Model Specifications of Transition Systems

BOGDAN A. BRUMAR, EMIL M. POPA,
IOANA C. BRUMAR, FLORENTINA L. CACOVEAN
Department of Computer Science
Lucian Blaga University of Sibiu, Faculty of Sciences
Sibiu, str. Dr. Ion Ratiu, no. 5-7, zip code 550012, Sibiu
ROMANIA

Abstract: - 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. 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.

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

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 $c \in$ Sem by means of their linguistic expressions $L(c) \in$ Syn in a way that there exists a total mapping ε : Syn→Sem such that $\varepsilon(L(c)) = c$ 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 Transition Systems

A transition system is a tuple Ts = $<\Pi, \Sigma, T, \Theta>$ where:

- Π is a set of typed symbols called the state variables. There are two kinds of variables in Π: data variables used to denote data values and control variables used to denote control values.
- Σ is a set of states. Each $s \in \Sigma$ is an assignment of the variables in Π, i.e., $s : \Pi \rightarrow D$, where D is the domain of values of the variables in Π. If $u \in \Pi$ then s[u] denotes the value assigned to u by s. The relation $s : \Pi \rightarrow D$ can be uniquely extended to the valuation homomorphism $Vs : W(\Xi, \Pi) \rightarrow A(\Xi, D)$ where Ξ is an operator scheme, $W(\Xi, \Pi)$ is the term generated by Ξ in symbols of Π, and $A(\Xi, D)$ is the values generated by Ξ on the domain D. If $e \in W(\Xi, \Pi)$ and $s:\Pi \rightarrow D$ is an assignment then s[e] denotes Vs(e). If ϕ is an assertion and $Vs(\phi)$ = true then we write $s \models \phi$ and call s a ϕ-state.
- T is a finite set of transitions. Each $\tau \in T$ is a mapping $\tau : \Sigma \rightarrow P(\Sigma)$ where $P(\Sigma)$ is the power set of Σ. That is, for each $s \in \Sigma$, $\tau(s) \subseteq \Sigma$ ($\tau(s)$ can be the $\varnothing$). If $\tau(s) \neq \varnothing$ and $s` \in \tau(s)$ then s` is called a τ-successor of s. In addition, we assume that there is $\tau I \in T$ such that for each $s \in \Sigma$, $\tau_I(s) = \{s\}$. We call τ_I the idling transition. The transitions in the set $TD = T \setminus \{\tau_I\}$ are called diligent.

- Θ is an assertion called the initial condition of T_s. A state $s \in \Sigma$ such that $s \models \Theta$ is called an initial state of T_s.

Each transition $\tau \in T$, is characterized by an assertion relating the values of the variables in states $s : \Pi \to \mathbf{D}$ and $s` : \Pi \to \mathbf{D}$ for each $s` \in \tau(s)$. Denote this relation by ρ_τ. Formally, if $s : \Pi \to \mathbf{D}$ and $s` \in \tau(s)$, $s` : \Pi \to \mathbf{D}$, then $\forall u \in \Pi\ (s[u], s`[u]) \in \rho_\tau$. Using infix notation this is expressed by $s[u]\ \rho_\tau\ s`[u]$. The assertion ρ_τ allows us to refer to the values of the variables u of Π as u, the value u before the transition τ, and $u`$, the value after the transition $\tau$. $u`$ is called the primed version of u. That is, we may think of two copies Π and $\Pi`$ of the variable of $T_s$ and denote the transition relation of $\tau$ by $\rho_\tau(\Pi, \Pi`)$. The transition relation $\rho_\tau(\Pi, \Pi`)$ has the form $\rho_\tau = C_\tau(\Pi) \wedge (y_1' = e_1) \wedge \ldots \wedge (y_k' = e_k)$ where:

- $C_\tau(\Pi)$ is an assertion called the *enabling condition* that depends only on the values of the variables in the state s before the transition. $C_\tau(\Pi)$ states the condition under which s may have a τ-successor, i.e., $\tau(s) \neq \emptyset$.
- $(y_1' = e_1) \wedge \ldots \wedge (y_k' = e_k)$ is the conjunction of the modifications performed by τ when it takes place. Each $y_i' = e_i$ requires that the primed value of the variable y_i to be computed using the non-primed value of e_i, i.e., $\forall s` \in \tau(s)(s`[y_i] = s[e_i])$, where $y_1, y_2, \ldots, y_k$ are pair wise distinct.

Notation:

The transition relation

$$\rho_\tau(\Pi, \Pi`) = C_\tau(\Pi) \wedge (y_1' = e_1) \wedge \ldots \wedge (y_k' = e_k)$$

can be denoted by

$$\rho_\tau: C_r \wedge (\bar{y}' = \bar{e})$$

where

$$\bar{y}' = (y_1', y_2', \ldots, y_k') \text{ and } \bar{e} = (e_1, e_2, \ldots, e_k).$$

If $r \in T$ and $s \in \Sigma$ then if $\tau(s) \neq \emptyset$ we say that τ is enabled on s and if $\tau(s) = \emptyset$ we say that τ is disabled on s. Notice that if $\rho_\tau: C_\tau \wedge (\bar{y}' = \bar{e})$ then τ is enabled on s iff $s \models C_r$. For a set of transitions, $T \subseteq T$ and $s \in \Sigma$ we say that T is enabled on s if there is $\tau \in T$ and τ is enabled on s; T is disabled on s if for each $\tau \in T$, τ is disabled on s. A state $s \in \Sigma$ is called *terminal* if the only enabled transition on s is the idling transition τ_I. Clearly all successors of a terminal state are terminals.

