Transition systems specified as a communication tool for e-learning

BOGDAN A. BRUMAR, EMIL M. POPA
Department of Computer Science
Faculty of Sciences
Lucian Blaga University of Sibiu
Str. Ion Ratiu, no. 5-7, 550012
Sibiu, ROMANIA

IULIAN PAH, DAN CHIRIBUCĂ
Department of Sociology,
Babes-Bolyai University
Cluj-Napoca, Bd.21 decembrie
1989, no.128-130, 400604,
Cluj-Napoca, ROMANIA

Abstract: - Due to the dynamic nature of the computing objects represented, the semantics of the specification languages used to formalize systems will be a transition system performing computation actions, while the syntax will be the linguistic expression of the actions performed by the transition system. Computing objects manipulated in computer science bear a lot of similarities with the mathematical objects produced by the systems discussed so far they also are very different. The concept of language is regarded as a communication tool that allows language users to develop knowledge, while interacting with their universe of discourse, and to communicate with each other, while exchanging knowledge, in that case for e-learning systems. The results presented here have as initial point [1], [7].

Key-Words: - system implementation language, system specification, system validation language, transition system, macro-operations.

1 Introduction

Systems represent computations with specific behavioral properties the major steps involved in this formalization are:

- **System specification**: defines the system as a computing object of the form System = <Specification, Behavior>.
- **System implementation**: expresses the behavior of the system such that it can be observed.
- **System validation**: shows that the behavior of the system has the requested properties.

Each of these steps requires a specific language for its representation. A language in this context is a tuple L = <Sem, Syn, L:Sem→Syn> where:

- Sem is the language semantics;
- Syn is the language syntax;
- L:Sem→Syn is a partial mapping that expresses computing objects c ∈ Sem by means of their linguistic expressions L(c) ∈ Syn in a way that there exists a total mapping ε : Syn→Sem such that ε (L(c)) = c whenever L is defined.

Due to the dynamic nature of the computing objects represented, the semantics of the specification languages used to formalize systems will be a transition system performing computation actions, while the syntax will be the linguistic expression of the actions performed by the transition system.

2 Language and systems as communication tool

A language in this paper is a mechanism of communication that allows its users to interact with their universe of discourse and with each other. Let’s define two terms that we need:

**Definition 1**: Cognition is the interaction with the universe of discourse.

**Definition 2**: Communication represents the interaction with each other.

Language elements are tuples of the form <symbol, entity> called knowledge.

- **Entity**: component of knowledge is an abstract or concrete object of the environment called knowledge meaning.
- **Symbol**: component of knowledge is a sensory (acoustic, visual, etc) representation of the object, called knowledge token.

Figure 1 provides an example of knowledge [7]:

![Figure 1. Knowledge representation](image-url)
The cognition process can be modeled in terms of the operations perception, recognition, and action [2], defined as follows:

- **perception**: associates the environment’s objects with innate or learned representations called matching concepts (M-concepts).
- **recognition**: instantiates perception patterns (i.e., M-concepts) as knowledge tokens, called internal concepts (I-concepts), by substituting appropriate values for M-concept parameters.
- **action**: maps I-concepts into the environment’s objects associated with their M-concepts by the identification of the appropriate parameter values. M-concepts, I-concepts, values, and parameter processing (evaluation, testing and substitution) in the cognition process are modeled here by computational objects called macro-operations and macro-processors.

2.1 Macro-operations

**Definition 3**: A macro-operation is a tuple \( M = \langle \text{name, body} \rangle \) where \( \text{body} = \) a class of objects and \( \text{name} = \) references to the objects of the \( \text{body} \). The process of mapping a macro-name or a macro-body into the object it specifies is called macro-expansion and is performed by a macro-processor, \( \mathcal{H} \). A macro-processor takes macro-operation names and parameter values as arguments and expands the associated macro-bodies by checking constraints and substituting parameter values for parameters. Macro-names are mapped into knowledge tokens representing M-concepts and macro-bodies are mapped into knowledge meanings representing I-concepts. Thus, the macro-processor that expands \( M.\text{name} (\langle@0 \ldots @n\rangle) \) into the object it specifies interprets the parameters @i, \( 0 \leq i \leq n \), different from the interpretation of the parameters @i, \( 0 \leq i \leq n \), by the macro-processor that expands \( M.\text{body}(\langle@0 \ldots @n\rangle) \), into the object it specifies [7].

