International Mathematical Forum, 1, 2006, no. 22, 1077 - 1098

A Graph Theoretic Semantics for Viewcharts

Javad Mehri and Ayaz Isazadeh

Faculty of Mathematical Sciences
Tabriz University, Tabriz, Iran
JMehri@TabrizU.AC.IR

Abstract

Most of current methods for systems behavioral specifications are suitable only for small systems. Behavioral requirements engineering of large-scale software systems using current Formal Description Techniques (FDT), is complex and difficult. In general, when the scale of the system grows linearly, the number of states (in FSM-based methods) grow exponentially. Therefore, much research continues on introducing new techniques for eliminating (or at least reducing) this problem.

The introduction of Viewcharts is a considerable step in this direction. Work on the semantics of Viewcharts, however, is limited to its semantics via translations to Statecharts. We believe that due to the visual nature of the Viewcharts formalism, a graph theory based semantics is more suitable to establish a sound foundation for the formalism. Furthermore, since FSMs are more established and have stronger foundation than Statecharts, we prefer an FSM-based semantics. In this paper we provide a semantics for Viewcharts based on solely FSMs and graph theory.

Mathematics Subject Classification: 68N30

Keywords: Software Requirements Engineering, Finite State Machines (FSM), Mealy Machines, Statecharts, Viewcharts

1 Introduction

Since 1960's computer scientists have shown special interest in Finite State Machines (FSM). This was because of FSMs simplicity and strong mathematical foundation. At first it was thought that FSMs can be used as a method of systems requirements specifications and, for small systems it was indeed a good method. However, turned out that as the scale of a system grows linearly, the complexity of its representing FSM grows exponentially. The next logical work was to extend FSM in hopes of overcoming this complexity. This work include Jahanian and Mok's *Modechart* [12], Hendricksen's *Augmented*

State-Transition Diagrams [6], Shaw's *Communicating Real-Time State Machines* [14], Harel's *Statecharts* [2, 3, 4, 5], and Isazadeh's *Viewcharts* [10, 8]. We believe that the Viewcharts formalism for its ability in combining the notion of *view* with an extended FSM provides a good method for large-scale systems requirements specification. There has been two semantics provided for Viewcharts: An algorithmic semantics [9] and a set theory based semantics [8], both of which, establish the semantics of Viewcharts via translation to Statecharts. Still, due to the visual nature of Viewcharts, it can be viewed as a graph. Thus a graph theory based semantics will be more suitable for Viewcharts. Furthermore, since FSMs are more established and have stronger foundation than Statecharts, an FSM-based semantics can provide a sound foundation for Viewcharts.

In this paper, we provide some introductory concepts from graph theory, extend Mealy machine [7] to include variables and conditions, and establish the semantics of Viewcharts via translation to our *Extended Mealy Machine (EMM).*

2 Graph Theoretical Background

In this section we provide some notations and operations from graph theory and introduce a new operation. *If D_1, D_2 are two digraphs with sets of vertices $V(D_1)$, $V(D_2)$ and sets of arcs $E(D_1)$, $E(D_2)$; then digraph $D = D_1 \times D_2$ is defined as follows:*

$$\begin{aligned} V(D) = \;& V(D_1) \times V(D_2) \\ E(D) = \;& \{((u_1, u_2), (v_1, v_2)) \mid \\ & (u_1 = v_1 \wedge (u_2, v_2) \in E(D_2)) \vee (u_2 = v_2 \wedge (u_1, v_1) \in E(D_1))\} \end{aligned}$$

If we associate each arc of a directed graph G with a member of a set L then G is called a labeled directed graph, and L is known as the set of labels. Formally, let G be a directed graph and a function l be defined on $E(G)$ as $l : E(G) \to L$, then G is a labeled directed graph. Now, for two labeled digraphs D_1 and D_2, we define labeled digraph $D = D_1 \times D_2$ by:

$$\begin{aligned} &\forall u_1, v_1 \in V(D_1) \wedge \forall u_2, v_2 \in V(D_2) \cdot \\ &\qquad ((u_1, u_2), (v_1, v_2)) = \begin{cases} l(u_1, v_1) & \text{if} \quad u_2 = v_2 \\ l(u_2, v_2) & \text{if} \quad u_1 = v_1 \end{cases} \end{aligned}$$

A digraph D is called *complete symmetric digraph* if for every two distinct vertices u and v, both arcs (u, v) and (v, u) are present in D. The complete symmetric digraph of order n is denoted by K_n^*.

We now define a new operation called restricted cartesian product of two (labeled) directed graphs. D is the cartesian product of D_1 and D_2 restricted

by $U \subseteq E(D_1 \times D_2)$ if:
$V(D) = V(D_1) \times V(D_2)$
$E(D) = E(D_1 \times D_2) \setminus U$

This operation is denoted by $\times_U$; that is, $D = D_1 \times_U D_2$. Note that if U is empty then $D_1 \times_U D_2 = D_1 \times D_2$.

3 Extended Mealy Machines

A Mealy Machine is defined as a tuple, $M = (Q, \Sigma, \delta, O)$, where Q is a set of states; Σ is an input alphabet, also called events; $\delta : Q \times L \rightarrow Q$ is a partial function called the transitions function (where $L = \Sigma \times O$); and finally O is an output alphabet, also called actions. Visually, an MM is represented by a *State Transition Diagram* (STD). We will consider an STD of an MM as a labeled digraph, where the states are represented as nodes and transitions are represented as labeled arcs.

Using variables and conditions in a machine can considerably reduce the number of states. MMs do not allow using variables or conditions. Therefore, we extend MMs to include variables and conditions and we call it the Extended Mealy Machine (EMM).

An EMM is defined as a tuple, $E_M = (Q, \Sigma, \delta, O, V, C)$, where the first 4 components are Mealy machine concepts. The next two components, V and C are (possibly empty) sets of variables and conditions.

The syntax of EMM is defined over the basic sets of states, transitions, primitive events, primitive conditions, and primitive variables. Using these basic sets of elements we define the extended sets of events, conditions, expressions and labels. There is the formal definitions:

The set of primitive variables is denoted by V_p. The set of expressions V is inductively defined by:

1. k is a number $\longrightarrow k \in V$
2. $v \in V_p \longrightarrow v \in V$
3. $v_1, v_2 \in V$, *op* is an algebraic operation $\longrightarrow \mathit{op}(v_1, v_2) \in V$

The set of primitive events is denoted by Σ_p. The set of events Σ is inductively defined by:

1. $\epsilon \in \Sigma$, the NULL event
2. $e \in \Sigma_p \longrightarrow e \in \Sigma$
3. $e_1, e_2 \in \Sigma \longrightarrow e_1 \wedge e_2, e_1 \vee e_2 \in \Sigma$

4. $v \in V$, j is a value which v can take $\longrightarrow$ changed(v^j) $\in \Sigma$

   changed(v^j) is an event that occurs at a point in time when the value of variable v changes to j. We will also use the notation v^j to represent the fact that the value of variable v is equal to j.

5. $c \in C \longrightarrow$changed(c^T), changed(c^F) $\in \Sigma$

   changed(c^T) or changed(c^F) are events that occurs at a point in time when c becomes TRUE or FALSE, respectively. We will also use the notation c^T or c^F to represent the fact that the condition c is TRUE or FALSE, respectively.

changed(x) is abbreviated to $ch(x)$.

The set of primitive conditions is denoted by C_p. The set of conditions C is inductively defined by:

1. TRUE, FALSE $\in C$
2. $c \in C_p \longrightarrow c \in C$
3. $e \in \Sigma \longrightarrow \neg e \in C$
4. $u, v \in V, \mathcal{R} \in \{=, >, <, \neq, \leq, \geq\} \longrightarrow u\mathcal{R}v \in C$
5. $c_1, c_2 \in C \longrightarrow c_1 \vee c_2, c_1 \wedge c_2 \in C$

TRUE and FALSE are abbreviated to T and F, respectively.

The set of actions (output alphabets) O is inductively defined by:

1. $\epsilon \in O$, the NULL action.
2. $c \in C_p, d \in C \longrightarrow c := d \in O$
3. $v \in V_p, u \in V \longrightarrow v := u \in O$
4. $e \in \Sigma_p \longrightarrow e \in O$
5. $c \in C_p \longrightarrow c :=\mathrm{T}, c :=\mathrm{F} \in O$
6. $a_i \in O, i = 0, \ldots, n \longrightarrow < a_0, \ldots, a_n > \in O$

   Any sequence of actions is also an action.

The set of labels is defined as follows:
$L = \{e[c]/a \mid e \in \Sigma, c \in C, a \in O\}$
If $e = \epsilon$, $c =$T or $a = \epsilon$, we normally omit e, c, or a, respectively, from the labels.

Visually, an EMM is represented by a labeled digraph, where the set of states, and transitions are used as the set of vertices, and arcs, respectively. This labeled digraph is bounded by a box. The variables of an EMM are listed at the top-left corner of the corresponding box.

3.1 Introducing conditions to Mealy Machines

If we represent an EMM, which includes a set of conditions C, by a digraph D, then the equivalent MM is represented by:

$$\left(\prod_{c \in C} K^*_{|dom(c)|}\right) \times_U D$$

where C is the set of conditions of the EMM, the set of vertices of $K^*_{|dom(c)|}$ is $dom(c) = \{\text{T, F}\}$, and $U = \{((c_1^{v_1}, c_2^{v_2}, \ldots, c_n^{v_n}, x), (c_1^{v_1}, c_2^{v_2}, \ldots, c_n^{v_n}, y)) | \ \exists e \in \Sigma, a \in O \cdot n = |C| \ \wedge \ v_i = \text{F} \ \wedge \ l(x, y) = e[c_i]/a\}$. Specifically. let $E = (Q, \Sigma, \delta, O, \emptyset, C)$ be an EMM. We define an MM, $M = (Q', \Sigma', \delta', O')$, where

$Q' = \{(c_1^{v_1}, c_2^{v_2}, \ldots, c_n^{v_n}, s) | \ s \in Q \wedge n = |C| \wedge c_i \in C, v_i \in \{\text{ T, F}\} \ (i = 1, \ldots, n)\}$
$\Sigma' = \Sigma_p \cup \{ch(c^x) | \ c \in C \ \wedge \ x \in \{\text{F, T }\}\}$
Define a function $f_M : Q \longrightarrow 2^L$ as
$f_M(s) = \{l \mid l \in L \ \wedge \ \exists s' \in Q \cdot \delta(s, l) = s'\}$.
$L' = \{e/a | \ e \in \Sigma', a \in O'\}$
$\delta'((c_1^{v_1}, c_2^{v_2}, \ldots, c_n^{v_n}, s), l) =$

$$\begin{cases} (c_1^{v_1}, c_2^{v_2}, \ldots, c_n^{v_n}, \delta(s, l)) & \text{if} \quad \exists e \in \Sigma, a \in O \cdot l' = e/a \wedge \\ & \qquad \exists l \in L \cdot (l = l' \in f_M(s) \\ & \qquad \vee \ (l = e[c_i]/a \in f_M(s) \\ & \qquad \wedge \ (v_i = \text{T})) \\ (c_1^{v_1}, \ldots, c_j^{v'_j}, \ldots, c_n^{v_n}, s) & \text{if} \quad l = \text{“}ch(c_j^{v'_j})\text{”} \\ (c_1^{v_1}, c_2^{v_2}, \ldots, c_n^{v_n}, s) & \text{otherwise} \end{cases}$$

$O' = O$

To see that E and M are equivalent, consider the fact that for every behavior of E there is a corresponding behavior in M and vice versa. Notice that any change in a condition of the EMM corresponds to a change in the state of the MM.

Figure 1 and 2 show an example of an EMM and the equivalent MM respectively.

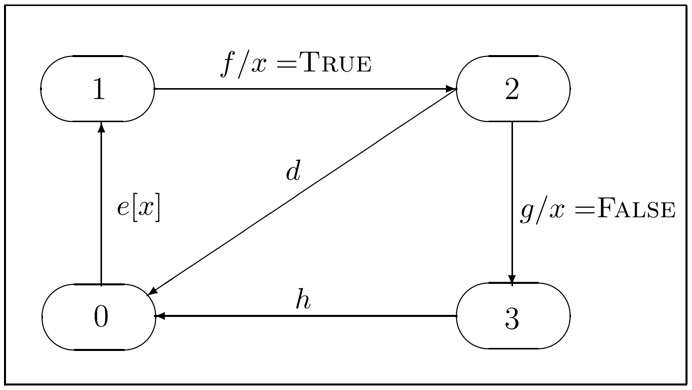

Figure 1: An EMM with the condition x.

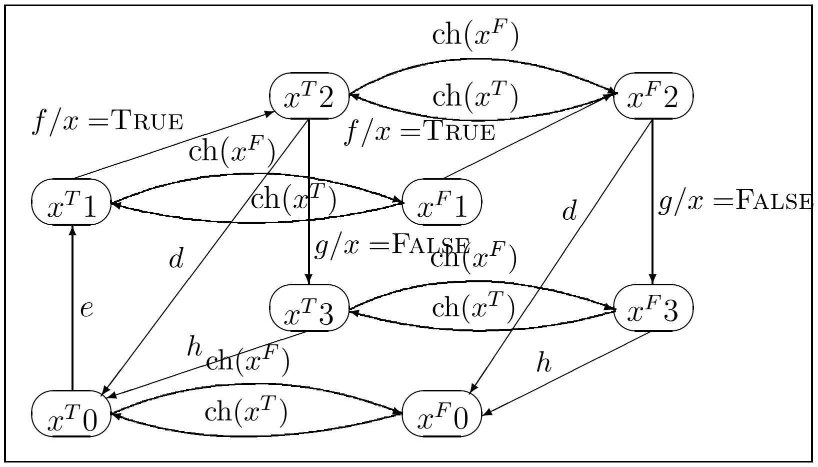

Figure 2: The MM equivalent to the above EMM.

Consider the state $x^T 2$ in the MM which is equivalent to the state 2 in the EMM where the value of x is TRUE. Now, if the system triggers g, then the action $x =$FALSE will be generated and state changes to 3 with x becoming FALSE. In the equivalent MM if the system triggers g, then with the generation of the action $x =$FALSE the event $ch(x^F)$ will occur. Next, the state will be $x^F 3$ which is equivalent to the discussed EMM.

3.2 Introducing variables to Mealy Machines

Variables can be introduced to MM either using actions and conditions or independently using graph theoretical concepts. In the first case we use actions and conditions. In an EMM a variable may be set or checked in two ways. In the case of setting a variable value we can use a simple action. For example, if the variable is x and we want to set its value to a we simply introduce an action of the form $/x = a$.

In the case of checking a variable value we can use a we simple condition. For example, if the variable is x and we want to see if the value of x is a, we simply introduce a condition of the form $[x = a]$.

In the second case we use graph theoretical notations. If we represent an EMM by a digraph D, then the equivalent MM is represented by:

$$\left(\prod_{x \in V} K^*_{|dom(x)|} \right) \times D$$

where V is the set of variables of the EMM, and $\{x^i | \, i \in dom(x)\}$ is the set of vertices of $K^*_{|dom(x)|}$. Specifically, let $E = (Q, \Sigma, \delta, O, V, \{\epsilon\})$ be an EMM. We define an MM, $M = (Q', \Sigma', \delta', O')$, where

$$Q' = \left\{ \left(x_1^{i_1}, x_2^{i_2}, \ldots, x_m^{i_m}, s \right) \middle| \, s \in Q \;\; \wedge \;\; m = |V|, \;\; x_j \in V \;\; (j = 1, 2, \ldots, m) \;\wedge\; i_j \in dom(x_j) \right\}$$

$$\Sigma' = \Sigma_p \cup \{ ch(x^i) | x \in V \;\wedge\; i \in dom(x) \}$$

$$\delta'((x_1^{i_1}, \ldots, x_m^{i_m}, s), l') =$$

$$\begin{cases} (x_1^{i_1}, \ldots, x_j^{i'_j}, \ldots, x_m^{i_m}, s) & \text{if} \quad \exists a \in O \cdot l = ch(x_j^{i'_j})/a \\ \left(x_1^{i_1}, \ldots, x_m^{i_m}, \delta(s, l) \right) & \text{otherwise if} \;\; \exists e_1, e_2 \in \Sigma', \exists a \in O \cdot \\ & \quad (l' = e_1/a \;\vee\; l' = e_2/a) \\ & \quad \wedge \, (\exists l = e_1 \vee e_2/a \in L) \\ \left(x_1^{i_1}, \ldots, x_m^{i_m}, \delta(s, l') \right) & \text{otherwise if} \quad l' \in L \\ \left(x_1^{i_1}, \ldots, x_m^{i_m}, s \right) & \text{otherwise} \end{cases}$$

$$O' = O$$

To see that E and M are equivalent, consider the fact that for any behavior of E there is a corresponding behavior in M and vice versa. Notice that any change in a variable of the EMM corresponds to a change in the state of the MM.

4 Viewcharts

In his Ph.D. thesis[8], Isazadeh introduces the notion of software *behavioral views*. Intuitively, this is a complete description of the behavior of a system observable from a specific point of view. A client's view of a server, for example, is the behavior that the client expects from the server. Isazadeh argues that *a fully-developed methodology based on views would significantly reduce the complexity of creating and understanding software requirements*[11]. He takes some efficient steps towards such a methodology. He define a formal notation, *Viewcharts*, with well-defined semantics based on Statecharts. In fact, Viewcharts can be viewed as an extension of Statecharts. It extends Statecharts to include behavioral views and their hierarchy of composition. This extension enables a systems analyst to specify the behavior of a large scale system by means of its simple views. In Viewcharts, the analyst does not have to specify the full behavior of a system and, therefore, is not concerned with the complexity or scale of the system. That is all he needs is to specify different views of the system and compose them.

In the remainder of this section we provide some informal descriptions of the Viewcharts concepts. The following section is dedicated to the formal description of the formalism.

4.1 Ownership of Elements

In general, we say an element (event, actions or variables) *belongs to* the view that declares it. Consequently the scope of an element is limited to the view that owns the element. On the other hand composition of views may require communication between the composed views; the scope of an event in one view, for example, may be extended to cover other views. An action belongs to the view(or views) that generates(or generate) the action. Similarly, a variable belongs to the view that declares it. The scope of a variable declared by a view is the view and all its subviews. An event or an action may have multiple owners while variables cannot. This notion of ownership, in Viewcharts, adds name space control to limit the scope of broadcast communication, solving a problem with Statecharts.

4.2 Composing Behavioral Views

Views can be composed in three ways: SEPARATE, OR, and AND. Except for the effect of ownership and scoping restrictions the OR and AND compositions of views, in Viewcharts, are similar to the OR and AND compositions of states, in Statecharts, respectively. The SEPARATE composition of views, however, is specific to Viewcharts. In a SEPARATE composition of views, all the views are active if any one of them is active [1] and no transition between the views is allowed.

The OR composition is similar to SEPARATE composition except that in an OR composition only one view can be active, but there can be transitions between the views. Notice that a transition from a source view to a destination view interrupts the source view, i.e., takes the system out of any state(s) of the source view; it is, therefore called an *interrupt transition.* In case of a conflict between the interrupt transition and one internal to the source view, the interrupt transition has higher priority.

In an AND composition of views, all the views are active. The scope of all elements owned by each view are extended to the other views.

Viewcharts adopts *Harel's synchrony hypothesis* that events are instantanous. Specifically, events, action, and checking the value of a condition expression ideally take no time; therefore, transitions are also instantanous[4].

5 Formal Description

Two types of semantics have already been presented for Viewcharts: A set theoretic foundation and an algorithmic semantics. However, since Viewcharts is a visual formalism and basically a graph, we believe that a graph theoretic foundation is more suitable. Basically, a Viewcharts consist of a hierarchical composition of views. Our approach is to consider the leaves of a viewchart as an independent EMM. To provide a semantic basis for Viewcharts, then, we need to show the result of different compositions of views is also an EMM. First some introductory and informal remarks:

The SEPARATE composition has been explained by renaming the events as in [8]. Here we also rename the events and then replace SEPARATEed views by ANDed views and continue the process upwards in the hierarchy of views.

OR composition can be explained by a "restricted" higraph in the following sense: Using graph notations, the vertex A in figure 3(a) is a hivertex that contains vertices and edges. The edge (A, B) means that there are edges from all vertices in A to all vertices in B (i.e. $\{(a, b)|a \in V(A) \;\wedge\; b \in V(B)\}$). In fact figure 3(a) and figure 3(b) are equivalent. When we declare a vertex in B as a default vertex, represented by an arrow on top of the vertex, there are edges from each of vertices in A to the default vertex in B. We define

[1]A view is active whenever the system is in a state of the view.

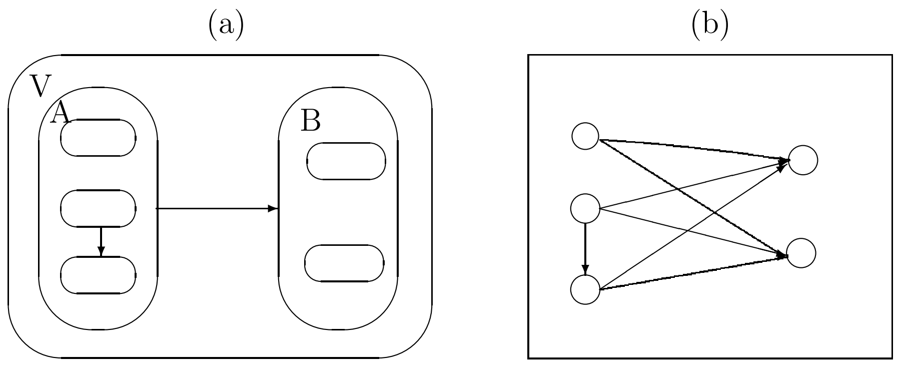

Figure 3: An OR-view and its equivalent EMM.

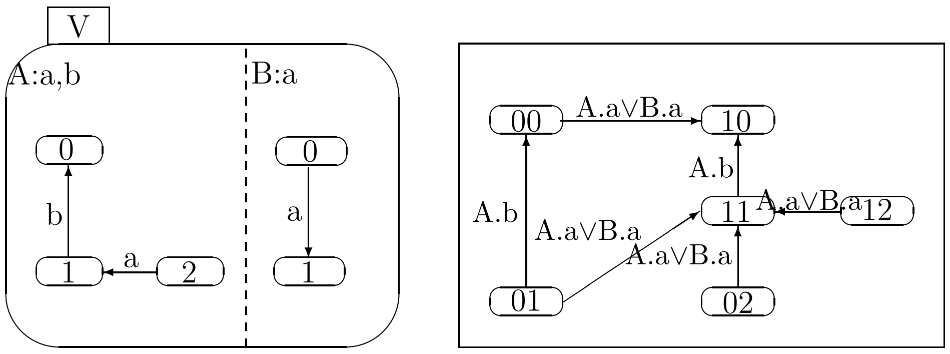

Figure 4: An AND-view and its equivalent EMM.

OR composition only between two or more views that are in the same level[2], therefore, one cannot define an OR composition between views in different levels. Intuitively we say the transitions corresponding to OR compositions cannot cross view boundaries.

The third composition is AND composition. Unlike OR composition, when two or more views are ANDed with each other, if their superview is active, then all of them are active. If we illustrate each view by a graph, then the AND composition of views is the cartesian product of the corresponding graphs.

[2]Two views are said to be in the same level, when both of them are immediate subviews of a view.

5.1 Definitions

This section is an exact adaptation from [8]. We will use the following primitives from Viewcharts.

- U is a set of views.
- V is a set of variables.
- E is a set of events and actions.
- T is a set of transitions.
- $D \subseteq U$ is a set of default views.
- $o : U \rightarrow 2^{\bar{E} \cup V}$ is an ownership function.
- $\tau : U \longrightarrow \{\text{AND, OR, SEP}\}$ is a type function.
- $\rho : U \rightarrow 2^U$ is a hierarchy function.
- r is the root view.

$\rho(u)$ is the set of *immediate subviews* of u; $\rho^+(u)$ is the set of *all subviews* of u; and $\rho^*(u)$ is $\{u\} \cup \rho^+(u)$. $\sigma(u)$ is the set of *basic views*, they are the leaves of the hierarchy. A basic view is an EMM.

Formally, $u \in \bar{U}$ is defined as $u = (Q_u, \Sigma_u, \delta_u, O_u, V_u, C_u)$ For $u \in U$ and $l \in o(u)$, $\psi(l, u)$ is the scope of $u.l$ (the scope of element l with respect to u).

For $u \in U$, $l \in E_u \cup V_u$ and $v \in V_u$, define

$$\theta(l,u) = \{x | u \in \psi(l,x)\}$$
$$\mu_v(v,u) = \begin{cases} v & \text{if} \quad \theta(u,v) = \emptyset \\ x.v \text{ where } x \in \theta(v,u) & \text{otherwise} \end{cases}$$
$$\mu_e(e,u) = \begin{cases} \{e[\mu_v(v,u)/v]\} & \text{if} \quad \theta(e,u) = \emptyset \\ \bigvee\limits_{x \in \theta}(e,u)x.e[\mu_v(v,u)/v] & \text{otherwise} \end{cases}$$

For $t = ((x,(e,a),y)) \in T$, define

$$\eta((x,(e,a),y)) = (x, e[\mu_e(\bar{e},u)/\bar{e}], a[\mu_e(\bar{a},u)/\bar{a}]), y) \text{ where } \exists_1 u \in U \cdot t \in T_u \wedge \bar{e} \in E_u \wedge \bar{a} \in E_u$$

We define an interface Viewcharts as follows:

$$V' = \{\mu_v(v,u) | u \in U, v \in V_u\}$$
$$T' = \{\eta(t) |\ t \in T\}$$
$$E' = \{e' | \exists (x,(e,a),y) \in T' \cdot e' = e \vee e' = a\}$$
$$\bar{E}' = \{\mu_e(e,u) | u \in U, e \in E_u\}$$
$$\tau'(s) = \begin{cases} \text{AND} & \text{if} \quad s \in U \wedge \tau(s) = \text{ SEP} \\ \tau(s) & \text{otherwise} \end{cases}$$

5.2 Semantics

For a given interface viewchart, $w = (U, E', V', T', D, o, \tau', \rho)$, we now construct an EMM, $E = (Q, \Sigma, \delta, O, V, C)$, as follows:

Assume, without loss of generality, that each AND-view in the interface viewchart contains exactly two immediate ANDed subviews.

For $u \in U$ define

$$Q^{[u]} = \begin{cases} Q_u & \text{if} \quad u \in \bar{U} \\ Q^{[u_1]} \times Q^{[u_2]} & \text{if} \quad u_1, u_2 \in \rho(u) \ \wedge \ and(u_1, u_2) \\ \bigcup\limits_{i=1}^{n} Q^{[u_i]} & \text{if} \quad \#\rho(u) = n \ \wedge \ \forall i = 1, \ldots, n \\ & \quad (u_i \in \rho(u) \ \wedge \ \exists j \cdot or(u_i, u_j)) \end{cases}$$

For $u \in U$ define

$$\Sigma^{[u]} = \begin{cases} \Sigma_u & \text{if} \quad u \in \bar{U} \\ \Sigma^{[u_1]} \ \cup \ \Sigma^{[u_2]} & \text{if} \quad u_1, u_2 \in \rho(u) \ \wedge \ and(u_1, u_2) \\ \left(\bigcup\limits_{i=1}^{n} \Sigma^{[u_i]}\right) \cup E'_u & \text{if} \quad \#\rho(u) = n \ \wedge \ \forall i = 1, \ldots, n \\ & \quad (u_i \in \rho(u) \ \wedge \ \exists j \cdot or(u_i, u_j)) \end{cases}$$

For $u \in U$ define

$$q_0^{[u]} = \begin{cases} s & \text{if} \quad u \in \bar{U} \ \wedge \ \exists s =\in Q_u \\ q_0^{[u_1]} & \text{if} \quad u_1 \in \rho(u) \ \wedge \ u_1 \in D \\ s & \text{if} \quad \rho(u) \cap D = \emptyset, \exists s \in \bigcup\limits_{u' \in \rho(u)} q_0^{[u']} \end{cases}$$

For $u \in U$ define

$$\delta^{[u]}(s, l) =$$
$$\begin{cases} \delta_u(s, l) & \text{if} \quad u \in \bar{U} \\ \left(\delta^{[u_1]}(s_1, l), \delta^{[u_2]}(s_2, l)\right) & \text{if} \quad \exists u_1, u_2 \in \rho(u), s_1 \in Q^{[u_1]}, s_2 \in Q^{[u_2]}. \\ & \qquad \wedge \ and(u_1, u_2) \ \wedge \ s = (s_1, s_2) \\ x & \text{if} \quad \exists u_1, u_2 \in \rho(u), s_1 \in Q^{[u_1]} \\ & \qquad \wedge \ or(u_1, u_2) \ \wedge \ (u_1, l, u_2) \in T'_u \ \wedge \ x \in q_0^{[u_2]} \\ \delta^{[u_1]}(s, l) & \text{if} \quad \exists u_1, u_2 \in \rho(u), s_1 \in Q^{[u_1]} \\ & \qquad \wedge \ or(u_1, u_2) \ \wedge \ (u_1, l, u_2) \notin T'_u \end{cases}$$

$$V^{[u]} = \bigcup_{w \in \rho^*(u)} V'_w$$

$$C^{[u]} = \bigcup_{w \in \rho^*(u)} C'_w$$

With above definitions we have $Q = Q^{[r]}$, $\Sigma = \Sigma^{[r]}$, $\delta = \delta^{[r]}$, $O = E'$, $V = V^{[r]}$, and $C = C^{[r]}$.

6 Example: Manufacturing Control System

This section presents an example, a Manufacturing Control System (MCS). An informal, but detailed, description of a similar system is given by Dunietz and others [1].

A system can be specified at different levels of abstraction. We choose a level of abstraction that illustrates the Viewcharts notation. Further refinements of a viewchart, beyond a certain level of abstraction, can be Statecharts tasks and may not provide any additional information in illustrating Viewcharts.

Consider a "flexible"[3] and "just-in-time"[4] manufacturing shop. It consists of a number of workstations, where each workstation performs a certain process on the product. Figure 5 shows an informal diagram representing the flow of product and information in the shop. Our objective is to specify the behavioral requirements of a MCS for this shop.

Central to the system is a database server (DBS) which maintains and supplies the information requirements of the workstations. At the beginning of the manufacturing line, the first workstation associates each product with a unique identification number/string *pid*, which must be communicated to DBS to create a record for the corresponding product. From there on, each product is identified and tracked by the associated *pid*. When a product arrives at a workstation, the *pid* is scanned and communicated to DBS which, in turn, informs the workstation of the process that must be performed on the product. The workstation then proceeds with the process and when it is completed, informs DBS to update the product record.

Considering that concurrent processes are performed on different products at different workstations, DBS may receive concurrent transaction requests. If we model DBS as a single entity which interacts with multiple workstations, then we must also specify the way in which DBS handles concurrent transactions. Doing so, not only complicates the specification, but also requires making design decisions regarding the concurrency. We can, however, simplify the specification and leave the design issues to designers by modeling the behavior of DBS as a collection of behavioral views that it exhibits to the workstations. Each workstation then interacts with its own view of DBS on a one to one basis.

Furthermore, considering that the purpose of this example is not to specify

[3]The term *flexible* refers to the capability of the shop to handle the manufacturing process of different types of products. A flexible circuit pack manufacturing shop, for example, may handle the manufacturing process of hundreds of different circuit packs.

[4]The term *just-in-time* refers to the capability of the shop in the on-time delivery of the products which are manufactured on the basis of actual orders (as opposed to anticipated orders). Such a shop requires that different components of a given product, at different stages of its manufacturing process, should come together just in time.

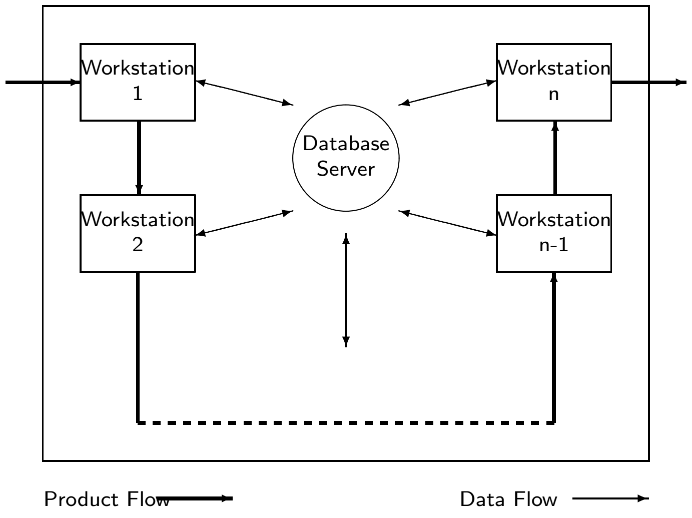

Figure 5: Product and information flow in a manufacturing line.

the details of a database management system, we use a single but compound variable *db* to represent the MCS database. We refer to a product record (a component of *db*) by a compound variable *prec*, which includes a *pid* and some other product attributes. *db.prec* then refers to the product record *prec* in the database and *db.prec.pid* is a product ID. We also use the notation *db.prec*(*pid*) to refer to the product record of the given *pid*.

Database transactions are time-consuming activities; they cannot be represented by events or actions (which are instantaneous). However, we can use actions like *add*, *retrieve* or *update* to initiate the corresponding transactions. Similarly we can use events like

added(*db.prec*), abbreviated as **ad**(*db.prec*),
retrieved(*db.prec*(*pid*)), abbreviated as **rt**(*db.prec*(*pid*)), or
updated(*db.prec*(*pid*)), abbreviated as **ud**(*db.prec*(*pid*))

which occur at a point in time when the associated transaction is completed.

With this introduction, we can now specify the behavioral requirements of MCS as a composition of its behavioral views.

6.1 Specifying Behavioral Views

The viewchart $\texttt{WS}_1$, shown in Figure 6, describes the behavior of the system observable at the Serialisation Workstation, which is the first workstation in the manufacturing line. It consists of two ANDed views: `WS`, which describes

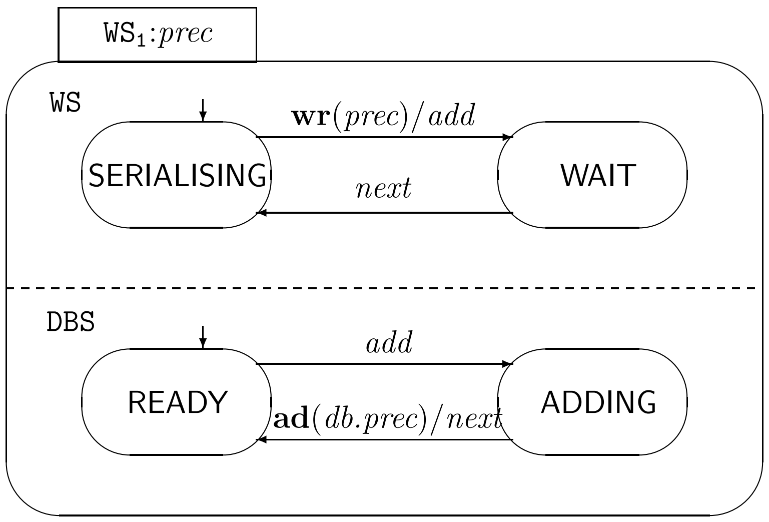

Figure 6: A viewchart for the Serialisation Workstation.

the behavior of the workstation, and DBS, which describes the workstation's view of DBS.

The declaration "$\mathtt{WS_1}$:*prec*", in this figure, shows that *prec* is owned by $\mathtt{WS_1}$ and so its scope is $\mathtt{WS_1}$. *db* is not declared anywhere in $\mathtt{WS_1}$ and so it is global to $\mathtt{WS_1}$. All other elements in $\mathtt{WS_1}$ are implicitly declared; they belong either to `WS` or `DBS` and their scope is $\mathtt{WS_1}$. the event **ad**(*db.prec*) can only be triggered by the state `ADDING` and consequently belongs to `DBS`. The scope of **ad**(*db.prec*), which is originally `DBS`, because of the AND composition of `WS` and `DBS` is extended to cover $\mathtt{WS_1}$.

`WS` specifies that the workstation by default is in the state of `SERIALISING`, where each product is associated with a unique product ID and a product record is prepared and written to the compound variable *prec*. When the writing is completed, the event **wr**(*prec*), which is an abbreviation for **written**(*prec*), occurs which, in turn, generates *add*. This, in turn, takes `DBS` to the state of `ADDING`. `DBS` in this state adds *prec* to the database and when it is done the event **ad**(*db.prec*) occurs, generating the action *next* and taking both `DBS` and `WS` back to their starting states.

The other workstations, at the abstraction level of this specification, have identical behaviors. A SEPARATE composition of $n-1$ behavioral views, as shown in Figure 7, can specify the behavior of the system observable at these workstations. Each views can be further refined to describe the specific and detailed behavior at the corresponding workstation.

The declaration "$\mathtt{WS_2}, \ldots, \mathtt{WS_n}$:*prec, pid*", in this figure, shows that for each view $\mathtt{WS_i}(\mathtt{i} = 2, \ldots, \mathtt{n})$, the elements *prec* and *pid* belong to $\mathtt{WS_i}$ and so their scope is $\mathtt{WS_i}$. *db* is not declared anywhere in $\mathtt{WS_i}$ and so it is global to $\mathtt{WS_i}$. All other elements in $\mathtt{WS_i}$ are implicitly declared; they belong either to $\mathtt{WS_i}$, `WS` or

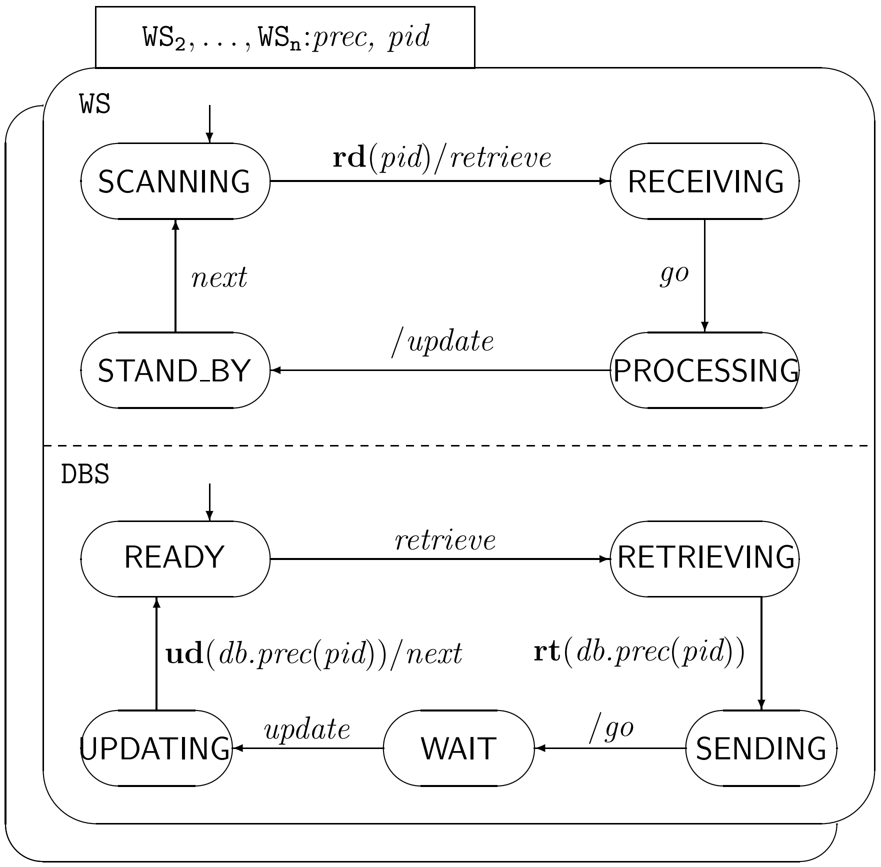

Figure 7: The workstations' views of the system.