We now use a transition system to define the computational behavior of programs:

Definition 1. Let $T_s = <\Pi, \Sigma, T, \Theta>$ be a transition system. A computation σ of T_s is an infinite sequence of states $\sigma : s_0, s_1, \ldots$ that satisfies the following requirements:

- **Initiation:** s_0 is initial, that is, $s_0 \models \Theta$.
- **Consecution:** $\forall i (s_{i+1} \in \tau(s_i))$ for some $\tau \in T$. The pair (s_i, s_{i+1}) is called a τ-computation step, or simply τ-step.
- **Diligence:** either σ contains infinitely many diligent τ-steps for $\tau \in T$ or it contains a terminal state. Since τ-steps of a terminal state leave that state terminal, a computation that contains a terminal state is called terminal.

3 System Specification

The universal language emerges into a specification language that is widely accepted in computer science. 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 $A :: [D][A_1 \parallel A_2 \parallel \ldots \parallel A_n]$ 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 **mode** *List* : **type where** ϕ where **mode** is one of *in, out, inout, local,* **type** is a type of value accepted in the system, and ϕ is an assertion satisfied by the variables in the *List.*
- $A_1, A_2, \ldots, A_n$ 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 $[D_i]$; S_i where D_i is a declaration and S_i 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 2. A process is a tuple

$$P = \langle Agent, Action, Status\rangle$$

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 π. Statements are simple or composed. Each statement has the form $l : body : \hat{l}$, where l is a label that identifies the statement by showing the entry point in the statement body and $\hat{l}$ 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:

- The *skip* statement has the form l : **skip** : $\hat{l}$ and its performance means “skip”.
- The *await* statement has the form l:**await** e:$\hat{l}$ where e is a boolean expression and its performance means “wait until e becomes true”.
- The assignment statement has the form
$$l : (x_1, x_2, \ldots, x_n) := (e_1, e_2, \ldots, e_n) : \hat{l}$$
denoted by
$$l{:}\,\bar{x} := \bar{e} : \hat{l}\,,$$
where
$$\bar{x} = (x_1, x_2, \ldots, x_n),\ \bar{e} = (e_1, e_2, \ldots, e_n),$$
and for I = 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 *Entry*(S) and the set of labels denoting the exit point of S is *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 ε. 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 *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** = $\langle Agent, A, Status\rangle$. This process when active performs the computations specified by *TS*. The formalization of this specification approach leads to a System Specification Language, *SSL*, $SSL = \langle SSL_{Sem}, SSL_{Syn}, \mathrm{L} : SSL_{Sem} \rightarrow SSL_{Syn}\rangle$

where:

The semantics SSL_{Sem} of the *SSL* are transition systems.

The syntax SSL_{Syn} of the *SSL* are actions.

The function L : $SSL_{Sem} \rightarrow SSL_{Syn}$ is determined by the process that allows us to express transition systems by actions. The language evaluation function $\varepsilon : SSL_{Sem} \rightarrow SSL_{Syn}$ is defined as follows: if $A :: [D][A_1 \parallel A_2 \parallel \ldots \parallel A_n]$ is an action in SSL_{Syn} then $\varepsilon\ (A)$ is the transition system $TS_A = \langle \Pi_A, \Sigma_A, T_A, \Theta_A\rangle$ in SSL_{Sem} constructed as follows:

- Π_A is the set of all variables declared in A together with a control variable π that runs over the power set of the collection of labels L_A in A.
- Each $s \in \Sigma_A$ ia an assignment $s : \Pi_A = \mathrm{D}_A$ where $\mathrm{D}_A = \bigcup_{x \in \Pi_A} Type(x) \cup L_A$.
- T_A is the set of transitions determined by the statements of A. The idling transition is $\rho_I : T$.
- The initial condition of Θ_A is $\Theta_A = (\pi = Entry(A_1) \cup \ldots \cup Entry(A_n)) \wedge \phi$ where ϕ 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 $\langle \pi,\ x_1, \ldots,\ x_n\rangle \xrightarrow{\rho_{S_1}} \langle \pi`,\ x`_1, \ldots,\ x`_n\rangle \xrightarrow{\rho_{S_2}} \ldots$

where $\{\pi, x_1,\dots, x_n\}| = \Theta$ and S_1, S_2,… 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 σ^r of a computation σ is determined as follows:

t_1 Replace each state s_i of σ by its observable part contained by restricting s_i to the observable variables.

t_2 Omit from the sequence of states of σ 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 $\Re(TS, O)$.

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 by A_1. Then two actions A_1 and A_2 are called congruent, denoted by $A_1 \approx A_2$, if $TS_{A(A_1)} \sim TS_{A(A_2)}$ 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 $\Re(TS_{A_1}) = \Re(TS_{A_2})$. The action A_2 implements the action A_1 if $\Re(TS_{A_2}) \subseteq \Re(TS_{A_1})$.

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.

References:

[1] S. Greibach, *Full AFL's and nested iterated substitution*, IEEE Conf. Record 10, Ann.Symp. Switching Automata Theory, 1969, pp. 222-230.

[2] J. Kral, *A modification of a substitution theorem and some necessary and sufficient conditions for sets to be context free*, Math. Systems Theory, 1970, no. 4, pp. 129-139.

[3] I. McWhirter, *Substitution expressions*, J. Comput. System Sci., 1971, no. 5, pp. 629-637.

[4] T. Rus, *A language Independent Scanner generator*, available at http://cs.uiowa.edu/rus/, 1999.

[5] M. Yntema, *Cap expressions for context free languages*, Information and Control, 1971, no. 18, pp. 311-318..