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:
  \[ \text{System} = \langle \text{Specification}, \text{Behavior} \rangle. \]

- **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 \( L = \langle \text{Sem}, \text{Syn}, L: \text{Sem} \to \text{Syn} \rangle \) where:

- \( \text{Sem} \) is the language semantics;
- \( \text{Syn} \) is the language syntax;
- \( L: \text{Sem} \to \text{Syn} \) is a partial mapping that expresses computing objects \( c \in \text{Sem} \) by means of their linguistic expressions \( L(c) \in \text{Syn} \) in a way that there exists a total mapping \( \varepsilon : \text{Syn} \to \text{Sem} \) 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 \( Ts = \langle \Pi, \Sigma, T, \Theta \rangle \) where:

- \( \Pi \) is a set of typed symbols called the state variables. There are two kinds of variables in \( \Pi \): data variables used to denote data values and control variables used to denote control values.
- \( \Sigma \) is a set of states. Each \( s \in \Sigma \) is an assignment of the variables in \( \Pi \), i.e., \( s : \Pi \to D \), where \( D \) is the domain of values of the variables in \( \Pi \). If \( u \in \Pi \) then \( s[u] \) denotes the value assigned to \( u \) by \( s \). The relation \( s : \Pi \to D \) can be uniquely extended to the valuation homomorphism \( Vs : W(\Xi, \Pi) \to A(\Xi, D) \) where \( \Xi \) is an operator scheme, \( W(\Xi, \Pi) \) is the term generated by \( \Xi \) in symbols of \( \Pi \), and \( A(\Xi, D) \) is the values generated by \( \Xi \) on the domain \( D \). If \( e \in W(\Xi, \Pi) \) and \( s : \Pi \to D \) is an assignment then \( s[e] \) denotes \( Vs(e) \). If \( \phi \) is an assertion and \( Vs(\phi) = \text{true} \) then we write \( s \models \phi \) and call \( s \) a \( \phi \)-state.
- \( T \) is a finite set of transitions. Each \( \tau \in T \) is a mapping \( \tau : \Sigma \to P(\Sigma) \) where \( P(\Sigma) \) is the power set of \( \Sigma \). That is, for each \( s \in \Sigma \), \( \tau(s) \subseteq \Sigma \) (\( \tau(s) \) can be the \( \emptyset \)). If \( \tau(s) \neq \emptyset \) and \( s' \in \tau(s) \) then \( s' \) is called a \( \tau \)-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 \( \tau I \) the idling transition. The transitions in the set \( TD = T \setminus \{ \tau I \} \) are called diligent.
\begin{itemize}
\item Θ is an assertion called the initial condition of \( T \). A state \( s \in \Sigma \) such that \( s \models Θ \) is called an initial state of \( T \).
\end{itemize}

Each transition \( τ \in T \) is characterized by an assertion relating the values of the variables in states \( s : Π \rightarrow D \) and \( s' : Π \rightarrow D \) for each \( s' \in τ(s) \). Denote this relation by \( ρ_τ \). Formally, if \( s : Π \rightarrow D \) and \( s' : Π \rightarrow D \), then \( ∀u \in Π (s[u], s'[u]) \in ρ_τ \). Using infix notation this is expressed by \( s[u] ρ_τ s'[u] \). 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 \( τ \). \( u' \) is called the primed version of \( u \). That is, we may think of two copies \( Π \) and \( Π' \) of the variable of \( T \), and denote the transition relation of \( τ \) by \( ρ_τ(Π, Π') \). The transition relation \( ρ_τ(Π, Π') \) has the form \( ρ_τ = C(Π) \wedge (y_i = e_i) \wedge \ldots \wedge (y_i = e_k) \) where:

\begin{itemize}
\item \( C(Π) \) is an assertion called the \textit{enabling condition} that depends only on the values of the variables in the state \( s \) before the transition. \( C(Π) \) states the condition under which \( s \) may have a \( τ \)-successor, i.e., \( τ(s) \neq ∅ \).
\item \( (y_i = e_i) \wedge \ldots \wedge (y_i = e_k) \) 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., \( ∀s' \in τ(s)(s'_[y_i] = s[e_i]) \), where \( y_1, y_2, \ldots, y_k \) are pair wise distinct.
\end{itemize}

\textbf{Notation:}

The transition relation \( ρ_τ(Π, Π') = C(Π) \wedge (y_i = e_i) \wedge \ldots \wedge (y_i = e_k) \) can be denoted by

\[ ρ_τ : C, \wedge (\bar{y} = \bar{e}) \]

where

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

If \( τ \in T \) and \( s \in Σ \) then if \( τ(s) = ∅ \) we say that \( τ \) is enabled on \( s \) and if \( τ(s) = ∅ \) we say that \( τ \) is disabled on \( s \). Notice that if \( ρ_τ : C, \wedge (\bar{y} = \bar{e}) \) then \( τ \) is enabled on \( s \) if \( s \models C \). For a set of transitions, \( T \subseteq T \) and \( s \in Σ \) we say that \( T \) is enabled on \( s \) if there is \( τ \in T \) and \( τ \) is enabled on \( s \); \( T \) is disabled on \( s \) if for each \( τ \in T, τ \) is disabled on \( s \). A state \( s \in Σ \) is called \textit{terminal} if the only enabled transition on \( s \) is the idling transition \( τ_\text{idle} \). Clearly all successors of a terminal state are terminals.