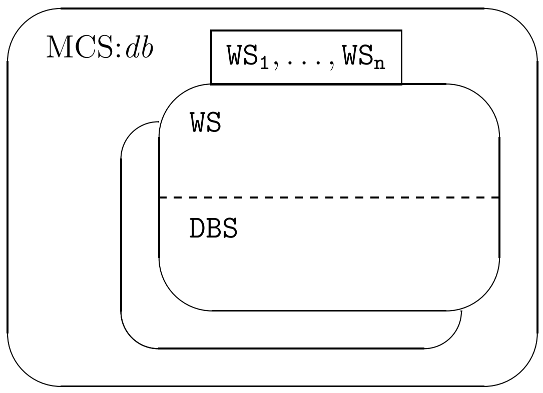

Figure 8: A viewchart for a manufacturing line.

$\texttt{WS}_\texttt{i}$.DBS and their scope is $\texttt{WS}_\texttt{i}$.

The event **rd**(*pid*), which is an abbreviation for **read**(*pid*), occurs at a point in time when a *pid* is read (scanned). When WS is in the state of RECEIVING, it expects instructions from DBS. On the other hand, DBS retrieves the product record and based on the history and type of the product sends out the appropriate instructions[5] regarding the process to be performed on the product.

While WS is in the state of PROCESSING, it may provide DBS with certain information regarding the status of the product and/or outcome of the process. This information is included in the product record and its effect on the system behavior can be specified by refining the PROCESSING and other affected states.

Figure 7 should now be self-explanatory.

6.2 Composing Behavioral Views

Having specified the behavioral views of the system, we can now compose them to form the overall system behavioral requirements specification. Figure 8 shows a SEPARATE composition of n views where each view describe the behavior of the system from a workstation's point of view.

Notice, once again, the declarations "$\texttt{WS}_1$:*prec*" of Figure 6, "$\texttt{WS}_2, \ldots, \texttt{WS}_\texttt{n}$:*prec, pid* " of Figure 7, and "MCS :*db*" of Figure 8. These declarations mean that

[5]Some examples of the instructions are outlined below:

- "Reject (wrong station) and reroute to the station x, where x is a station ID."
- "Perform the current process on the product."
- "Perform a different process: transmitting the required program or instructions."
- "Ship" or "Do not ship"; at the shipping station.
- "Transmitting defect information;" at the repair stations.

These and similar details can be specified by refining the states.

each view $\mathtt{WS_i}(\mathtt{i} = \mathtt{2}, \ldots, \mathtt{n})$ has its own variables *pid* and *prec*, $\mathtt{WS_1}$ has its own variable *prec*, and *db* is global to all $\mathtt{WS_i}(\mathtt{i} = \mathtt{1}, \ldots, \mathtt{n})$. All the views can access and update the same database *db*, while they have their local variables for the information retrieved from, or to be added to, the database.

6.3 Discussion

The EMM specification of this example, containing a large number of states and transitions, is practically impossible to represent in this paper. So, we represent its Statecharts specification. The Statecharts specification of this example would consist of $n + 1$ orthogonal components: one for each workstation and another one for DBS. If the manufacturing line consists of only a few workstations, then there is no problem; however, the specification becomes complex when the number of workstations increases. Figure 9 shows an attempt to specifying MCS in Statecharts. Each orthogonal component, in this figure, is basically the corresponding view of Viewcharts specification. however, the figure is not a valid statechart; it has complications and issues that must be resolved.

First of all, the example requires a notation to represent the repetition of workstations. In Viewcharts we represented this repetition simply by a SEPARATE composition of views. In statecharts we need a capability to represent the repetitions in orthogonal components. Statecharts does not have this capability; however, Leveson's extension of Statecharts, RSML [13], provides a method for representing the repetitions. Assuming that Statecharts is extended to support this capability, we have to also parameterize the components to distinguish between the elements of one component from those of the others. For example, the Viewcharts description of a workstation, for all workstations but the first one, specifies a local variable *pid*, which is used for passing a product ID from `WS` to `DBS`. To provide this specification in a global environment of Statecharts, we have to parameterize this variable to distinguish the product IDs being scanned (concurrently) by different workstations. The following are some of the other complications with the diagram of figure 9:

- The event like **rd**(*pid*), which occur in different orthogonal components, are ambiguous. We have to specify the workstation that reads *pid*.
- Different occurrences of a variable like *pid* or *prec* in different orthogonal components are different. We have to replace each variable by an array of variables.
- The actions like *add, retrieve, update, go* and *next*, which occur in different orthogonal components, are also ambiguous. We have to provide

