INFORMATICA, 2008, Vol. 19, No. 3, 345–362

Viewcharts: Syntax and Semantics

Ayaz ISAZADEH, Jaber KARIMPOUR
Department of Computer Science, University of Tabriz
29 Bahman Blvd., 5166616471 Tabriz, Iran
e-mail: karimpour@tabrizu.ac.ir

Received: 3 September 2006; accepted: 3 July 2007

Abstract. In this paper, we present a method for describing the syntax and semantics of viewcharts. Viewcharts is a visual formalism for describing the dynamic behavior of system components. We define the syntax of viewcharts as attributed graphs and, based on this graph, describe dynamic semantics of viewcharts by object mapping automata. This approach covers many important constructs of viewcharts, including hierarchy of views, ownership of elements, scope, and composition of views in SEPARATE, OR and AND ways. It also covers completion and interlevel transitions as well as history transitions without violating the independence of views. Viewcharts was originally based on statecharts; in this paper we also change the basis of viewcharts to an extended version of Finite State Machine (EFSM).

Keywords: visual languages, viewcharts, syntax definition, formal operational semantics, object mapping automata.

1. Introduction

The most costly errors, as Parnas states (Parnas, 2000), are those made early in the process. Requirements specification is an early phase in the software engineering process. Statistically speaking, studies in Bell Labs and IBM have shown that 80% of all defects in software systems are inserted in the requirements phase. A complete and precise requirements specification, therefore, is an important part of the software engineering process. The Precise specification of the requirements, in turn, requires a formal method. For large-scale software systems, however, using current formal methods can be complex and difficult. The difficulties of managing a large volume of formal specifications have made formal methods impractical for large-scale systems, Partly due to the way in which formal specifications are presented. Large volumes of specifications presented textually, many pages of mathematical and/or logical statements, are indeed difficult to produce, read, understand, and verify, specially for the user side. We may accept that the specifiers are requirements engineers, who are technical people and are supposed to be experts in producing formal specifications. However, the users who are supposed to at least read, understand, and verify the specifications are not normally too enthusiastic to get involved with such formal texts. The *presentation* of formal specifications, therefore, is an important factor for practicality of formal methods. This is one factor that the software industry has been resisting against using formal approaches in software engineering

practices. Visual formalisms are introduced as an attempt to simplify the presentations of formal specification. Visual formalisms are generally FSM-based techniques, the most popular of which is statecharts.

1.1. *Complexity of Scale*

In conventional finite state machines, the number of states grows exponentially as the scale of the system grows linearly. This growth leads to a blow-up in the number states for large-scale systems. Drusinsky and Harel (Drusinsky and Harel, 1988; Drusinsky and Harel, 1994) prove that statecharts is exponentially more succinct than finite state machines. The proof is based on the cooperative concurrency mechanism (i.e., orthogonality) of statecharts and can be applied to other models that use this mechanism such as Petri Nets (Peterson, 1977) or CSP (Davies, 1993; Hoare, 1978). If we assume that an increase in the scale of a system results in additional orthogonal components in the corresponding statechart, then the number of states in the statechart has a linear relationship with the scale of the system. Orthogonality is, infact, a powerful feature in statecharts. However, it is not clear that any increase in the scale of a system does result in additional orthogonal components. For example, if an increase in the scale of a system, corresponds to additional complexities in the existing orthogonal components, then the increase in the number of states would still be exponential.

Another problem with statecharts is the *global name space*. There is no "visibility" control mechanism in statecharts. (The term *visibility* is defined in terms of *declaration*, *scope*, and *binding*; a visibility control mechanism, essentially, refers to a mechanism that controls scope (Wolf *et al.*, 1988).) When an event occurs, it is sensed throughout the system, so it must have a unique name. Managing the name space in the global environment of statecharts, for large scale software systems, can be difficult. Name management, in general, is one of the fundamental issues in software engineering (Kaplan and Wileden, 1995).

These problems, specially the *complexity of scale*, are inevitable problems as log as we try to specify the system as a whole. How practical is it to accurately and fully describe a system while we can only work with our limited "views" of the system? One can only describe one's "view" of the world. A user's attempt to specify a system, at best, can only result in the specification of his or her "view" of the system. The viewcharts formalism (Isazadeh *et al.*, 1999) accepts this reality and specifies the behavior of a system, formally and visually, as a composition of "views". And, that is the formalism which we will work with.

1.2. *Problem*

The overall and general problem is to discover whether there can be a practical and useful formal method for behavioral requirements specification of large and complex systems.

A solution to this problem is to pick (and if necessary modify) an existing formalism, or introduce a new one, with the following desirable characteristics:

1. Must be visual and thereby simple to use.
2. Must solve the problem of *complexity of scale* in the best possible way.
3. Must solve the problem of *global name space* in the best possible way.
4. Must have precise and sound syntax and semantics.
5. Must allow the possibility of system modeling, simulation and verification.
6. Must allow the possibility of formally reasoning about the system behavior.

The problem addressed by this paper is to present the solution, satisfying the items 1–4 above, and setting the stage ready for a future work on the next two items.

1.3. *Paper Outline*

The paper is organized in four sections. In this section, we have stated the problem and have provided the desirable characteristics of the proposed solution. The list characterizes a good solution, basically, as a formal method based on a sound syntactic and semantic foundation. Section 2 provides a brief overview of viewcharts, which is the notation of our choice, and the related work on this notation. Section 3 describes our solution by presenting the sound syntactic and semantic foundation for viewcharts. Finally, Section 4 returns to the list of desirable characteristics of our solution, presents a discussion of the viewcharts' syntax and semantics with respect to the list, and concludes the paper with a summary of the results and future directions.

2. Previous Work

In this section, we provide a brief overview of the viewcharts formalism (a separate paper (Isazadeh *et al.*, 1999) describes it in more detail) and discuss the related work on the syntax and semantics of this formalism.

2.1. *Overview of Viewcharts*

As mentioned above, the viewcharts formalism (Isazadeh *et al.*, 1999) attempts to specify the behavior of a system as a composition of "views". Intuitively, a *view* (or *behavioral view*) 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. The caller view of a telephone set and the telephone set's view of a switching system are also examples of behavioral views.

Using this notion of view, viewcharts is designed to specify the behavioral requirements of large-scale complex systems on a *need-to-specify* basis. In viewcharts, one does not have to specify the full behavior of a system and, therefore, is not concerned with the complexity or scale of the system. A complex system may consist of many different sub-systems and components, distributed worldwide, and it may exhibit a combination of many different and identical behavioral views. Current research and industrial advances in networking and distributed systems indicate that software systems will continue to get larger and more complex. One cannot envision producing an integrated behavioral

requirements specification for an arbitrarily large and complex system. However, if we define the behavior of a system in terms of behavioral views, then all we need to do is to specify the views of our interest. The viewcharts formalism allows these views to be specified independent of each other. Furthermore, views in viewcharts limit the scope of broadcast communication, solving the problem of *global name space*.

2.1.1. *The Origin of Viewcharts*

Introduced by Ayaz Isazadeh in his Ph.D. thesis (Isazadeh, 1996), the viewcharts notation was defined based on statecharts (Harel, 1987; Harel and Naamad, 1995). Statecharts, however, has no concept of behavioral views. Viewcharts extends statecharts to include views and their hierarchical compositions. The leaves of the hierarchy, described originally by independent statecharts, represent the behavioral views of the system or its components. The higher levels of the hierarchy are composed of the lower level views.

The first thing we do in this paper is to use classic finite state machines, with some extensions (EFSM), to represent the leaves of the viewcharts hierarchy. This would provide us with a firm foundation for our syntax and semantics.

2.1.2. *Ownership of Elements*

Viewcharts normally limits the scope of an element (event, action, or variable) to a given view. However, 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. In a given view, therefore, viewcharts must distinguish two different types of events:

- Events that *belong* to (or are *owned* by) the view: These are the events that the view *can trigger*. They must be declared by the view.
- Events that *do not belong* to the view: The view cannot trigger these events. An event of this type can occur only if it is triggered elsewhere and if the view is covered by the scope of the event.

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 action may have multiple owners.

Syntactically, elements owned by a view can be declared by listing them following the name of the view either in the viewchart, as in Fig. 1, or out of it as a separate text. It may be necessary to identify a view by its fully or partially *qualified name*, which consists of the *base name* prefixed by the names of its ancestors in the hierarchy separated by dots.

2.1.3. *Composing Behavioral Views*

Views can be composed in three ways: ş, o̧r, and a̧nd compositions. Except for the effect of ownership and scoping restrictions, the o̧r and a̧nd compositions of views, in viewcharts, are similar to the o̧r and a̧nd compositions of states, in statecharts. The ş composition of views, however, is specific to viewcharts. In a composition of views, similar to the notion of depth in statecharts, the composed views form a superview which is an *encapsulation mechanism*, inherent to the composition.

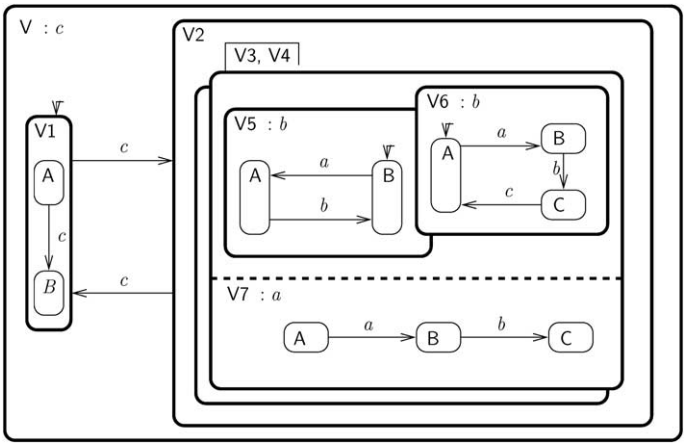

Fig. 1. Composition of views in a viewchart (from (Isazadeh, 1996)).