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

\textbf{Definition 1.} Let \( T = <Π, Σ, T, Θ> \) be a transition system. A computation \( σ \) of \( T \) is an infinite sequence of states \( σ : s_0, s_1, … \) that satisfies the following requirements:

\begin{itemize}
\item \textbf{Initiation:} \( s_0 \) is initial, that is, \( s_0 \models Θ \).
\item \textbf{Consecution:} \( \forall i (s_{i+1} ∈ τ(s_i)) \) for some \( τ ∈ T \).
\item \textbf{Diligence:} either \( σ \) contains infinitely many diligent \( τ \)-steps for \( τ ∈ T \) 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.
\end{itemize}

\section{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 \textit{abstract data types}.

An \textit{action} specification is a program in the language used to express reactive systems and is provided in the specification by the keyword \textbf{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 \( A :: [D][A_1 || A_2 || \ldots || A_n] \) where the following notation is used:

\begin{itemize}
\item \( A \) is the name of the action performed by the system.
\item \( D \) is a sequence of typed lists of variables of the form \textbf{mode} \textit{List} : \textit{type} where \( φ \) where \textbf{mode} is one of in, out, inout, local, \textit{type} is a type of value accepted in the system, and \( φ \) is an assertion satisfied by the variables in the \textit{List}.
\item \( A_1, A_2, …, A_n \) 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 \([D_i]; S_i \) 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 \( A_i(\text{arg}) \) where \( \text{arg} \) is the list of variables used by \( A_i \) for its arguments. Arguments can
be: \textbf{in}, \textbf{out}, \textbf{inout}. The \textbf{in} arguments are imported and not modifiable, the \textbf{out} arguments are exported and \textbf{inout} arguments are imported and modifiable.

\textbf{Definition 2.} A process is a tuple

\[ P = \langle \text{Agent}, \text{Action}, \text{Status} \rangle \]

where:
- \text{Agent} is a processor capable to perform statements composing the actions in \text{Action} and \text{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 \( l : \text{body} : \hat{l} \), 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 \textit{skip}, \textit{await} and \textit{assignment} and are defined as follows:

- The \textit{skip} statement has the form \( l : \text{skip} : \hat{l} \) and its performance means “skip”.
- The \textit{await} statement has the form \( l : \text{await } e : \hat{l} \) where \( e \) is a boolean expression and its performance means “wait until \( e \) becomes true”.
- The \textit{assignment} statement has the form \( l : (x_1, x_2, \ldots, x_n) := (e_1, e_2, \ldots, e_n) : \hat{l} \) denoted by

\[ l; x := e; \hat{l} , \]

where

\[ x = (x_1, x_2, \ldots, x_n), \ e = (e_1, e_2, \ldots, e_n), \]

and for \( i = 1, 2, \ldots, n \), \( x_i \) and \( e_j \) 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 \( \varepsilon \). This language of actions can be freely extended in order to express various computation performed by different systems.

\section{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 \( e : SSL_{\text{Sem}} \rightarrow SSL_{\text{Syn}} \) is defined as follows: if \( A : [D[A_1] || A_2] \ldots || A_n] \) is an action in \( SSL_{\text{Syn}} \) then \( e(A) \) is the transition system \( TS_A = \langle \Pi_A, S_0, T_A, \Theta_A \rangle \) 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 \( s \in S_0 \) is an assignment \( s : \Pi_A = D_A \) where \( D_A = \bigcup_{x \in \Pi_A \text{Type}(x)} L_A \).
- \( T_A \) is the set of transitions determined by the statements of \( A \). The idling transition is \( \rho_T : T \).
- The initial condition of \( \Theta_A \) is \( \Theta_A = (\pi = \text{Entry}(A_1) \cup \ldots \cup \text{Entry}(A_n)) \land \phi \) where \( \phi \) is the conjunction of all \textbf{where} assertions in the expression of \( A \).

A computation performed by the transition system \( TS_A \) is expressed by the sequence of transition

\[
\langle \pi, x_1, \ldots, x_n \rangle \xrightarrow{\rho_T} \langle \pi', x_1, \ldots, x_n \rangle \xrightarrow{\rho_{T_2}} \ldots
\]
where \( \{ \pi, x_1, \ldots, x_n \} \models \Theta \) and \( S_1, S_2, \ldots \) are statements of the action performed by \( TS_i \).

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 transition 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 \( A \) by \( A_1 \). Then two actions \( A_1 \) and \( A_2 \) are called congruent, denoted by \( A_1 \approx A_2 \), if \( TS(A_1) \sim TS(A_2) \) for every action \( A(S) \). Example of congruent actions are provided by the associatively of the operators "\( \cdot \)" (concatenation), "\( \lor \)" (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 \( \mathcal{R}(TS_{A_1}) = \mathcal{R}(TS_{A_2}) \). 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 validation language. Therefore, we use \( C \) as the software system validation language.

References: