined by the rule

$(rule\ \ (let :: seq\ \ x,\ \ y\ \ be\ \ (z),\ \ u\ \ in\ \ v)\ \ var\ \ (x)\ \ seq\ \ (y,\ \ z,\ \ u,\ \ v)\ \ abn$

$then\ \ (let\ \ x\ \ be\ \ z\ \ in\ \ (let :: seq\ \ y\ \ be\ \ u\ \ in\ \ v)))$;

$(rule\ \ (let :: seq\ \ be\ \ in\ \ v)\ \ seq\ \ (v)\ \ abn\ \ then\ \ v)$.

The elements x, y and z are called a substitution variables specification, substitution values specification and substitution body in $[\![e_l]\!]$. The elements of x and y are called substitution variables and substitution values in $[\![e_l]\!]$.

The iterator e_l of the form $(while\ x\ do\ y)$ is defined by the rule

$(if\ \ (while\ \ x\ \ do\ \ y)\ \ var\ \ (x)\ \ seq\ \ (y)\ \ abn\ \ then\ \ (if\ \ x\ \ then\ \ y\ \ (while\ \ x\ \ do\ \ y)))$.

The elements x and y are called a condition and body in $[\![e_l]\!]$.

The iterator e_l of the form $(foreach\ x\ in\ y\ do\ z)$ is defined as follows:

$(rule\ (foreach\ x\ in\ y\ do\ z)\ var\ (x,\ y)\ seq\ (z)\ val\ (y)\ abn\ where\ (y :: *\ is\ sequence)$
$\ then\ (foreach1\ x\ in\ y :: *\ do\ z))$.

The objects x, y and z are called an iteration variable, iteration structure specifier and body in $[\![e_l]\!]$. The element e_l executes sequentially z for values of x from $e_{l.1}$, where $e_{l.1}$ is the value of y.

The element $(foreach1\ x\ in\ y\ do\ z)$ is defined by the rules

$(rule\ (foreach1\ x\ in\ ()\ do\ y)\ var\ (x)\ seq\ (y)\ abn\ then)$;

$(rule\ (foreach1\ x\ in\ (y\ z)\ do\ v)\ var\ (x,\ y)\ seq\ (z,\ v)\ abn$
$\ then\ (let\ x\ be\ y\ in\ v),\ (foreach1\ x\ in\ (z)\ do\ v))$.

5.4. Characteristic functions for defined concepts

An object $d_{f.c}$ is a concept definition if $d_{f.c}$ is an atomic transition relation of the form $(transition\ n_m\ if\ (e_{l.1}\ is\ e_{l.2})\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ then\ f_n)$, or $d_{f.c}$ is a transition rule of the form $(rule\ n_m\ if\ (e_{l.1}\ is\ e_{l.2})\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ then\ b_d)$. Concept definitions specify concepts and their instances. Concepts specified by them are called defined concepts. The elements $e_{l.1}$ and $e_{l.2}$ are called an instance pattern and concept pattern in $[\![d_{f.c}]\!]$. The element $(e_{l.1}\ is\ e_{l.2})$ is called a characteristic function in $[\![d_{f.c}]\!]$. Let $D_{f.c}$ be a set of concept definitions.

An element $c_{ncp.d}$ is a defined concept in $[\![d_{f.c}, s_b]\!]$ if c_{ncp} is an instance in $[\![(e_{l.2}, var\ (v_{r.*})\ seq\ (v_{r.s.*})), m_t, s_b]\!]$. An element $c_{ncp.d}$ is a defined concept in $[\![d_{f.c}]\!]$ if there exists s_b such that $c_{ncp.d}$ is a concept in $[\![d_{f.c}, s_b]\!]$. An element $c_{ncp.d}$ is a defined concept in $[\![c_{nf}]\!]$ if there exists $d_{f.c}[\![c_{nf}]\!]$ such that $c_{ncp.d}$ is a concept in $[\![d_{f.c}]\!]$. Let $C_{ncp.d}$ be a set of defined concepts.

An element i_{nstn} is an instance in $[\![d_{f.c}, s_b]\!]$ if i_{nstn} is an instance in $[\![(e_{l.1}, var\ (v_{r.*})\ seq\ (v_{r.s.*})), m_t, s_b]\!]$. An element i_{nstn} is an instance in $[\![d_{f.c}]\!]$ if there exists s_b such that $c_{ncp.d}$ is an instance in $[\![d_{f.c}, s_b]\!]$.

An element i_{nstn} is an instance in $[\![c_{ncp.d}, c_{nf}, d_{f.c}]\!]$ if i_{nstn} is an instance in $[\![d_{f.c}]\!]$, $c_{ncp.d}$ is a defined concept in $[\![d_{f.c}]\!]$, and there exist $c_{nf.1}$ and v_l such that $(execute{-}exogenous{-}transition, (i_{nstn}\ is\ c_{ncp.d}), (n_m))\ \#\ c_{nf} \to^* \#\ v_l\ \#\ c_{nf.1}$, and $v_l \neq und$. An element i_{nstn} is an instance in $[\![c_{ncp.d}, c_{nf}]\!]$ if there exists $d_{f.c}$ such that i_{nstn} is an instance in $[\![c_{ncp.d}, c_{nf}, d_{f.c}]\!]$. An element $c_{ncp.d}$ is an instance in $[\![c_{nf}]\!]$ if there exists $c_{ncp.d}$ such that i_{nstn} is an instance in $[\![c_{ncp.d}, c_{nf}]\!]$. Let I_{nstn} be a set of instances.

A set s_t is called a content in $[\![c_{ncp.d}, c_{nf}]\!]$ if s_t is a set of all i_{nstn} such that i_{nstn} is an instance in $[\![c_{ncp.d}, c_{nf}]\!]$. Let $[content\ c_{ncp.d}\ c_{nf}]$ denote the content in $[\![c_{ncp.d}, c_{nf}]\!]$.

The notion of defined concepts is extended to the rules of the form $(rule\ (e_{l.1}\ is\ e_{l.2})\ var\ (v_{r.*})$ $seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ val\ (v_{r.*.3})\ where\ c_{nd}\ then\ b_d)$. Let r_l have this form. An element $c_{ncp.d}$ is a defined concept in $[\![r_l, s_b]\!]$ if $c_{ncp.d}$ is a defined concept in $[\![r_{l.1}, s_b]\!]$, where $r_{l.1}$ is a rule of the form $(rule\ (e_{l.1}\ is\ e_{l.2})\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ then\ b_{d.1})$ such that r_l is reduced to $r_{l.1}$.

The element $(x\ is\ atom)$ specifying that x is an atom is defined by the rule
$(rule\ \ (x\ \ is\ \ atom)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ atom) :: value)$.

The element $(x\ is\ update)$ specifying that x is an element update is defined by the rule
$(rule\ \ (x\ \ is\ \ update)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ update) :: value)$.

The element $(x\ is\ multi{-}attribute)$ specifying that x is a multi-attribute element is defined by the rule
$(rule\ \ (x\ \ is\ \ multi{-}attribute)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ multi{-}attribute) :: value)$.

The element $(x\ is\ attribute)$ specifying that x is an attribute element is defined by the rule
$(rule\ \ (x\ \ is\ \ attribute)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ attribute) :: value)$.