2.2 Communicators and language systems

To understand better the cognition process we try to define formally its operations:

**Perception**: is defined by an open-ended list of specification rules of the form \( M = \langle \text{parameter-pattern}: \text{macro-name, macro-body} \rangle \), where parameter pattern is a relation of the form \( \text{lhs} \rightarrow \text{rhs} \) (left-hand-side \( \rightarrow \) right-hand-side) [1] which specifies the parameters, and macro-name and macro-body are macro-operations defining an M-concept and its associated I-concept, respectively, using the parameters in the parameter-pattern, as seen in Table 1 [7].

Table 1: Example of a cognition environment specification

<table>
<thead>
<tr>
<th>Command</th>
<th>VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>name:</td>
<td>@0.\text{surface} = @1.\text{surface};</td>
</tr>
<tr>
<td>body:</td>
<td>@0.\text{action} = @1.\text{action}</td>
</tr>
<tr>
<td>2. VP</td>
<td>( \vee ) N;</td>
</tr>
<tr>
<td>name:</td>
<td>@0.\text{surface} = @1.\text{surface} \ast @2.\text{surface};</td>
</tr>
<tr>
<td>body:</td>
<td>@1.\text{type} = @2.\text{type} \rightarrow @0.\text{type};</td>
</tr>
<tr>
<td></td>
<td>( @0.\text{action} = \text{call}(@1.\text{action}(@2.\text{args})) )</td>
</tr>
<tr>
<td>3. N</td>
<td>( \rightarrow ) “triangle”;</td>
</tr>
<tr>
<td>name:</td>
<td>@0.\text{surface} = \text{triangle};</td>
</tr>
<tr>
<td>body:</td>
<td>@0.\text{args} = \text{int size1, size2, size3};</td>
</tr>
<tr>
<td></td>
<td>( \Delta (\text{size1, size2, size3}) )</td>
</tr>
<tr>
<td>4. V</td>
<td>( \rightarrow ) “search”;</td>
</tr>
<tr>
<td>name:</td>
<td>@0.\text{surface} = \text{search};</td>
</tr>
<tr>
<td>body:</td>
<td>@0.\text{type} = \text{int \times int \times int} \rightarrow \Delta;</td>
</tr>
<tr>
<td></td>
<td>( @0.\text{action} = \text{SearchTriangle}(\text{int, int, int}) )</td>
</tr>
<tr>
<td>5. V</td>
<td>( \rightarrow ) “make”;</td>
</tr>
<tr>
<td>name:</td>
<td>@0.\text{surface} = \text{make};</td>
</tr>
<tr>
<td>body:</td>
<td>@0.\text{type} : \text{int \times int \times int} \rightarrow \Delta;</td>
</tr>
<tr>
<td></td>
<td>( @0.\text{action} = \text{MakeTriangle}(\text{int, int, int}) )</td>
</tr>
</tbody>
</table>

This list contains innate elements for natural systems and may be expanded (by learning) or shrunk (by forgetting).

**Recognition**: takes as input a macro-name (knowledge token), and some parameters, and expands the macro-body associated with the macro-name by processing the appropriate parameters to obtain an internal representation of the knowledge meaning (I-concept).

For example, make triangle \((2,3,4) \rightarrow \Delta (2, 3, 4)\).

**Action**: takes as input a macro-body (knowledge meaning), and some parameters, and uses the associated macro-name to deliver the object from the environment, i.e., the knowledge token, represented by the I-concept.

For example, search \(\Delta (2, 3, 4) \rightarrow \text{triangle}\).

By macro-processing, a macro-processor \( M_n \) expands macro-names into external representations of I-concepts (i.e., into knowledge tokens), while a macro-processor \( M_s \) expands the macro-bodies into internal representation of I-concepts (i.e., into knowledge meanings). Macro-names serve as abstract representations of classes of similar objects of the environment; the associated macro-bodies identify individual objects of the class and the operations defined on these objects. Using this representation of M-concepts and I-concepts, a knowledge can be defined as a tuple \( \langle \text{symbol, entity} \rangle \), where symbol is a macro-name representing an M-concept and entity is a macro-body where some parameters are replaced by parameter values.