In a ş composition of views, all the views are active if any one of them is active;[1] no transition between the views is allowed; the scopes of all the elements are unaffected; and any subview or state in one view is hidden from (i.e., cannot be referenced by) the other views. Visually, the şed views are drawn on the top of each other, giving the impression that they are located on different planes and, consequently, are hidden from each other. As shown in Fig. 1, these views are either identical, like V3 and V4, or different, like V5 and V6.

The o̧r and ş compositions are similar, except that in an o̧r composition, only one view can be active and there can be transitions between the views. Like the ş composition, any subview or state in one view is hidden from (i.e., cannot be referenced by) the other views. In Fig. 1, for example, the view V consists of an o̧r composition of V1 and V2.

In an a̧nd composition of views, all the views are active; the scopes of all the elements owned by each view are extended to the other views. All the subviews and states in one view are visible to (i.e., can be referenced by) the other views. The viewchart of Fig. 1, for example, is composed of a ş composition of V5 and V6, which in turn is a̧nded with V7 forming V3. A ş composition of two identical views V3 and V4 forms V2. The full view V is an o̧r composition of V1 and V2.

2.2. *Related Work*

There is a body of work on statecharts semantics in the literature; e.g., (Jin *et al.*, 2004; Beeck, 2002; Borger *et al.*, 2000). In (Michelle *et al.*, 2005), there is the result of a comparative literature survey on approaches to formally capturing the semantics of UML state machines; it categorizes and compares 26 different approaches.

Work on the formal syntax and semantics of viewcharts, however, is limited to the following two major cases:

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

1) *a set theoretic-based semantics of Viewecharts* (Isazadeh, 1996) and,
2) *an algorithmic semantics of viewcharts via translattion to statecharts* (Isazadeh and Lamb, 1996).

In the first case, Isazadeh establishes a sound foundation for the formalism by providing a set theoretic-based semantics. The resulting semantics, however, does not have any tool support and is not suitable to interface with modeling tools, violating the characteristics 5 and 6 of our solution. The second case is an attempt to establish the semantics of viewcharts via translating it to statecharts. This attempt, at its best, would loose any advantages that viewcharts has compared to statecharts, violating the characteristics 2 and 3 of our solution. Consequently, a formal semantics, providing an unambiguous interpretation of viewcharts diagrams, independent of statecharts is needed.

3. Syntax and Semantics of Viewcharts

We use a predicate-based approach, the Graph Type Definition Language (GTDL) (Janneck, 2000; Janneck, 1998a), for defining the syntax and static semantics of viewcharts. The reason for choosing this approach is mainly due to the strong tool support from the Moses tool suite (Janneck, 1997).

For the dynamic semantics viewcharts, we present an operational approach using Object Mapping Automata (OMA) (Janneck, 2000; Janneck, 1998b). In this approach, the attributed graph of a well-formed viewcharts diagram is compiled into OMA algebraic structures. Based on these structures, the diagram is interpreted as an OMA composed of two rules: an *initialization rule* and a *run-to-completion rule*. The first rule includes the tasks for setting the initial system states (system configuration), when the viewcharts is instantiated. After the initialization, the OMA iteratively executes the second rule, allowing the viewcharts instance to dispatch and process events in its event queue.

Our method for describing the syntax and semantics of viewcharts is inspired by Jin Yan and others (Jin *et al.*, 2004), where the syntax of UML statechart represented by the Graph Type Definition Language (GTDL), which is a small domain-specific language and part of the Moses tool suite (Janneck, 1997). Well-formedness rules are represented as predicates over the abstract syntax of these graphs. The semantics of UML statechart is then represented as Object Mapping Automata (OMA) (Janneck, 2000; Janneck, 1998b). The Moses tools suite allows users to edit, simulate and automatically analyze such a system (Jin *et al.*, 2004; Janneck, 1997). In this section we expand Jin Yan and others method (Jin *et al.*, 2004) to present an approach for describing the syntax and semantics of viewcharts.

3.1. *Syntax Definition*

This section briefly outlines the description of the syntax of viewcharts in the Graph Type Definition Language (GTDL). GTDL is used in the Moses project to define new graph types (Janneck, 2000; Janneck, 1998b). The definition of a graph type basically consists of two parts:

- A list of the kinds of vertices and edges in graphs of this type, together with definitions of their attributes and information about their graphical appearance.
- A list of constraints (predicates) on an attributed graph structure which must be fulfilled for the graph to be well-formed.