The element $(x\ is\ sorted)$ specifying that x is a sorted element is defined by the rule
$(rule\ \ (x\ \ is\ \ sorted)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ sorted) :: value)$.

The element $(x\ is\ undefined)$ specifying that x equals und is defined by the rule
$(rule\ \ (x\ \ is\ \ undefined)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ undefined) :: value)$.

The element $(x\ is\ defined)$ specifying that x does not equal und is defined by the rule
$(rule\ \ (x\ \ is\ \ defined)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ defined) :: value)$.

The element $(x\ is\ exception)$ specifying that x is an exception is defined by the rule
$(rule\ \ (x\ \ is\ \ exception)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ exception) :: value)$.

The element $(x\ is\ normal)$ specifying that x is a normal element is defined by the rule
$(rule\ \ (x\ \ is\ \ normal)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ normal) :: value$.

The element $(x\ is\ normal)$ specifying that x is an abnormal element is defined by the rule
$(rule\ \ (x\ \ is\ \ abnormal)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ abnormal) :: value$.

The element $(x\ is\ sequence)$ specifying that x is a sequence element is defined by the rule
$(rule\ \ (x\ \ is\ \ sequence)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ sequence) :: value)$.

The element $(x\ is\ set)$ specifying that the elements of the sequence element x are pairwise distinct is defined as follows:
$(rule\ \ (x\ \ is\ \ set)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ set) :: value)$.

The element $(x\ is\ empty)$ specifying that x is an empty element is defined by the rule

$(rule\ \ (x\ \ is\ \ empty)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ empty) :: value)$.

The element $(x\ is\ nonempty)$ specifying that x is not an empty element is defined by the rule

$(rule\ \ (x\ \ is\ \ nonempty)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ nonempty) :: value)$.

The element $(x\ is\ conceptual)$ specifying that x is a conceptual is defined by the rule

$(rule\ \ (x\ \ is\ \ conceptual)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ conceptual) :: value)$.

The element $(x\ is\ (conceptual\ in\ y))$ specifying that x is a conceptual in the context of the state y is defined by the rule

$(rule\ \ (x\ \ is\ \ (conceptual\ \ in\ \ y))\ \ var\ \ (x,\ \ y)\ \ abn\ \ then\ \ (x\ \ is\ \ (conceptual\ \ in\ \ y)) :: value$.

The element $(x\ is\ state)$ specifying that x is a conceptual state is defined by the rule

$(rule\ \ (x\ \ is\ \ state)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ state) :: value)$.

The element $(x\ is\ configuration)$ specifying that x is a conceptual configuration is defined by the rule

$(rule\ \ (x\ \ is\ \ configuration)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ configuration) :: value)$.

The element $(x\ is\ nat)$ specifying that x is a natural number is defined by the rule

$(rule\ \ (x\ \ is\ \ nat)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ nat) :: value)$.

The element $(x\ is\ nat0)$ specifying that x is either a natural number, or a zero is defined by the rule

$(rule\ \ (x\ \ is\ \ nat0)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ nat0) :: value)$.

The element $(x\ is\ int)$ specifying that x is an integer is defined by the rule

$(rule\ \ (x\ \ is\ \ int)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ int) :: value)$.

The element $(x\ is\ (satisfiable\ in\ y))$ specifying that x is satisfiable in the context of variables y is defined by the rule

$(rule\ \ (x\ \ is\ \ (satisfiable\ \ in\ \ y))\ \ var\ \ (x)\ \ seq\ \ (y)\ \ abn$
$\ then\ \ (x\ \ is\ \ (satisfiable\ \ in\ \ (y))) :: value)$.

The element $(x\ is\ (valid\ in\ y))$ specifying that x is valid in the context of variables y is defined by the rule

$(rule\ \ (x\ \ is\ \ (valid\ \ in\ \ y))\ \ var\ \ (x)\ \ seq\ \ (y)\ \ abn\ \ then\ \ (x\ \ is\ \ (valid\ \ in\ \ (y))) :: value)$.

The element $(x\ is\ (sequence\ y))$ specifying that x is a sequence element such that the value in $[\![(e_l\ is\ y)]\!]$ does not equal und for each element e_l of x is defined by the rule

$(rule\ \ ((x\ \ y)\ \ is\ \ (sequence\ \ z))\ \ var\ \ (x,\ \ z)\ \ seq\ \ (y)\ \ abn$

$then\ ((x\ is\ z)\ and\ ((y)\ is\ (sequence\ z)))$;

$(rule\ (()\ is\ (sequence\ x))\ var\ (x)\ abn\ then\ true)$.

The element $(x\ is\ rule)$ specifying that x is a rule is defined as follows:

$(rule\ (x\ is\ rule)\ var\ (x)\ abn\ then\ (x\ is\ rule) :: value)$;

$(interpretation\ (x\ is\ rule)\ var\ (x)\ then\ f_n)$,

where $[f_n\ s_b] = [if\ [x_0 \in R_l]\ then\ true\ else\ und]$.

The element $(x\ is\ (rule\ in\ y))$ specifying that x is a rule in the context of the state y is defined as follows:

$(rule\ (x\ is\ (rule\ in\ y))\ var\ (x,\ y)\ abn\ then\ (x\ is\ (rule\ in\ y)) :: value)$;

$(definition\ (x\ is\ (rule\ in\ y))\ var\ (x,\ y)\ where\ ((x\ is\ rule)\ and\ (y\ is\ state))$

$then\ (x\ is\ conceptual\ in\ y) :: atm)$;

$(interpretation\ (x\ is\ (conceptual\ in\ y)) :: atm\ var\ (x,\ y)\ then\ f_n)$,

where $[f_n\ s_b] = [if\ [x_0 \in R_l[\![y_0]\!]]\ then\ true\ else\ und]$.

The element $(x\ is\ transition)$ specifying that x is a transition is defined as follows:

$(rule\ (x\ is\ transition)\ var\ (x)\ abn\ then\ (x\ is\ transition) :: value)$;

$(interpretation\ (x\ is\ transition)\ var\ (x)\ then\ f_n)$,

where $[f_n\ s_b] = [if\ [x_0 \in T_{rn}]\ then\ true\ else\ und]$.

5.5. Elements operations

The element () is defined by the rule

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

The element $(len\ x)$ specifying the length of the element x is defined by the rule

$(rule\ (len\ x)\ var\ (x)\ val\ (x)\ abn\ then\ (len\ x :: * :: q) :: value)$.

The element $(x\ =\ y)$ specifying the equality of the elements x and y is defined by the rule

$(rule\ (x\ =\ y)\ var\ (x,\ y)\ val\ (x,\ y)\ abn\ then\ (x :: * :: q\ =\ y :: * :: q) :: value)$.

The element $(x\ !=\ y)$ specifying the inequality of the elements x and y is defined in the similar way.

The element $(x\ .\ y)$ specifying the y-th element of the sequence element x is defined by the rule

$(rule\ (x\ .\ y)\ var\ (x,\ y)\ val\ (x,\ y)\ abn\ then\ (x :: * :: q\ .\ y :: * :: q) :: value)$.

The element $(x\ ..\ y)$ specifying the value of the attribute element x for the attribute y is defined by the rule

$(rule\ \ (x\ \ ..\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x)\ \ abn\ \ then\ \ (x :: * :: q\ \ ..\ \ y) :: value)$.