A cognition system, \( CS \), can be modeled by a tuple \( CS = \langle \text{Environment,Record,Action,Recognition,Control} \rangle \) where Environment is a universe of discourse defined by a given list of knowledge specification rules (as seen in Table 1), Record is a mechanism of knowledge representation, for example \( \langle \text{triangle, \Delta} \rangle \), Action maps objects in the environment into knowledge in the
Record, Recognition maps knowledge in the Record into the objects they represent, and Control is a logical expression in terms of Action and Recognition that controls the system behavior.

For example, repeat forever (Recognition → Action) V (Action → Recognition), is such a control expression. While the Environment and the Record of two different cognition systems are in general different, the operations Action, Recognition, and Control are the same. Therefore, we use the notation Env, Record, as references to the environment and the knowledge representation of the cognition system c.

**Definition 4:** A language user (communicator) is a natural or artificial system consisting of

1. a cognition system 
   
c = < Env, Record, Action, Recognition, Control >

2. a collection of knowledge represented by a database, Dbase, defined by
   
   \[ Dbase_c = \{ w | \exists o \in \text{Enc}_c, \Lambda(w,o) \in \text{Record}_c \} \]

3. a mechanism that consists of a learning action, denoted Learn, that maps objects in Env into knowledge representations in Dbase, and an evaluation action, denoted Eval, that maps knowledge representations in Dbase into the objects of Env, as shown in Figure 2 [7].

![Fig. 2. The structure of a communicator](image-url)

We observe that Env is used twice in Figure 2 to emphasize its double relation with the Dbase, as the source of knowledge represented in the Dbase, and as the target of the actions represented by the knowledge in the Dbase. Hence, communicators can be modeled by tuples of the form <Env, Dbase, Learn, Eval> where Learn and Eval are binary relations, Learn ⊆ Env × Dbase, Eval ⊆ Dbase × Env. As usual, ∀m ∈ Env, Learn(m) = {w ∈ Dbase | (m, w) ∈ Learn} and ∀w ∈ Dbase, Eval(w) = {m ∈ Env | (w, m) ∈ Eval}.

A language L is modeled here by a tuple

\[ L = \langle U, P, \mathcal{L}, C \rangle \]

where U is a universe of discourse (i.e., a cognition environment), and P is a collection of word-expressions, called parlance, \( \mathcal{L} \subseteq U \times P \) is a learning relation that associates objects \( u \in U \) with word-expressions \( p \in P \) thus generating knowledge \( (u, p) \), and \( C \subseteq P \times U \) is an evaluation relation that interprets word-expressions \( p \in P \) and objects \( u \in U \) to determine if \((u, p)\) is a language knowledge.

**Definition 5:** A linguistic system, \( LS \), is a tuple \( LS = (L, C) \) where \( L \) is a language and \( C \) is a collection of communicators using \( L \) to interact with their environments and to communicate with each other. For \( LS = (L, C) \), \( L = \langle U, P, \mathcal{L}, \subseteq U \times P, C \subseteq P \times U \rangle \) and \( c \in C \), \( c = \langle \text{Env}_c, \text{Dbase}_c, \text{Learn}_c, \text{Eval}_c \rangle \) is provided with two mappings, \( \text{Generate}_c : \text{Dbase}_c \rightarrow P \) and \( \text{Interpret}_c : P \rightarrow \text{Dbase}_c \), such that

\[ \text{Learn}_c \circ \text{Generate}_c \subseteq \mathcal{L}, \text{Interpret}_c \circ \text{Eval}_c \subseteq C \]

where \( \text{“} \circ \text{”} \) is the relation composition. In order to interact and communicate using \( L \), the assumption is that \( c \) has learned \( L \), i.e., \( \text{Env}_c \subseteq U \) and \( \forall m \in \text{Env}_c \wedge \forall w \in \text{Learn}_c(m). \text{(m, \text{Generate}_c(w))} \in \mathcal{L} \) and \( \forall w \in P \wedge \forall m' \in \text{Eval}_c(\text{Interpret}_c(w)). (w, m') \in C \).