Attributes are simply name-value pairs, where technically the name is a Java String Object, while the values may be any objects suitable as an assignment for the attribute.

Given viewcharts, its abstract syntax (kinds of elements and their connectedness), is defined as an *attributed graph*. Formally, an *attributed graph* is a tuple:

$$(Ver, Edg, src, dst, cntr, \alpha),$$

where

- Ver and Edg are disjoint sets of vertices and edges,
- src, dst: $Edg \rightarrow Ver$ are total functions, mapping each edge to its source and target vertices.
- $cntr$: $Ver \rightarrow Ver$ is a partial *container function* mapping a vertex $v \in Ver$ to its *container*.
- α: $(Ver \cup Edg) \times A \rightarrow U$ is a partial *attribute function* mapping a graph object $o \in Ver \cup Edg$ and an *attribute name* in A to an *attribute value* in U. Here, we assume A and U are universal sets of attribute names and values respectively.

The *attribution function* α contains all information except for the connection structure of the graph. It is important to choose U to contain a relevant set of different kinds of attribute values, e.g., attributes may contain subgraphs.

The syntax of viewcharts is defined in two parts: (1) a definition of language-specific *semantic attributes* and *appearance* of graph objects (such as states and transitions) and (2) a specification of the *well-formedness* rules of viewcharts.

Fig. 2 illustrates a part of the GTDL syntax specification for viewcharts. After attribute definition, types of graph objects are defined in specification. These include three kinds of vertices (composite view, basic view and history pseudostates) and one kind of edge (transations). In defining each vertex or edge, the semantic attributes of this type are listed. For example, in Compositeview vertex, an attribute hold the graph type of that graph for describing hierarchy of views. In addition to semantic attributes, each type of graph object also can have attributes representing their graphical appearance (e.g., shape, color and default size). Apart from these, a composite view have additional boolean attributes (attribute is s, if it is a s composition and is $Cuncurrent$, if it is an and composition).

Predicates describe the well-formedness rules of viewcharts (Number 6–9). Each predicate defines a boolean formula which must be true for every well-formed viewcharts diagram. A predicate my be declared to declare attributes of a vertex. For instance, predicate P_2 declares local elements (events, actions and variables) of a vertex and predicate P_3 shows that the visible elements of a given view are elements either declared by it or declared by its superview. Predicate P_4 shows that each view can only trigger its visible events.

graph type Viewcharts{
1. **attribute set** *Events*, *Actions*, *Variables*.
2. **vertex type** Compositeview(**String** Name,
   graph(Viewcharts, Subview, Basic View),
   bool isş, **bool** isConcurrent)
   graphics(**string** shape= “RoundRect", ...)
3. **vertex type** Basicview(**String** Name, **graph**(EFSM))
   graphics(**string** shape= “RoundRect", ...)
4. **vertex type** History(**bool** isDeep)
   graphics(**string** shape=“Ellipse", Lable= if **Deep** then “H*" else “H”, ...)
5. **edge type** Transition (**expr** trigger, **expr** guard, **expr** action)
   graphics(...).
6. **predicate** $P_1 : \forall t \in \mathbf{Transition}$:
   $src(t)$ and $dst(t) \in$ Basicview+History+Compositeview
7. **predicate** $P_2 : \forall v \in \mathbf{Vertex}$:
   $\alpha(v, LocallyElements) = Events + Actions + Variables$
8. **predicate** $P_3 : \forall v \in \mathbf{Vertex}$:
   $\alpha(v, VisibleElements) = LocallyElements +$
   $\alpha(cntr(v), VisibleElements)$
9. **predicate** $P_4 : \forall t \in \mathbf{Transition}$:
   $t(trigger) \in \alpha(v, VisibleElements)$

... }

Fig. 2. Syntax definition of viewcharts(abridged).

As mentioned in Section 2, in viewcharts a basic view is defined as an statechart. In our definition viewcharts, however, we use an Extended version of Finite State Machine (EFSM) to represent a basic view. Fig. 3 describes the syntax specification of the EFSM.

3.2. *Semantics Definition*

In this section, we use Object Mapping Automata (OMA) as a formal language to define the semantics of viewcharts. We specify the semantics in two steps: structure formalization and operational semantics definition. At the first step we compile the attributed graph representing viewcharts diagram into OMA algebraic structures. then, we specify the operational semantics of the diagram by two rules: (1) an initialization rule is executed when an instance of the viewcharts diagram is created and initializes the state configuration of the instance; (2) a run-to-completion rule starts to run after the initialization and iteratively processes events one at a time.

graph type EFSM{

1. **vertex type** Initial()
   graphics(**string** shape="BlackDot",**int** width=8,**int** width=8).
2. **vertex type** State(**string** Name)
   graphics(**string** shape= "RoundRect", **color** Fillcolor="white",...)
3. **vertex type** History(**bool** isDeep)
   graphics(**string** shape="Ellipse", Lable= "H", ...)
4. **vertex type** Final(...)
   graphics(**string** shape= "CircledDot", ...).
5. **edge type** Transition (**expr** trigger,**expr** guard,**expr** action)
   graphics(...).
6. **predicate** $p_1 : \forall e \in Transition$:
   $src(e) \in$ Initial + History+ State
   $dst(e) \in$ History+ State + Final

... }

Fig. 3. Syntax definition of EFSM(abridged).