The element $(x\ \ +\ \ y)$ specifying the concatenation of the sequence elements x and y is defined by the rule

$(rule\ \ (x\ \ +\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x,\ \ y)\ \ abn\ \ then\ \ (x :: * :: q\ \ +\ \ y :: * :: q) :: value)$.

The element $(x\ .+\ \ y)$ specifying the addition of the element x to the head of the sequence element y is defined by the rule

$(rule\ \ (x\ \ .+\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x,\ \ y)\ \ abn\ \ then\ \ (x :: * :: q\ \ .+\ \ y :: * :: q) :: value)$.

The element $(x\ \ .+\ ::\ set\ y)$ specifying the addition of the element x to the head of the sequence element y representing a set is defined as follows:

$(rule\ \ (x\ \ .+ :: set\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x,\ \ y)\ \ abn\ \ where\ \ (y :: *\ \ is\ \ set)$
$\ \ then\ \ (x :: * :: q\ \ .+ :: set\ \ y :: * :: q) :: value)$.

The element $(x\ \ +.\ y)$ specifying the addition of the element y to the tail of the sequence element x is defined by the rule

$(rule\ \ (x\ \ +.\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x,\ \ y)\ \ abn\ \ then\ \ (x :: * :: q\ \ +.\ \ y :: * :: q) :: value)$.

The element $(x\ \ +.\ ::\ set\ y)$ specifying the addition of the element y to the tail of the sequence element x representing a set is defined by the rule

$(rule\ \ (x\ \ +. :: set\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x,\ \ y)\ \ abn\ \ where\ \ (x :: *\ \ is\ \ set)$
$\ \ then\ \ (x :: * :: q\ \ +. :: set\ \ y :: * :: q) :: abn)$.

The element $(x\ \ -.\ ::\ set\ y)$ specifying the deletion of the element y from the sequence element x representing a set is defined by the rule

$(rule\ \ (x\ \ -. :: set\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x,\ \ y)\ \ abn\ \ where\ \ (x :: *\ \ is\ \ set)$
$\ \ then\ \ (x :: * :: q\ \ -. :: set\ \ y :: * :: q) :: value)$.

The element $(upd\ x\ y_1 : z_1, ..., y_{n_t} : z_{n_t})$ specifying the sequential updates of the attribute element x at the points y_1, ..., y_{n_t} by z_1, ..., z_{n_t} is defined by the rules

$(rule\ \ (upd\ \ x\ \ y)\ \ var\ \ (x)\ \ seq\ \ (y)\ \ val\ \ (x)\ \ abn$
$\ \ where\ \ ((x :: *\ \ is\ \ attribute)\ \ and\ \ ((y)\ \ is\ \ (sequence\ \ update)))\ \ then\ \ (upd :: att\ \ x :: *\ \ y))$;
$(rule\ \ (upd :: att\ \ x\ \ y\ \ z)\ \ var\ \ (y)\ \ seq\ \ (z)\ \ und\ \ (x)\ \ abn$
$\ \ then\ \ (upd :: att\ \ (upd1 :: att\ \ x\ \ y)\ \ z))$;
$(rule\ \ (upd :: att\ \ x)\ \ var\ \ (x)\ \ then\ \ x)$;
$(rule\ \ (upd1 :: att\ \ x\ \ y : z)\ \ var\ \ (x,\ \ y,\ \ z)\ \ val\ \ (z)\ \ abn$
$\ \ then\ \ (upd1 :: att\ \ x\ \ y : z :: * :: q) :: value)$.

The element $(upd\ x\ y : z)$ specifying the update of the sequence element x at the index y by z is defined by the rule

$(rule\ \ (upd\ \ x\ \ y\ \ z)\ \ var\ \ (x,\ \ y,\ \ z)\ \ val\ \ (x,\ \ y,\ \ z)\ \ abn$

$then\ \ (upd :: seq\ \ x :: * :: q\ \ y :: * :: q\ \ z :: * :: q) :: value)$.

The element $(x\ in :: set\ y)$ specifying that x is an element of the sequence element y is defined as follows:

$(rule\ \ (x\ \ in :: set\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x,\ \ y)\ \ abn\ \ then\ \ (x\ \ in :: set\ \ y) :: value)$.

The element $(x\ includes :: set\ y)$ specifying that the sequence element x includes the elements of the sequence element y is defined as follows:

$(rule\ \ (x\ \ includes :: set\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x,\ \ y)\ \ abn\ \ then\ \ (x\ \ includes :: set\ \ y) :: value)$.

The element $(attributes\ in\ x)$ specifying the sequence of attributes of the attribute element x is defined by the rule

$(rule\ \ (attributes\ \ in\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (attributes\ \ in\ \ x) :: value)$.

The element $(values\ in\ x)$ specifying the sequence of attribute values of the attribute element x is defined by the rule

$(rule\ \ (values\ \ in\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (values\ \ in\ \ x) :: value)$.

The element $(element\ in\ x)$ specifying the element of the sorted element x is defined by the rule

$(rule\ \ (element\ \ in\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (element\ \ in\ \ x) :: value)$.

The element $(sort\ in\ x)$ specifying the sort of the sorted element x is defined by the rule

$(rule\ \ (sort\ \ in\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (sort\ \ in\ \ x) :: value)$.

The element $(attribute\ in\ x)$ specifying the attribute of the element update x is defined by the rule

$(rule\ \ (attribute\ \ in\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (attribute\ \ in\ \ x) :: value)$.

The element $(value\ in\ x)$ specifying the value of the element update x is defined by the rule

$(rule\ \ (value\ \ in\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (value\ \ in\ \ x) :: value)$.

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

$(rule\ \ (unbracket\ \ (x))\ \ seq\ \ (x)\ \ abn\ \ then\ \ x)$.

5.6. Boolean operations

The element $true$ is defined by the rule:

$(rule\ \ true\ \ abn\ \ then\ \ true :: value)$.

The element $(x\ and\ y)$ specifying the conjunction of x and y is defined by the rule:

$(rule\ \ (x\ \ and\ \ y)\ \ var\ \ (x,\ \ y)\ \ abn\ \ then\ \ (if\ \ x\ \ then\ \ y\ \ else\ \ und))$.

The elements $(x\ o_p\ y)$, where $o_p \in \{or, =>, <=>\}$ specifying the disjunction, implication and equivalence of x and y are defined in the similar way.

The element $(x_1\ and\ x_2\ and\ ...\ and\ x_{n_t})$ specifying the conjunction of x_1, x_2, ..., x_{n_t} is defined by the rule

$(rule\ \ (x\ \ and\ \ y\ \ and\ \ z)\ \ var\ \ (x,\ \ y)\ \ seq\ \ (z)\ \ abn\ \ then\ \ ((x\ \ and\ \ y)\ \ and\ \ z)$.

The element $(x_1\ or\ x_2\ or\ ...\ or\ x_{n_t})$ specifying the disjunction of x_1, x_2, ..., x_{n_t} is defined in the similar way.

The element $(not\ x)$ specifying the negation of x is defined by the rule

$(rule\ \ (not\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (if\ \ x\ \ then\ \ und\ \ else\ \ true))$.

5.7. Integers

The element i_{nt} is defined by the rule

$(rule\ \ x\ \ var\ \ (x)\ \ abn\ \ where\ \ (x\ \ is\ \ int)\ \ then\ \ x :: q) :: name :: ("@",\ \ int)$.

The rule satisfies the property: $("@", exception) \prec_{[\![o_{rd.trn.ex}]\!]} ("@", int)$.

The element $(x\ +\ y)$ specifying the sum of x and y is defined by the rule

$(rule\ \ (x\ \ +\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x,\ \ y)\ \ abn\ \ then\ \ (x :: * :: q\ \ +\ \ y :: * :: q) :: value)$.

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

The element $(x\ <\ y)$ specifying that x is less than y is defined by the rule

$(rule\ \ (x\ \ <\ \ y)\ \ var\ \ (x,\ \ y)\ \ val\ \ (x,\ \ y)\ \ abn\ \ then\ \ (x :: * :: q\ \ <\ \ y :: * :: q) :: value)$.

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

5.8. Conceptuals operations

The element $(x\ in\ y)$ specifying the value of the conceptual x in the state y is defined by the rule

$(rule\ \ (x\ \ in\ \ y)\ \ var\ \ (x,\ \ y)\ \ abn\ \ then\ \ (x\ \ in\ \ y) :: value)$.

The element $x :: state :: y$ specifying the value of the conceptual x in the substate with the name y of the current configuration is defined by the rule

$(rule\ \ x :: state :: y\ \ var\ \ (x,\ \ y)\ \ abn\ \ then\ \ (x :: state :: y) :: value)$.

The element c_{ncpl} is a shortcut for $c_{ncpl} :: ()$.

The assignment $(c_{ncpl} :: state :: n_m \ ::= \ e_l)$ of e_l to $c_{ncpl} :: state :: n_m$ is defined as follows:
$(rule \ (x :: state :: z \ ::= \ y) \ var \ (x, \ y, \ z) \ val \ (y) \ abn \ where \ (x \ is \ conceptual)$
$then \ (x :: state :: z \ ::= \ y :: *) :: atm)$;
$(transition \ (x :: state :: z \ ::= \ y) :: atm \ var \ (x, \ y, \ z) \ then \ f_n)$,
where $(x_0 :: state :: z_0 ::= y_0) :: atm, e_{l.*} \ \# \ c_{nf} \to_{f_n, s_b} e_{l.*} \ \# \ [[c_{nf} \ z_0] \ x_0 : y_0]$.

The element $(c_{ncpl} \ ::= \ e_l)$ is a shortcut for $(c_{ncpl} :: () \ ::= \ e_l)$. The elements $(c_{ncpl} :: state :: n_m \ ::=)$ and $(c_{ncpl} \ ::=)$ are shortcuts for $(c_{ncpl} :: state :: n_m \ ::= \ und)$ and $(c_{ncpl} \ ::= \ und)$.

5.9. Countable concepts operations

A normal element $c_{ncp.c}$ is a countable concept in $[\![c_{nf}]\!]$ if $[[c_{nf} \ countable\!-\!concept] \ (0 : c_{ncp.c})] \in N_t$. Thus, the substate $countable\!-\!concept$ specifies countable concepts. Let $C_{ncp.c}$ be a set of countable concepts. The element $[[c_{nf} \ countable\!-\!concept] \ (0 : c_{ncp.c})]$ is called an order in $[\![c_{ncp.c}, c_{nf}]\!]$. Let $O_{rd.cncp.c}$ be a set of orders of countable concepts. An element $n_t :: cc :: c_{ncp.c}$ is called an instance in $[\![c_{ncp.c}]\!]$. An element $n_t :: cc :: c_{ncp.c}$ is an instance in $[\![c_{ncp.c}, c_{nf}]\!]$ if $n_t \le o_{rd.cncp.c}[\![c_{ncp.c}, c_{nf}]\!]$.

The element $(x \ is \ countable\!-\!concept)$ specifying that x is a countable concept is defined as follows:
$(rule \ (x \ is \ countable\!-\!concept) \ var \ (x) \ abn \ then \ (x \ is \ countable\!-\!concept) :: value)$.

The element $n_t :: cc :: c_{ncp.c}$ is defined by the rule:
$(rule \ x :: cc :: y \ var \ (x, \ y) \ abn \ then \ x :: cc :: y :: value)$.

Let c_{ncpl} denote $(0 : x) :: countable\!-\!concept$. The element $(new \ x)$ called an instance generator generates a new instance of the countable concept x and adds this concept if it was not. It is defined as follows:
$(rule \ (new \ x) \ var \ (x) \ abn \ then \ (new \ x) :: atm)$;
$(transition \ (new \ x) :: atm \ var \ (x) \ then \ f_n)$,
where $(new \ x_0) :: atm, e_{l.*} \ \# \ c_{nf} \to_{f_n, s_b} (let \ w \ be \ c_{ncpl} \ in \ (if \ (w \ is \ int) \ then \ (seq \ (c_{ncpl} \ ::= (w+1)), \ (let \ w1 \ be \ (w+1) \ in \ w1 :: x :: cc)) \ else \ (seq \ (c_{ncpl} \ ::= \ 1), \ 1 :: x :: cc))), \ e_{l.*} \ \# \ c_{nf}$.

5.10. Matching operations

The conditional pattern matching element e_l of the form $(if \ x \ matches \ y \ var \ z \ seq \ u \ then \ v \ else \ w)$, where (y, z, u) is a pattern specification, is defined as follows:

$(rule\ (if\ x\ matches\ y\ var\ z\ seq\ u\ then\ v\ else\ w)\ var\ (x,\ y,\ z,\ u)\ seq\ (v,\ w)\ abn$
$where\ ((z\ is\ sequence)\ and\ (u\ is\ sequence)\ and\ (z\ includes :: set\ u))$
$then\ (if\ x\ matches\ y\ var\ z\ seq\ u\ then\ v\ else\ w) :: atm)$**;**
$(transition\ (if\ x\ matches\ y\ var\ z\ seq\ u\ then\ v\ else\ w) :: atm$
$var\ (x,\ y,\ z,\ u,\ v,\ w)\ then\ f_n)$**,**

where $(if\ x_0\ matches\ y_0\ var\ z_0\ seq\ u_0\ then\ v_0\ else\ w_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \to_{f_n,s_b} [if\ [x_0$ is an instance in $[\![(y_0, z_0, u_0), m_t, s_{b.1}]\!]$ for some $s_{b.1}]$ $then\ [subst\ s_{b.1} \cup (conf :: in : c_{nf}, val :: in : v_l[\![c_{nf}]\!])\ v_0]\ else\ [subst\ (conf :: in : c_{nf}, val :: in : v_l[\![c_{nf}]\!])\ w_0],\ e_{l.*}\ \#\ c_{nf}$. The objects x, y, z, u, v and w are called a matched element, pattern, variable specification, sequence variable specification, *then*-branch and *else*-branch in $[\![e_l]\!]$. The elements of z are called pattern variables in $[\![e_l]\!]$. The element e_l executes the instance of the *then*-branch v in $[\![s_{b.1}]\!]$ if x is an instance in $[\![y, s_{b.1}]\!]$. Otherwise, the element e_l executes the *else*-branch w.

Let $\{v_{r.*}\}$, $\{v_{r.s.*}\}$, $\{v_{r.*.1}\}$ and $\{v_{r.*.2}\}$ are pairwise disjoint, and $\{v_{r.*.3}\} \subseteq \{v_{r.*}\} \cup \{v_{r.*.1}\} \cup \{v_{r.*.2}\}$. The form $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ (v_{r.*.1})\ und\ (v_{r.*.2})\ val\ (v_{r.*.3})\ where\ c_{nd}\ then\ e_{l.1}\ else\ e_{l.2})$ is defined as follows:

- $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ val\ (v_{r.*.3})\ where\ c_{nd}\ then\ e_{l.1}\ else\ e_{l.2})$ is a shortcut for $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ (v_{r.*.1})\ und\ (v_{r.*.2})\ val\ (v_{r.*.3})\ then\ (if\ c_{nd}\ then\ e_{l.1}\ else\ e_{l.2} :: (nosubstexcept\ conf :: in, val :: in))\ else\ e_{l.2})$;
- $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ val\ (v_{r.*.3},\ v_r)\ then\ e_{l.1}\ else\ e_{l.2})$ is a shortcut for $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ val\ (v_{r.*.3})\ then\ (let\ w\ be\ v_r\ in\ [subst\ (v_r :: * : w)\ e_{l.1}])\ else\ e_{l.2})$, where w is a new element that does not occur in this definition;
- $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ val\ ()\ then\ e_{l.1}\ else\ e_{l.2})$ is a shortcut for $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ then\ e_{l.1}\ else\ e_{l.2})$;
- $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1},\ v_r)\ abn\ (v_{r.*.2})\ then\ b_d)$ is a shortcut for $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ (v_{r.*.1})\ abn\ (v_{r.*.2})\ then\ (if\ (v_r\ is\ undefined)\ then\ und\ else\ e_{l.1})\ else\ e_{l.2})$;
- $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ und\ ()\ abn\ (v_{r.*.2})\ then\ e_{l.1}\ else\ e_{l.2})$ is a shortcut for $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ (v_{r.*.2})\ then\ e_{l.1}\ else\ e_{l.2})$;
- $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ (v_{r.*.2},\ v_r)\ then\ e_{l.1}\ else\ e_{l.2})$ is a shortcut for $(if\ e_l\ matches\ p_t\ var\ (v_{r.*})\ seq\ (v_{r.s.*})\ abn\ (v_{r.*.2})\ then\ (if\ (v_r\ is\ abnormal)\ then\ v_r\ else$

$e_{l.1})$ $else$ $e_{l.2})$;

- $(if$ e_l $matches$ p_t var $(v_{r.*})$ seq $(v_{r.s.*})$ abn $()$ $then$ $e_{l.1}$ $else$ $e_{l.2})$ is a shortcut for $(if$ e_l $matches$ p_t var $(v_{r.*})$ seq $(v_{r.s.*})$ $then$ $e_{l.1}$ $else$ $e_{l.2})$.

The element c_{nd} specifies the restriction on the values of the pattern variables. The undefined value is propagated through the variables of $v_{r.*.1}$. Abnormal values are propagated through the variables of $v_{r.*.2}$. The special element $v_r :: *$ references to the value of element associated with the pattern variable v_r. A pattern variable is evaluated if the element associated with it is evaluated. Thus, the sequence $v_{r.*.3}$ contains evaluated pattern variables. A pattern variable is quoted if the element associated with it is not evaluated.

The objects var $(v_{r.*})$, seq $(v_{r.s.*})$, und $(v_{r.*.1})$, abn $(v_{r.*.2})$, val $(v_{r.*.3})$, $where$ c_{nd} and $else$ $e_{l.2}$ in this form can be omitted. The omitted objects correspond to var $()$, seq $()$, und $()$, abn $()$, val $()$, $where$ $true$ and $else$ $skip$, respectively.

The form $(e_l$ $matches$ p_t var $(v_{r.*})$ seq $(v_{r.s.*})$ und $(v_{r.*.1})$ abn $(v_{r.*.2})$ val $(v_{r.*.3})$ $where$ $c_{nd})$ is a shortcut for $(if$ e_l $matches$ p_t var $(v_{r.*})$ seq $(v_{r.s.*})$ und $(v_{r.*.1})$ abn $(v_{r.*.2})$ val $(v_{r.*.3})$ $where$ c_{nd} $then$ $true$ $else$ $und)$. The objects var $(v_{r.*})$, seq $(v_{r.s.*})$, und $(v_{r.*.1})$, abn $(v_{r.*.2})$, val $(v_{r.*.3})$ and $where$ c_{nd} in this form can be omitted. The omitted objects correspond to var $()$, seq $()$, und $()$, abn $()$, val $()$ and $where$ $true$, respectively.

5.11. Interpretations operations

The element $(x$ is $definition{-}form)$ specifying that x is a definition form is defined as follows:

$(rule$ $(x$ is $definition{-}form)$ var (x) abn $then$ $(x$ is $definition{-}form) :: value)$;

$(transition$ $(x$ is $definition{-}form)$ var (x) $then$ $f_n)$,

where $[f_n\ s_b] = [if\ [x_0 \in F_{rm.d}]\ then\ true\ else\ und]$.

The element $f_{rm.d} :: name :: n_m$ specifying a definition with the name n_m is defined as follows:

$(rule$ $x :: name :: y$ var $(x,$ $y)$ abn $where$ $(x$ is $definition{-}form)$
$then$ $x :: name :: y :: atm :: definition)$;

$(transition$ $x :: name :: y :: atm :: definition$ var $(x,$ $y)$ $then$ $f_n)$,

where

- if $y_0 \in [support\ [c_{nf}\ (0 : definitions) :: state :: interpretation]] \cup [support\ i_{ntr.a.s}]$, then $x_0 :: name :: y_0 :: atm :: definition,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ und\ \#\ c_{nf}$;

- if $y_0 \notin [support\ [c_{nf}\ (0 : definitions) :: state :: interpretation]] \cup [support\ i_{ntr.a.s}]$, and x_0 is reduced to d_f, then $x_0 :: name :: y_0 :: atm :: definition,\ e_{l.*} \# c_{nf} \rightarrow_{f_n,s_b} e_{l.*} \# [c_{nf}\ interpretation.(0 : definitions).y_0 : d_f]$.

The element $(add{-}interpretation\ x)$ adding the interpretation with the name x is defined as follows:

$(rule\ (add{-}interpretation\ x)\ var\ (x)\ abn\ then\ (add{-}interpretation\ x) :: atm)$;
$(transition\ (add{-}interpretation\ x) :: atm\ var\ (x)\ then\ f_n)$,

where

- if $x_0 \in [support\ [c_{nf}\ (0 : definitions) :: state :: interpretation]] \cup [support\ i_{ntr.a.s}]$, then $(add{-}interpretation\ x_0) :: atm,\ e_{l.*} \# c_{nf} \rightarrow_{f_n,s_b} e_{l.*} \# [c_{nf}\ interpretation.(0 : order) : [value\ [[c_{nf}\ (0 : order) :: state :: interpretation] :: q\ +. :: set\ x_0 :: q]\ c_{nf}]]$;
- if $x_0 \notin [support\ [c_{nf}\ (0 : definitions) :: state :: interpretation]] \cup [support\ i_{ntr.a.s}]$, then $(add{-}interpretation\ x_0) :: atm,\ e_{l.*} \# c_{nf} \rightarrow_{f_n,s_b} e_{l.*} \# und \# c_{nf}$.