These relations are satisfied if \( \text{Dbase}_c \subseteq P \).

### 2.3. Language use

Language use by communicators consists of generating knowledge and exchanging them with other communicators. A communicator \( c \) generates knowledge \( (m, w_m) \in \text{Env}_c \times \text{Dbase}_c \), by its own cognition process (described by macro-expansion). When \( c \) uses a linguistic system to communicate, \( c \) embeds \( m \) in the language universe of discourse and maps \( w_m \) into a language well-formed word-expression \( w_m \). Hence, for each \( m \in U \) the word-expression \( w_m \in P \) such that \( (m, w_m) \in \mathcal{L} \wedge (w_m, m) \in C \) is the literal meaning of \( (m, w_m) \). The linguistic phenomenon by which a word-expression, \( w_m \), means the object, \( m \), of the universe of discourse represented by \( w_m \) in knowledge is called a reference relation of \( m \).

For a language \( L = \langle U, P, \mathcal{L}, \subseteq U \times P, C \subseteq P \times U \rangle \), the reference modeling can also be described by a macro-expansion process. Literal meanings \( w_m \) are language types (M-concepts) defined by the parameterized macro-operations of the cognition environment defining \( U \). The contextual references \( w_m \) are language tokens (I-concepts) obtained by the expansion of macro-operations defining the cognition environment of the communicator \( c \), i.e., \( w_m \in \text{Dbase}_c \).

Hence, the reference in a linguistic system generates two notions of meaning [rus]:

1. a literal meaning, also called speaker’s meaning, is a word-expression \( w_m \) such that there is a knowledge meaning \( m \) expressed by \( w_m \), i.e., \( (m, w_m) \in \mathcal{L} \) implies \( (w_m, m) \in C \), and (ii) a utterance meaning, also called auditor’s meaning, is a word-expression \( w_m \) such that there is a knowledge meaning \( m' \) to which \( w_m \) is evaluated, i.e., \( (w_m, m') \in C \) implies \( (m', w_m) \in \mathcal{L} \).

The relationship between \( w_m \) (which exists independent of its contextual use) and \( w_m \) (whose existence depends upon the existence of a language user) is described by the communication system in Figure 3 [7], organized on
three levels: language level, where a language $L$ is introduced as a universal communicator, speaker level, where a speaker $S$ is shown as a particular communicator using $L$ to generate knowledge, and auditor level, where an auditor $A$ is seen as a particular communicator using $L$ to interpret knowledge.

The specification language borrows form computer science the idea of using keywords in order to relate it to the natural language of its users. This language has already penetrated the field of computer science under the name of abstract data types.

An action specification is a program in the language used to express reactive systems and is provided in the specification by the keyword Actn. An action consists of two parts, the name, and the linguistic expression of the action. The name of the action is separated by double colon, ::, from its linguistic expression. The linguistic expression of the action is composed of a declaration part and an action part. Formally, an action specification is a linguistic expression of the form $A ::= [D][A_1 || A_2 || ... || A_n]$ where the following notation is used:

- $A$ is the name of the action performed by the system.
- $D$ is a sequence of typed lists of variables of the form

\[ \text{mode} \text{ List } : \text{type where } \phi \text{ where mode is one of in, out, inout, local, type is a type of value accepted in the system, and } \phi \text{ is an assertion satisfied by the variables in the List.} \]

- $A_1, A_2, ..., A_n$ are actions in terms of which the action $A$ is specified. Each $A_i$ is either a call to a previously defined action, or has the form $[D]; S$, where $D$ is a declaration and $S$ is a statement describing the action to be performed on the variables in $D \cup D_i$. When $A_i$ is an action call, its expression in $A$ is $A_i(arg)$ where $arg$ is the list of variables used by $A_i$ for its task. Arguments can be: in, out, inout. The in arguments are imported and not modifiable, the out arguments are exported and inout arguments are imported and modifiable.

**Definition 6.** A process is a tuple $P = (Agent, Action, Status)$ where Agent is a processor capable to perform statements composing the actions in Action and Status is the state of this performance.