3.2.1. *OMA Algebraic Structure Formalization*

Given the attributed graph of a viewcharts diagram $(Ver, Edg, src, dst, cntr, \alpha)$, we first compile it into OMA algebraic structures.

At first, we classify the view and state vertices in Fig. 4. views and states are identified according to attribute "type" of the vertices including composite views V_c, basic views V_b, simple states S_s, final states S_f and initial states S_i. According on attribute "isConcurrent" and "isş", V_c is further divided into three disjoint views: aṇd views V_{cc}, ş views V_{se} and o̧r views V_{sc}. As mentioned in 2.1.3, in ş and o̧r composition of views, the scopes of all the elements are unaffected, but in aṇd composition of views, the scopes of all elements owned by each view are extended to other views, we have shown this fact in number 3 of Fig. 4. Finally, S_s, S_f and S_i constitute the set of states S. We define three kinds of pseudostates: initial P_i, shallow history P_{sh} and deep history P_{dh}. Each pseudostate include set of views and states. We let $\mathsf{P}_h = \mathsf{P}_{sh} \cup \mathsf{P}_{dh}$ and $\mathsf{P} = \mathsf{P}_i \cup \mathsf{P}_h$. Also on the basis of the edges Edg, we let $T := Edg$ for set of transitions in viewcharts.

Then, we model the view hierarchy in the diagram, using the OMA. Fig. 5 includes the definitions of five relations over vertices. Function $cntr$ is a containment function mapping a vertex into the composite vertices directly enclosing it. For a vertex v at the top level we will have $cntr(\mathsf{v}) = \perp$. Function $subs(\mathsf{v})$ refers to the set of (direct) subviews of a given composite view $\mathsf{v} \in \mathsf{V}_c$. We let $subs(\mathsf{v})$ return an empty set for a simple states in Basic view. For example we have $subs(\mathsf{V}) = \{\mathsf{V1}, \mathsf{V2}\}$, $subs(\mathsf{V3}) = \{\mathsf{V5}, \mathsf{V6}, \mathsf{V7}\}$, $subs(\mathsf{V7}) = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ and $subs(\mathsf{A}) = \emptyset$ in Fig. 1. Function $default$ returns the initial view or state of a given view. For instance, we have $default(\mathsf{V}) = \mathsf{V}_1$ in Fig. 1. The boolean function "$cover$" determines whether views v transitively contains view u, in

1. $\mathsf{V}_c \equiv \{\mathsf{v} \in Ver | \alpha(\mathsf{v}, \text{“}type\text{”}) = \text{“}Compositeview\text{”}\}$
2. $\mathsf{V}_b \equiv \{\mathsf{v} \in Ver | \alpha(\mathsf{v}, \text{“}type\text{”}) = \text{“}Basicview\text{”}\}$
3. $\mathsf{V}_{cc} \equiv \{\mathsf{v} \in \mathsf{V}_c | \alpha(\mathsf{v}, \text{“}isConcurrent\text{”}), \alpha(\mathsf{v}, \text{“}VisibleElement\text{”}) := \bigcup_{\forall \mathsf{v} \in \mathsf{V}_{cc})} \alpha(\mathsf{v}, \text{“}VisibleElement\text{”})\}$
4. $\mathsf{V}_{se} \equiv \{\mathsf{v} \in \mathsf{V}_c | \alpha(\mathsf{v}, \text{“}is\text{”}\}$
5. $\mathsf{V}_{sc} \equiv \mathsf{V}_c \backslash (\mathsf{V}_{cc} \cup \mathsf{V}_{se})$
6. $\mathsf{V} \equiv \mathsf{V}_b \cup \mathsf{V}_c$
7. $\mathsf{S}_s \equiv \{\mathsf{v} \in \mathsf{V}_b | \alpha(\mathsf{v}, , \text{“}type\text{”}) = \text{“}Simple\text{”}\}$
8. $\mathsf{S}_f \equiv \{\mathsf{v} \in \mathsf{V}_b | \alpha(\mathsf{v}, \text{“}type\text{”}) = \text{“}Finale\text{”}\}$
9. $\mathsf{S}_i \equiv \cup\{\mathsf{v} \in \mathsf{V}_b | \alpha(\mathsf{v}, \text{“}type\text{”}) = \text{“}Initial\text{”}\}$
10. $\mathsf{S} \equiv \mathsf{S}_s \cup \mathsf{S}_f \cup \mathsf{S}_i$

Fig. 4. Views and states classification.