The element $(add{-}interpretation\ x\ after\ y)$ adding the interpretation with the name x after the interpretation with the name y is defined as follows:

$(rule\ (add{-}interpretation\ x\ after\ y)\ var\ (x,\ y)\ abn$
$\ then\ (add{-}interpretation\ x\ after\ y) :: atm)$;
$(transition\ (add{-}interpretation\ x\ after\ y) :: atm\ var\ (x,\ y)\ then\ f_n)$,

where

- if $x_0 \in [support\ [c_{nf}\ (0 : definitions) :: state :: interpretation]] \cup [support\ i_{ntr.a.s}]$, and $y_0 \notin [c_{nf}\ (0 : order) :: state :: interpretation] :: q\ -. :: set\ x_0]$, then $(add{-}interpretation\ x_0) :: atm,\ e_{l.*} \# c_{nf} \rightarrow_{f_n,s_b} e_{l.*} \# und \# c_{nf}$;
- if $x_0 \in [support\ [c_{nf}\ (0 : definitions) :: state :: interpretation]] \cup [support\ i_{ntr.a.s}]$, and $[value\ [c_{nf}\ (0 : order) :: state :: interpretation] :: q\ -. :: set\ x_0] = n_{m.*.1}\ y_0\ n_{m.*.2}$, then $(add{-}interpretation\ x_0) :: atm,\ e_{l.*} \# c_{nf} \rightarrow_{f_n,s_b} e_{l.*} \# [c_{nf}\ interpretation.(0 : order) : n_{m.*.1}\ y_0\ x_0\ n_{m.*.2}]$;
- if $x_0 \notin [support\ [c_{nf}\ (0 : definitions) :: state :: interpretation]] \cup [support\ i_{ntr.a.s}]$, then $(add{-}interpretation\ x_0) :: atm,\ e_{l.*} \# c_{nf} \rightarrow_{f_n,s_b} e_{l.*} \# und \# c_{nf}$.

The element $(delete{-}interpretation\ x)$ deleting the interpretation with the name x is defined as follows:

$(rule\ (delete{-}interpretation\ x)\ var\ (x)\ abn\ then\ (delete{-}interpretation\ x) :: atm)$;
$(transition\ (delete{-}interpretation\ x) :: atm\ var\ (x)\ then\ f_n)$,

where

- if $x_0 \in [support\ [c_{nf}\ (0 : definitions) :: state :: interpretation]] \cup [support\ i_{ntr.a.s}]$, then $(delete{-}interpretation\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [c_{nf}\ interpretation.(0 : order) : [value\ [c_{nf}\ (0 : order) :: state :: transition] :: q\ -.:: set\ x_0 :: q\ c_{nf}]]$;
- if $x_0 \notin [support\ [c_{nf}\ (0 : definitions) :: state :: interpretation]] \cup [support\ i_{ntr.a.s}]$, then $(delete{-}interpretation\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ und\ \#\ c_{nf}$.

5.12. Configurations operations

The element $conf :: cur$ specifying the current configuration is defined as follows:

$(rule\ \ conf :: cur\ \ abn\ \ then\ \ conf :: cur :: value)$.

The element $val :: cur$ specifying the value in the current configuration is defined as follows:

$(rule\ \ val :: cur\ \ abn\ \ then\ \ val :: cur :: value)$;

$(definition\ \ val :: cur\ \ then\ \ val :: cur :: value)$;

$(interpretation\ \ val :: cur\ \ then\ \ f_n)$,

where $[f_n\ s_b] = v_l[\![c_{nf}]\!]$.

5.13. Transitions operations

The element $(x\ is\ rule{-}form)$ specifying that x is a rule form is defined as follows:

$(rule\ \ (x\ \ is\ \ rule{-}form)\ \ var\ \ (x)\ \ abn\ \ then\ \ (x\ \ is\ \ rule{-}form) :: value)$;

$(transition\ \ (x\ \ is\ \ rule{-}form)\ \ var\ \ (x)\ \ then\ \ f_n)$,

where $[f_n\ s_b] = [if\ [x_0 \in F_{rm.r}]\ then\ true\ else\ und]$.

The element $f_{rm.r} :: name :: n_m$ specifying a rule with the name n_m is defined as follows:

$(rule\ \ x :: name :: y\ \ var\ \ (x,\ \ y)\ \ abn\ \ where\ \ (x\ \ is\ \ rule{-}form)$

$\ then\ \ x :: name :: y :: atm :: rule)$;

$(transition\ \ x :: name :: y :: atm :: rule\ \ var\ \ (x,\ \ y)\ \ then\ \ f_n)$,

where

- if $y_0 \in [support\ [c_{nf}\ (0 : rules) :: state :: transition]] \cup [support\ t_{rn.rlt.ex.s}] \cup [support\ t_{rn.rlt.en.s}]$, then $x_0 :: name :: y_0 :: atm :: rule,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ und\ \#\ c_{nf}$;
- if $y_0 \notin [support\ [c_{nf}\ (0 : rules) :: state :: transition]] \cup [support\ t_{rn.rlt.ex.s}] \cup [support\ t_{rn.rlt.en.s}]$, and x_0 is reduced to r_l, then $x_0 :: name :: y_0 :: atm :: rule,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [c_{nf}\ transition.(0 : rules).y_0 : r_l]$.

The element $(add{-}transition\ x)$ adding the transition with the name x is defined as follows:

$(rule\ \ (add{-}transition\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (add{-}transition\ \ x) :: atm)$;

$(transition\ \ (add{-}transition\ \ x) :: atm\ \ var\ \ (x)\ \ then\ \ f_n)$,

where

- if $x_0 \in [support\ [c_{nf}\ (0 : rules) :: state :: transition]] \cup [support\ t_{rn.rlt.ex.s}]$, then $(add{-}transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [c_{nf}\ transition.(-1 : exogenous, 0 : order) : [value\ [[c_{nf}\ (-1 : exogenous, 0 : order) :: state :: transition] :: q\ \ +.\ :: set\ x_0 :: q]\ c_{nf}]]$;
- if $x_0 \in [support\ t_{rn.rlt.en.s}]$, then $(add{-}transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [c_{nf}\ transition.(-1 : endogenous, 0 : order) : [value\ [[c_{nf}\ (-1 : endogenous, 0 : order) :: state :: transition] :: q\ \ +.\ :: set\ x_0 :: q]\ c_{nf}]]$;
- if $x_0 \notin [support\ [c_{nf}\ (0 : rules) :: state :: transition]] \cup [support\ t_{rn.rlt.ex.s}] \cup [support\ t_{rn.rlt.en.s}]$, then $(add{-}transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ und\ \#\ c_{nf}$.