In order to perform the statements of an action, the processor has a control mechanism that shows the label of the statement currently executed. Denote this control by $\pi$. Statements are simple or composed. Each statement has the form $l : body : \hat{i}$, where $l$ is a label that identifies the statement by showing the entry point in the statement body and $\hat{i}$ is a label showing the exit point from the statement body. The simple statements are performed by the processor atomically. There are three types of simple statements in a system specification.

They are called skip, await and assignment and are defined as follows:

- **Skip** statement has the form $l : skip : \hat{i}$ and its performance means “skip”.
- **Await** statement has the form $l : await e : \hat{i}$ where $e$ is a boolean expression and its performance means “wait until $e$ becomes true”.
- **Assignment** statement has the form $l : (x_1, x_2, ..., x_n) := (e_1, e_2, ..., e_n) : \hat{i}$ denoted by $l : \overline{x} := \overline{e} : \hat{i}$, where $\overline{x} = (x_1, x_2, ..., x_n)$, $\overline{e} = (e_1, e_2, ..., e_n)$, and for $l = 1, 2, ..., n$, $x_i$ and $e_i$ have the same type. This is also called a multiple assignment.

The composed statement of the specification language is concatenation, branch, loop, choice, parallel and block. The statement composition generates redundant labels.
To simplify this we group together all redundant labels in equivalence classes. A class of equivalence contains all labels that denote the entry point or the exist point of a statement. Each equivalence class is represented by one label. That is, we assume that each statement $S$ has just one entry point and one exit point. The set of labels denoting the entry point of $S$ is $\text{Entry}(S)$ and the set of labels denoting the exit point of $S$ is $\text{Exit}(S)$.

The labeling of statements is however optional. The entry and exit points of a statement that has no labels coincides with the textual begin and end of that statement and its labels are considered to be the empty string. This language of actions can be freely extended in order to express various computation performed by different systems.

4 System specification language

Formally, a system can be defined as a pair $\text{System} = \langle TS, A \rangle$ where $TS$ is a transition system and $A$ is an action expression specifying the computations performed by the $TS$. Using the systematic approach for a system construction we develop a system by successive iterations. At each iteration we construct a version of $TS$ and then express it by an appropriate action $A$ to obtain a process $P = \langle \text{Agent}, A, \text{Status} \rangle$. This process when active performs the computations specified by $TS$. The formalization of this specification approach leads to a System Specification Language, $SSL$, $SSL = \langle SSL_{\text{Sem}}, SSL_{\text{Syn}}, L : SSL_{\text{Sem}} \rightarrow SSL_{\text{Syn}} \rangle$ where:

- The semantics $SSL_{\text{Sem}}$ of the $SSL$ are transition systems.
- The syntax $SSL_{\text{Syn}}$ of the $SSL$ are actions.
- The function $L : SSL_{\text{Sem}} \rightarrow SSL_{\text{Syn}}$ is determined by the process that allows us to express transition systems by actions. The language evaluation function $\varepsilon : SSL_{\text{Sem}} \rightarrow SSL_{\text{Syn}}$ is defined as follows: if $A :: [D][A_1 || A_2 || \ldots || A_n]$ is an action in $SSL_{\text{Syn}}$ then $\varepsilon(A)$ is the transition system $TS_A = \langle \Pi_A, \Sigma_A, T_A, \Theta_A \rangle$ in $SSL_{\text{Sem}}$ constructed as follows:
  - $\Pi_A$ is the set of all variables declared in $A$ together with a control variable $\pi$ that runs over the power set of the collection of labels $L_A$ in $A$.
  - Each $s \in \Sigma_A$ is an assignment $s : \Pi_A = D_A$ where $D_A = \bigcup_{x \in \Pi_A} \text{Type}(x) \cup L_A$.
  - $T_A$ is the set of transitions determined by the statements of $A$. The idling transition is $\rho_A : T$.
  - The initial condition of $\Theta_A$ is $\Theta_A = (\pi = \text{Entry}(A) \cup \ldots \cup \text{Entry}(A_n)) \land \phi$ where $\phi$ is the conjunction of all where assertions in the expression of $A$.