1. $cntr \equiv \{\mathsf{v} \rightarrow \mathsf{t} | \mathsf{v} \in \mathsf{V} \cup \mathsf{S} \cup \mathsf{P}, \mathsf{t} \in \mathsf{V}_c, cntr(\mathsf{v}) = \mathsf{t}\} \cup \{\mathsf{v} \rightarrow \mathsf{TOP} | \mathsf{v} \in \mathsf{V}, cntr(\mathsf{v}) = \perp\}$,
2. $subs \equiv \{(cntr(\mathsf{v}), \mathsf{v}) | \mathsf{v} \in \mathsf{V}, \mathsf{v} \neq \mathsf{TOP}\} \cup \{\emptyset | cntr(\mathsf{v}) \in \mathsf{V}_b\}$,
3. $default \equiv \{cntr(src(e)) \rightarrow dst(e) | e \in T, src(e) \in P_i\}$,
4. $cover \equiv \{(\mathsf{v}, \mathsf{u}) | \mathsf{v} \in \mathsf{V}, \mathsf{u} \in \mathsf{V} \cup \mathsf{S}, \mathsf{v} \in cntr^{+}(\mathsf{u})\}$,
5. $co\acute{v}er \equiv cover \cup \{(\mathsf{v}, \mathsf{v}) | \mathsf{v} \in \mathsf{V}\}$.

Fig. 5. View hierarchy decoding.

other words, whether there exists a sequence of composite views $\mathsf{v}_1, ..., \mathsf{v}_k \in \mathsf{V}_c$ for $k < |\mathsf{V}_c|$ such that $cntr(\mathsf{u}) = \mathsf{v}_1, ..., cntr(\mathsf{v}_k) = \mathsf{v}$. The function $cntr^{+}$ denotes the transitive closure of $cntr$. For instance in Fig. 1, $(\mathsf{V}, \mathsf{V2}), (\mathsf{V}, \mathsf{V5}), (\mathsf{V7}, \mathsf{A}) \in cover$ and $(\mathsf{V1}, \mathsf{V2}) \notin cover$.

3.2.2. *Initialization*

If we represent a system by a single Finite State Machine (FSM) (where the system can only be in one state of the FSM at any instant in time), then each state of the system corresponds to a set of the FSM. However, if we represent the system by an extended FSM such as viewcharts (where the system can be in many states of these machines at any instance in time), we will refer to the state of a system as the *configuration* of the system; a system *configuration* can be represented by a set of active views and states in viewcharts. To decode the behavior of a view, it is important to understand its configuration. We let $\triangle$ denote the current configuration of viewcharts in Fig. 6.

The initialization rule, shown in Fig. 6, initializes configuration of viewcharts. It first shows an auxiliary set φ of pending views and states to be entered. Initially ti contains the top view, and then running in a loop. The loop continues while φ is not empty. In each run of the loop, all views or states currently in φ are handled in parallel. For a OR view,

1. Set $\triangle = \emptyset$;
2. TOP:=Full VIEW
3. Initialize;
4. Set $\varphi := \{\mathsf{TOP}\}$.
5. $\forall \mathsf{v} \in \varphi$:
6. $\quad \triangle := \triangle + \{\mathsf{v}\}, \varphi := \varphi - \{\mathsf{v}\}$,
7. $\quad$ if $\mathsf{v} \in \mathsf{V}_{cc}$ then $\varphi := \varphi + subs(\mathsf{v})$;
8. $\quad$ if $\mathsf{v} \in \mathsf{V}_{sc}$ then $\varphi := \varphi + \{default(\mathsf{v})\}$;
9. $\quad$ if $\mathsf{v} \in \mathsf{V}_{se}$ then $\varphi := \varphi + subs(\mathsf{v})\}$;
10. $\quad$ if $\mathsf{v} \in \mathsf{V}_{b}$ then $\varphi := \varphi + \{default(\mathsf{v})\}$;
11. *genCmplevt*

Fig. 6. Initialization rules of viewcharts.

only its unique default subview is added, for an AND view, all its direct subviews are added, for a S composition of views all its subviews are added, while, for a Basic view, its initial state is added.

After the loop, the rule attempts to generate a completion event using a macro genCmplevt.