The element $(add{-}transition\ x\ after\ y)$ adding the transition with the name x after the transition with the name y is defined as follows:

$(rule\ \ (add{-}transition\ \ x\ \ after\ \ y)\ \ var\ \ (x,\ \ y)\ \ abn$
$\ then\ \ (add{-}transition\ \ x\ \ after\ \ y) :: atm)$;

$(transition\ \ (add{-}transition\ \ x\ \ after\ \ y) :: atm\ \ var\ \ (x,\ \ y)\ \ then\ \ f_n)$,

where

- if $x_0 \in [support\ [c_{nf}\ (0 : rules) :: state :: transition]] \cup [support\ t_{rn.rlt.ex.s}]$, and $y_0 \notin [c_{nf}\ (-1 : exogenous, 0 : order) :: state :: transition] :: q\ \ -.\ :: set\ x_0]$, then $(add{-}transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ und\ \#\ c_{nf}$;
- if $x_0 \in [support\ [c_{nf}\ (0 : rules) :: state :: transition]] \cup [support\ t_{rn.rlt.ex.s}]$, and $[value\ [c_{nf}\ (-1 : exogenous, 0 : order) :: state :: transition] :: q\ \ -.\ :: set\ x_0] = n_{m.*.1}\ y_0\ n_{m.*.2}$, then $(add{-}transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [c_{nf}\ transition.(-1 : exogenous, 0 : order) : n_{m.*.1}\ y_0\ x_0\ n_{m.*.2}]$;
- if $x_0 \in [support\ t_{rn.rlt.en.s}]$, and $y_0 \notin [c_{nf}\ (-1 : endogenous, 0 : order) :: state :: transition] :: q\ \ -.\ :: set\ x_0]$, then $(add{-}transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ und\ \#\ c_{nf}$;
- if $x_0 \in [support\ t_{rn.rlt.en.s}]$, and $[value\ [c_{nf}\ (-1 : endogenous, 0 : order) :: state :: transition] :: q\ -.\ :: set\ x_0] = n_{m.*.1}\ y_0\ n_{m.*.2}$, then $(add{-}transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [c_{nf}\ transition.(-1 : endogenous, 0 : order) : n_{m.*.1}\ y_0\ x_0\ n_{m.*.2}]$;
- if $x_0 \notin [support\ [c_{nf}\ (0 : rules) :: state :: transition]] \cup [support\ t_{rn.rlt.ex.s}] \cup [support$

$t_{rn.rlt.en.s}]$, then $(add\!-\!transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ und\ \#\ c_{nf}$.

The element $(delete\!-\!transition\ x)$ deleting the transition with the name x is defined as follows:

$(rule\ \ (delete\!-\!transition\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (delete\!-\!transition\ \ x) :: atm)$;

$(transition\ \ (delete\!-\!transition\ \ x) :: atm\ \ var\ \ (x)\ \ then\ \ f_n)$,

where

- if $x_0 \in [support\ [c_{nf}\ (0 : rules) :: state :: transition]] \cup [support\ t_{rn.rlt.ex.s}]$, then $(delete\!-\!transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [c_{nf}\ transition.(-1 : exogenous, 0 : order) : [value\ [c_{nf}\ (-1 : exogenous, 0 : order) :: state :: transition] :: q\ -. :: set\ x_0 :: q\ c_{nf}]]$;
- if $x_0 \in [support\ t_{rn.rlt.en.s}]$, then $(delete\!-\!transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [c_{nf}\ transition.(-1 : endogenous, 0 : order) : [value\ [c_{nf}\ (-1 : endogenous, 0 : order) :: state :: transition] :: q\ -. :: set\ x_0 :: q\ c_{nf}]]$;
- if $x_0 \notin [support\ [c_{nf}\ (0 : rules) :: state :: transition]] \cup [support\ t_{rn.rlt.ex.s}] \cup [support\ t_{rn.rlt.en.s}]$, then $(delete\!-\!transition\ x_0) :: atm,\ e_{l.*}\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ und\ \#\ c_{nf}$.

The element e_l of the form $(modify\ x)$ or $(modify :: n\ x)$ is defined as follows:

$(rule\ \ (modify\ \ x)\ \ var\ \ (x)\ \ then\ \ (modify\ \ x) :: atm)$;

$(rule\ \ (modify :: n\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (modify\ \ x) :: atm)$;

$(transition\ \ (modify\ \ x) :: atm\ \ var\ \ (x)\ \ then\ \ f_n)$,

where $(modify\ x_0) :: atm, e_{l.*}\ \#\ v_l\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [if\ [$there exists $c_{nf.1}$ such that $[value\ [subst\ (conf :: in : c_{nf}, val :: in : v_l[\![c_{nf}]\!], conf :: out : c_{nf.1}, val :: out : v_l[\![c_{nf.1}]\!])\ x_0]\ c_{nf}] \neq und]\ then\ v_l\ \#\ c_{nf.1}\ else\ und\ \#\ c_{nf}]$. The element x is called a transition condition in $[\![e_l]\!]$. It specifies the set of configurations reachable from c_{nf} for one transition. The elements $conf :: in$ and $conf :: out$ reference to the input state and the output state, and the elements $val :: in$ and $val :: out$ reference to values in these states.

⨁ The execution of the element $(modify\ (((-1 : value,\ 0 : x,\ 1 : variable)\ inconf :: out) = 0))$ initiates the transition to a state in which the value of the variable x equals to 0.

⨁ The execution of the element $(modify\ (((-1 : value,\ 0 : x,\ 1 : variable) = "green")\ and\ (((-1 : value,\ 0 : x,\ 1 : variable)\ in\ conf :: out) = "red")))$ initiates the transition from a state in which the value of the variable x equals to "green" to a state in which the variable x equals to "red".

The element e_l of the form $(modify{-}exist\ (x)\ y)$ or $(modify{-}exist :: n\ (x)\ y)$ is defined as follows:

$(rule\ \ (modify{-}exist\ \ (x)\ \ y)\ \ var\ \ (y)\ \ seq\ \ (x)\ \ then\ \ (modify{-}exist\ \ (x)\ \ y) :: atm)$;

$(rule\ \ (modify{-}exist :: n\ \ (x)\ \ y)\ \ var\ \ (y)\ \ seq\ \ (x)\ \ abn\ \ then\ \ (modify{-}exist\ \ (x)\ \ y) :: atm)$;

$(transition\ \ (modify{-}exist\ \ (x)\ \ y) :: atm\ \ var\ \ (y)\ \ seq\ \ (x)\ \ then\ \ f_n)$,

where $(modify{-}exist\ (x_0)\ y_0) :: atm, e_{l.*}\ \#\ v_l\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [if$ [there exists $c_{nf.1}$ such that $[[subst\ (conf :: in : c_{nf}, val :: in : v_l[\![c_{nf}]\!], conf :: out : c_{nf.1}, val :: out : v_l[\![c_{nf.1}]\!])\ y_0]$ is satisfiable in $((x_0), c_{nf})]$ $then\ v_l\ \#\ c_{nf.1}\ else\ und\ \#\ c_{nf}]$. The element y is called a transition condition in $[\![e_l]\!]$. The elements of x are called existential variables in $[\![e_l]\!]$.

5.14. Safety operations

The element e_l of the form $(assert\ x)$ or $(assert :: n\ x)$ is defined as follows:

$(rule\ \ (assert\ \ x)\ \ var\ \ (x)\ \ then\ \ (assert\ \ x) :: atm)$;

$(rule\ \ (assert\ \ x :: n)\ \ var\ \ (x)\ \ abn\ \ then\ \ (assert\ \ x) :: atm)$;

$(transition\ \ (assert\ \ x) :: atm\ \ var\ \ (x)\ \ then\ \ f_n)$,

where $(assert\ x_0) :: atm, e_{l.*}\ \#\ v_l\ \#\ c_{nf} \rightarrow_{f_n,s_b} e_{l.*}\ \#\ [if\ [[value\ [subst\ (conf :: inc_{nf}, val :: in : v_l)\ x_0]\ c_{nf}]\ \neq\ und]\ then\ v_l\ else\ und]\ \#\ c_{nf}$. The element x is called a safety condition in $[\![e_l]\!]$.