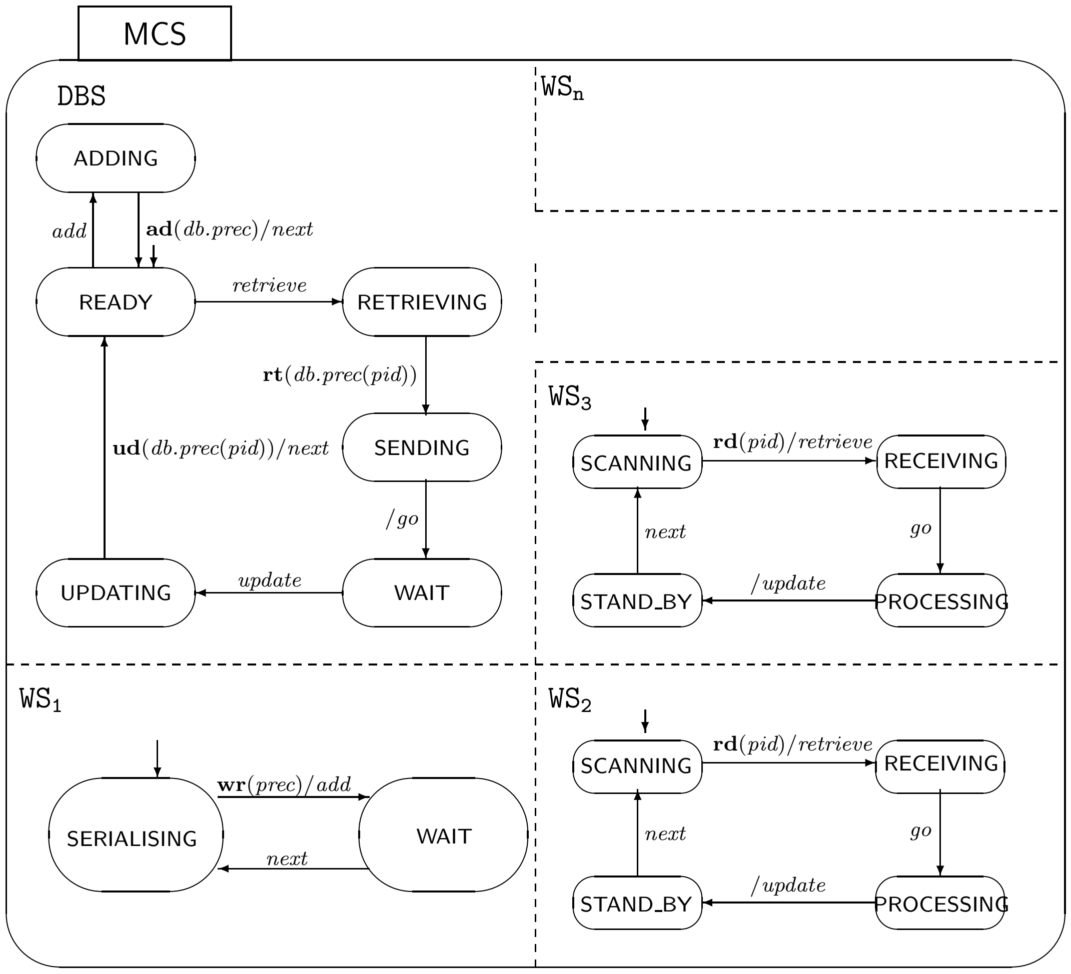

Figure 9: An attempt to specify MCS in Statecharts.

more detail to resolve the ambiguity; e.g., we have to specify which product record to be added, retrieved or updated or to which workstation the action *next* or *go* is intended for.

All these details are also provided by Viewcharts specification of MCS, but not explicitly by the specifier. In the Viewcharts notation, all these details, the parametrization, is within the structure of the notation; there is no need for the explicit parametrization.

Suppose that an action to update a product record is generated by a workstation while DBS is in a state of updating another one. DBS must be ready, at all times, to sense events that are indications of database transaction requests. The transactions are time-consuming tasks; DBS must be ready for new events while processing transactions. In Viewcharts, this creates no complexity, because each workstation is interacting with its own view of the database on a one-to-one basis. In Statecharts, however, we have to introduce an additional orthogonal component, a queuing mechanism, which is always ready to sense the events, queue the corresponding transaction requests, and pass them to DBS as it becomes available for accepting requests. An additional orthogonal component does not necessarily complicate the specification. However, this component is a queuing mechanism, which has nothing to do with the observable behaviour of the system, it is there only for making the specification possible (or simple); and later in the design phase, a designer may or may not choose to use the mechanism. Viewcharts does not require such a mechanism, which shows the independence of Viewcharts from the design issues.

In specifying behavioral requirements of a system, it is a weakness for a notation to require using any mechanism internal to the system. Using such a mechanism (in any level of abstraction) is an added complexity. It may also violate the independence of requirements from the design and implementation. Viewcharts allows specification of the requirements to be expressed only in terms of observable events and, unlike Statecharts, does not force using any internal mechanism of the system.

7 Conclusion

By introducing conditions and variables to MM, we have defined a new machine, EMM. Using graph theory and set theory we have shown that EMMs and MMs are equivalent. We have then presented the semantics of Viewcharts via translation to EMM. The translation is rigorous and shows that for a given Viewchart there exists an equivalent EMM. Our semantics, therefore, is based on sound concepts from graph theory and Finite State Machines, described using rigorous mathematical approach, and consequently provides a strong foundation for Viewcharts. This work should help systems requirements engineers use Viewcharts with more confidence.

References

[1] I. S. Dunietz , J. L.C. Hsu , M. T. McEachern , J. H. Stocking , M. A. Swartz, and R. M. Trombly, MPCS–the manufacturing process control system. AT&T Technical Journal, 65(4):35-45, July 1986.

[2] D., Harel, Statecharts: A visual formalism for complex systems. **Science of Computer Programming**, 8:231-274, 1987.

[3] D. Harel, On visual formalisms. **Communications of the ACM**, 31(5):514-530, May 1988.

[4] D. Harel and A. Naammad, **The STATEMATE semantic of Statecharts**. Technical Report, i-Logix, Inc., 22 Third Avenue, Burlington, Mass. 01803, USA, November 1995.

[5] D. Harel and A. Pnueli, On the development of reactive systems. In K. R. Apt, editor, **logics and Models of Concurrent Systems**, page 477-498. Springer-Verlag, New York, 1985.

[6] C.S. Hendricksen, Augmented state-transition diagrams for reactive software. **ACM SIGSOFT Software Engineering Notes**, 14(6) pages 61-67, October 1989.

[7] J.E. Hopcroft and J.D. Ulman, Introduction to automata theory, languages, and computation, Addison-wesley, Reading, Mass. 1979

[8] A. Isazadeh, Behavioral views for software requirements engineering. Ph.D. thesis, Department of Computing and Information Science, Queen's University, Kingston, Canada, September 1996.

[9] A. Isazadeh and D.A. Lamb, An Algorithmic Semantic for Viewcharts. **second IEEE International Conference on Engineering of Complex Systems (ICECCS'96)**, page 293-296, Montreal, Canada, October 1996. IEEE Computer Society Press.

[10] A. Isazadeh and D.A. Lamb, and MacEwen G.H. Viewchart: A Behavioral Specification Langugae for Complex Systems. **In Proceedings of International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)**, page 208-215, Honolulu, Hawaii, April 1996. IEEE Computer Society Press.

[11] A. Isazadeh and D.A. Lamb, and Shepard T. Behavioral Views for Software Requirements Engineering. **J. Requirements Eng.**, 4, Pages 19-37, 1999.

[12] F. Jahanian and A.K. Mok, Modechart: A specification language for real-time systems. **IEEE Transaction on Software Engineering**, 20(12), pages 933-947, December 1994.

[13] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth and J. D. Reese, Requirements specification for process control systems. **IEEE Transaction on Software Engineering**, 20(9), pages 684-707, 1994.

[14] A.C. Shaw, Communicating real-time state machines. **IEEE Transaction on Software Engineering**, 18(9), pages 805-816, September 1992.

Received: October 14, 2005