3.2.3. *Preliminary Definitions*

A viewcharts instance is ready to execute after the initialization. Before execute the execution rule, we introduce a few more macro definitions shown in Fig. 7. Firstly, given a transition $t \in T$, $lcp(t)$ maps it to the *Last Common Progenitor (LCP)* view of t, the lowest composite view that contains all the source and target of t. In other word $lcp(t)$ is the view covered by any other view in the chain of views hierarchy. Auxiliary macro cp determines the set of common progenitor for a given set of views or states. The $lcp(t)$ is decoded by considering composition of views and a consequently extended scope. The main sources $ms(t)$ of a transition t are either $src(t)$ if $lcp(t)$ is sequential or separate view, or union of $src(t)$ for all subviews in $lcp(t)$ if $lcp(t)$ is concurrent view. The main target $mt(t)$ of t has the same definition except that it ought to cover all the target views of t in the last case. In addition, the states or views that, if active, must be exited when t is executed are defined by $exited(t)$. This set includes the main source of t and all the views and states covered by it. For example in Fig. 1, we have $lcp(a) = \mathsf{V4}$, $ms(a) = \{\mathsf{V5.A}, \mathsf{V6.A}, \mathsf{V7.A}\}$ and $mt(a) = \{\mathsf{V5.B}, \mathsf{V6.B}, \mathsf{V7.B}\}$.

Recall that the scope of an element owned by a superview covers all its subviews. Therefore, different subviews can use an element owned by their superview in different values; doing so can result in conflicting views. Mean while in an AND composition, the scope of element covers all the composed views, so anding views can result in conflict (Isazadeh, 1996; Isazadeh *et al.*, 1999). The function $conflict\_views$ decode this condition. Also two transitions t and t' are *in conflict* if they can be enabled by the same event and also if firing them results in a common non-empty set of views or states to be

1. $cp \equiv \{X \mapsto \bigcap_{\mathsf{q} \in X} cntr^{+}(\mathsf{q}) | \mathsf{X} \subseteq \mathsf{V} \cup \mathsf{S} \cup \mathsf{P}_h\}$;
2. $lcp \equiv \{t \mapsto \mathsf{I} | t \in T, cntr(\mathsf{I}) \in \mathsf{V}_{ce} \cup \mathsf{V}_{sc}, \mathsf{I} \in cp(src(t) \cup dst(t)),$
   $\forall \acute{\mathsf{I}} \in cp(src(t) \cup dst(t)), cover^{'}(\acute{\mathsf{I}}, \mathsf{I}), t \in \alpha(\mathsf{v}, VisibleElements)\}$
   $\cup \{t \mapsto cntr(\mathsf{I}) | t \in T, cntr(\mathsf{I}) \in \mathsf{V}_{cc}, \mathsf{I} \in cp(src(t) \cup dst(t)),$
   $\forall \acute{\mathsf{I}} \in cp(src(t) \cup dst(t)), cover^{'}(\acute{\mathsf{I}}, \mathsf{I})\}$
3. $ms \equiv \{t \mapsto \mathsf{m} | t \in T, \mathsf{m} \in src(t), cntr(\mathsf{m}) = lcp(t) \wedge lcp(t) \in \mathsf{V}_{se} \cup \mathsf{V}_{sc}\},$
   $\cup \{t \mapsto \bigcup_{\mathsf{v} \in lcp(t)} \mathsf{v.m} | t \in T, \mathsf{m} \in src(t), lcp(t) \in \mathsf{V}_{cc}\}$
4. $ms \equiv \{t \mapsto \mathsf{m} | t \in T, \mathsf{m} \in dst(t), cntr(\mathsf{m}) = lcp(t) \wedge lcp(t) \in \mathsf{V}_{se} \cup \mathsf{V}_{sc}\},$
   $\cup \{t \mapsto \bigcup_{\mathsf{v} \in lcp(t)} \mathsf{v.m} | t \in T, \mathsf{m} \in dst(t), lcp(t) \in \mathsf{V}_{cc}\}$
5. $exited \equiv \{(t, \mathsf{v}) | t \in T, \mathsf{v} \in \mathsf{V}, cover^{'}(ms(t), \mathsf{v})\} \cup \{(t, \mathsf{s}) | t \in T, \mathsf{s} \in \mathsf{S},$
   $cover^{'}(ms(t), \mathsf{s})\},$
6. $conflict\_views \equiv \{(\mathsf{v}, \mathsf{v}^{'}) | \mathsf{v}, \mathsf{v}^{'} \in \mathsf{V}, \mathsf{v} \neq \mathsf{v}^{'}, x \in \alpha(\mathsf{v}, VisibleElement)$
   $\cap \alpha(\mathsf{v}^{'}, VisibleElement), value(x, \mathsf{v}) \neq value(x, \mathsf{v}^{'}),$
7. $conflict\_transitions \equiv \{(t, t^{'}) | t, t^{'} \in T, t \neq t^{'} \wedge trg(t) = trg(t^{'})$
   $src(t) \cap exited(t^{'}) \neq \emptyset\},$
8. $priority \equiv \{(t, t^{'}) \in conflict | t, t^{'} \in T, \exists \mathsf{v} \in src(t), \mathsf{u} \in src(t^{'}),$
   $cover(\mathsf{v}, \mathsf{u}) \wedge \forall \mathsf{v} \in src(t), \mathsf{u} \in src(t^{'}), \neg cover(\mathsf{u}, \mathsf{v})\}$

Fig. 7. Introductory definitions.

exited. Otherwise, they are called *consistent*. Relation $conflict\_transitions$ consists of pairs of transitions that are *in conflict* with each other. Clearly, *conflict* is irreflexive and symmetric.