5.15. Branching operations

The element e_l of the form $(branching\ x)$ is defined as follows:

$(rule\ \ (branching\ \ x)\ \ seq\ \ (x)\ \ abn\ \ then\ \ (branching\ \ x) :: atm)$;

$(transition\ \ (branching\ \ x) :: atm\ \ var\ \ (x)\ \ then\ \ f_n)$,

where $(branching\ x_0) :: atm, e_{l.*}\ \#\ v_l\ \#\ c_{nf} \rightarrow_{f_n,s_b} \#\ (type : assume) :: exc\ \#\ [c_{nf}\ branching.\ (0\ :\ ())\ :\ [((x_0),\ c_{nf},\ (e_{l.*}))\ .+\ [[c_{nf}\ branching]\ (0\ :\ ())]]]$. The elements of x are called branches in $[\![e_l]\!]$. The element e_l generates the branchpoint with the branches x. The exception $(type : assume) :: exc$ specifies the failure of the execution of the current branch. The substate $branching$ contains information about branching. The conceptual $(0 : ()) :: state :: branching$ specifies the current sequence of branchpoints.

The endogenous transition relation specifying branching is defined as follows:

$(endogenous{-}transition\ \ f_n) :: name :: branching$

where

- if $[[c_{nf}\ \ branching]\ \ (0\ :\ ())]\ =\ (((e_{l.*.1},\ \ e_l,\ \ e_{l.*.2}),\ \ c_{nf.1},\ \ (e_{l.*.3})), e_{l.*})$, then $\#\ (type\ :$

$assume) :: exc$ # $c_{nf} \rightarrow_{branching} e_{l.*.3}$ # $[c_{nf.1}\ branching.(0 : ()) : (((e_{l.*.1},\ e_{l.*.2}),\ c_{nf.1},\ (e_{l.*.3})), e_{l.*})]$;

- if $[[c_{nf}\ branching]\ (0\ :\ ())] = (((),\ c_{nf.1},\ (e_{l.*.3})), e_{l.*})$, then # $(type : assume) :: exc$ # $c_{nf} \rightarrow_{branching}$ # $(type : assume) :: exc$ # $[c_{nf.1}\ branching.(0 : ()) : (e_{l.*})]$.

The element e_l of the form $(assume\ x)$ or $(assume :: n\ x)$ is defined as follows:

$(rule\ \ (assume\ \ x)\ \ var\ \ (x)\ \ then\ \ (assume\ \ x) :: atm)$;

$(rule\ \ (assume :: n\ \ x)\ \ var\ \ (x)\ \ abn\ \ then\ \ (assume\ \ x) :: atm)$;

$(transition\ \ (assume\ \ x) :: atm\ \ var\ \ (x)\ \ then\ \ f_n)$,

where $(assume\ x_0) :: atm, e_{l.*}$ # v_l # $c_{nf} \rightarrow_{f_n,s_b} [if\ [[value\ [subst\ (conf :: in : c_{nf}, val :: in : v_l[\![c_{nf}]\!])\ x_0]\ c_{nf}] \neq und]\ then\ e_{l.*}$ # $v_l\ else$ # $(type : assume) :: exc]$ # c_{nf}. The element x is called a continuation condition in $[\![e_l]\!]$. The violation of this condition initiates the failure of the execution of the current branch.

The element e_l of the form $(assume{-}exist\ (x)\ y)$ or $(assume{-}exist :: n\ (x)\ y)$ is defined as follows:

$(rule\ \ (assume{-}exist\ \ (x)\ \ y)\ \ var\ \ (y)\ \ seq\ \ (x)\ \ then\ \ (assume{-}exist\ \ x) :: atm)$;

$(rule\ \ (assume{-}exist :: n\ \ (x)\ \ y)\ \ var\ \ (y)\ \ seq\ \ (x)\ \ abn\ \ then\ \ (assume{-}exist\ \ x) :: atm)$;

$(transition\ \ (assume{-}exist\ \ (x)\ \ y) :: atm\ \ var\ \ (y)\ \ seq\ \ (x)\ \ then\ \ f_n)$,

where $(assume\ (x_0)\ y_0) :: atm, e_{l.*}$ # v_l # $c_{nf} \rightarrow_{f_n,s_b} [if\ [[subst\ (conf :: in : c_{nf}, val :: in : v_l[\![c_{nf}]\!])\ y_0]$ is satisfiable in $[\![(x_0), c_{nf}]\!]]\ then\ e_{l.*}$ # $v_l\ else$ # $(type : assume) :: exc]$ # c_{nf}. The element y is called a continuation condition in $[\![e_l]\!]$. The elements of x are called existential variables in $[\![e_l]\!]$.

6. Justification of requirements for conceptual transition systems

In this section, we establish that CTSs meet the additional requirements stated in section 1:

8. *The formalism must have language support. The language associated with the formalism must define syntactic representations of models of states, state objects, queries, query objects, answers and answer objects and includes the set of predefined basic query models.* The CTSL language associated with CTSs defines syntactic representations of models of states, state objects, queries, query objects, answers and answer objects and includes the set of predefined basic query models.
9. *It must model the change of the conceptual structure of states and state objects of the ITS.* The change of the conceptual structure of the ITS is described by the transition relation

on conceptual configurations specifying conceptual structures of the ITS with different sets of ontological elements.

10. *It must model the change of the content of the conceptual structure.* The change of the content of the conceptual structure of the ITS is described by the transition relation on conceptual states specifying the same conceptual structure of the ITS. In fact, the distinction between requirements 9 and 10 is relative, for conceptuals allow to define classifications of ontological elements with different granularity.
11. *It must model the transition relations of the ITS.* The transition relations of the ITS are modelled by the transition relation $t_{rn.rlt}$ of the CTS.
12. *The model of the exogenous transition relation must be extensible.* The model of the exogenous transition relation of the IQS is extended by addition of trnasition rules.

Thus, the additional requirements are met for CTSs.

7. Conclusion

In the paper two formalisms (ITSs and CTSs) for abstract unified modelling of the artifacts of the conceptual design of information systems have been proposed by ontological elements with arbitrary conceptual granularity. The basic definitions of the theory of CTSs have been given. The language of CTSs has been defined.

We plan to use CTSs to design and prototype software systems as well as to specify operational and axiomatic semantics of programming languages. In the case of operational semantics of a programming language, CTSs model an abstract machine of the language. In the case of axiomatic semantics of a programming language, CTSs model a verification conditions generator for programs in the language.

References

1. Sokolowski J., Banks C. Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains. Wiley, 2010.
2. Chen P. Entity-relationship modeling: historical events, future trends, and lessons learned // Software pioneers. Springer-Verlag New York, 2002. P. 296-310.
3. Anureev I.S. Formalisms for conceptual design of closed information systems // System Informatics. 2016. N 7. P. 69-148.
4. Gurevich Y. Abstract state machines: An Overview of the Project // Foundations of Information and Knowledge Systems. Lect. Notes Comput. Sci. 2004. Vol. 2942. P. 6-13.

5. Gurevich Y. Evolving Algebras. Lipari Guide // Specification and Validation Methods. Oxford University Press, 1995. P. 9-36.