A computation performed by the transition system $TS_A$ is expressed by the sequence of transition $\langle \pi, x_1, \ldots, x_n \rangle \xrightarrow{\rho_1} \langle \pi', x_1', \ldots, x_n' \rangle \xrightarrow{\rho_2} \ldots$ where $\{ \pi, x_1, \ldots, x_n \} = \Theta$ and $S_1, S_2, \ldots$ are statements of the action performed by $TS_A$.

An important question in our study is to determine when two computing objects are equivalent. We call two computing objects equivalent if the transition systems performing their behavior are equivalent. Two transition systems $TS_1$ and $TS_2$ are equivalent when they generate the same set of computations. Since every computation of a transition system in an infinite sequence of state transitions where the entire state is seen, this concept of equivalence is too discriminating. So, the equivalence of transition systems should be defined up to a set of state variables that are observable. The set of observable state variables should be specified by the user. That is, the computations generated by two transitions systems are considered to be the same if the values of the observable state variables are the same. Consequently, one can define the reduced behavior of a transition system to be the set of its computations where only the values taken by the observable variables are seen. If a reduced state is the observable part of that state then the reduced behavior $\sigma'$ of a computation $\sigma$ is determined as follows:

- $t_1$ Replace each state $s_i$ of $\sigma$ by its observable part contained by restricting $s_i$ to the observable variables.
- $t_2$ Omit from the sequence of states of $\sigma$ each state that coincides with its predecessor but differs from its successor.

The reduced behavior of a transition system $TS$ with respect to a given set of observable variables $O$ is denoted by $\mathcal{R}(TS, O)$.

Since actions are specified in terms of other actions this concept of equivalence should detect the situations where two actions can be used interchangeably. Let us assume that the variable $S$ runs over actions. Denote an action that depends on $S$ by $A(S)$ and by $A(A_1)$ the action that is obtained from $A(S)$ by replacing all occurrences of $S$ in $A_1$ by $A$. Then two actions $A_1$ and $A_2$ are called congruent, denoted by $A_1 \approx A_2$, if $TS_{\mathcal{R}(A_1)} \sim TS_{\mathcal{R}(A_2)}$ for every action $A(S)$. Example of congruent actions are provided by the associatively of the operators “\|$” (concatenation), “or” (choice), “||” (parallel) used to construct composed statements.

There are two relations among actions $A_1, A_2$ that allow the replacement of $A_1$ by $A_2$, emulation and implementation. Such a replacement is desirable when $A_2$ is expressed in terms of language constructs that can be run on a given computer. The action $A_1$ emulates the
action $A_2$ if $\mathcal{R}(TS_{A_2}) = \mathcal{R}(TS_{A_1})$. The action $A_2$ implements the action $A_1$ if $\mathcal{R}(TS_{A_2}) \subseteq \mathcal{R}(TS_{A_1})$.

5 System implementation language

To achieve its goal, the objects and the operations used to specify a system should behave as the data and operations of an abstract machine which performs the computation task specified by the system. In other words, the system expression written in the system specification language should be transformed into a computation object of an abstract machine. The abstract machine used to express computing objects is a programming language implemented on an actual computer.

The programming language that allows us to express systems as computation objects is called the system implementation language. The process of mapping the system specification language into the system implementation language is called the system implementation. We use the C programming language as the system implementation language. C is regarded here mere as a tool for the software system designer.

6 System validation language

The validation of a system is the process of showing that the system performs the function for which it was designed. The validation process consists of actually using the system in appropriate applications according to the function that it performs.

The language used to express applications which use a system in order to validate the system is called the system validation language. Usually a system is validated using the system implementation language. Therefore, we use C as the software system validation language.

7 Conclusions

Personal contributions are linked to system analysis at specification, implementation and validation level, by representing semantic objects using symbolic notations, and the process of using knowledge by mapping symbolic notations into the semantic objects they represent. This view of a language splits the language users into two classes: speakers, who are communicators producing language expressions, and auditors, who are communicators interpreting language expressions.

References:

[7] T. Rus, A Unified Language Processing Methodology, University of Iowa, Department of Computer Science, 2001