Relation $priority$ specifies the firing priority between two conflicting transitions. More specifically, a transition originating from a vertex $\mathsf{v} \in \mathsf{V}$ has a higher priority than another transition originating from a vertex $\mathsf{v}^{'} \in \mathsf{V}$ such that $cover(\mathsf{v}, \mathsf{v}^{'})$. Given two transitions $t, t^{'}$, $(t, t^{'}) \in prior$ indicates that, if both enabled, $t^{'}$ has a higher priority to be executed than t. In the following, we shall use $conflict\_transitions(t)$ and $priority(t)$ for a given transition t to denote the set of transitions in conflict with t and the set of transitions with priority over t, respectively.

3.2.4. *Specifying Executable Steps*

We are now able to specify the event processing of an *active view* by Fig, 8. First of all, three global variables Q_{evt}, $cmplevt$ and hc are declared. Q_{evt} represents the event queue of a view. $cmplevt$ is a boolean variable indicating the existence of a pending completion event. Completion events are a special kind of event generated when some view is completed. They have priority over normal events in Q_{evt} to be processed. hc is a map associating a history pseudostate with a set of the last active views contained (or covered) by the composite view directly containing the pseudostate. Note that $\triangle$ is considered as a map from each view to a boolean denoting whether the view is active.

The $exeute$ rule firstly set events e triggered by active views in Q_{evt}, then this rule checks for a pending completion event and, if it succeeds, handles it using a macro

1. **set** $Q_{evt} = \emptyset$; **bool** $cmplevt = false$; **function** hc **arity** 1;
2. **rule** $exequte$:
3. $Q_{evt} = \{e \in \alpha(\mathsf{v}, VisibleElements) | \mathsf{v} \in \triangle\}$
4. **if** $cmplevt$ **then** $handleCmplevt$
5. **elseif** $Q_{evt} \neq 0$ **then**
6. **choose** $ce \in Q_{evt}$:
7. $Q_{evt} := Q_{evt} - \{ce\}$,
8. $enabled = \{t \in T | trg(t) = ce \wedge src(t) \subseteq \triangle \wedge \mathbf{eval}(grd(t), \triangle)\}$
9. $\mathbf{firing}(enabled)$

Fig. 8. Execute rule.

$handleCmplevt$ (described later). If no such an event exists, it randomly dequeues an event ce in the event queue (if any) of an active view. This event is called the current event. Next, a set of transitions enabled by ce is computed. This set consists of transitions whose source views or states are all currently active, whose triggers match the current event, and whose guards are evaluated to TRUE. The guards are evaluated by $\mathbf{eval}()$, which we have assumed is an external function provided by the runtime environment. When enabled transitions are present, a macro $\mathbf{firing}(enabled)$ (illustrate in Fig, 9.) is used.

The main job of **firing** is to compute and fire a maximal subset of the enabled transitions which are not in conflict with each other. This subset is called a *maximal consistent* set denoted by $firable$. In Fig. 9, initially, the set is empty. At each pass of the loop, a new transition is added that is enabled and consistent with all the existing transitions and over which no enabled transition has priority. The loop finishes when no transition can be added. Note that in computing $firable$, we accommodate a nondeterministic choice between conflicting transitions when the conflicts cannot be resolved by priority. This allows viewcharts to be used for capturing requirements at early stages of system design where many design decisions have not yet been made. With the maximal consistent set computed, we can execute the transitions in the set in parallel due to their mutual consistency. The execution of a single transition consists of a sequence of steps (Lines 7..10): exiting the main source, executing the transition effect, and entering the main target.

$\mathbf{Exit}(ms)$ macro in Fig. 10 involves exiting all views or states in given *main source(ms)*. At first, a local set variable φ records views and states to be exited. It includes the active states covered by the *main source*. Then, at each step of the loop, all states or views in φ are exited independently. Also, at each pass φ is refreshed with the containers of the exited states. Special attention is paid to a view that can be exited only if none of its substates or subviews are active (Line 5). For handle history transitions, Fig. 11, update history pseudostates for a given ms. In this figure H includes all the history pseudostates contained by active views and covered by ms. The contents of this set refreshed for shallow and deep history.

```
1. firing(enabled) ≡ begin
2. set firable = ∅;
3. loop
4.     choose t ∈ enable − firing
5.     with (conflict_transitions(t) ∩ firable = ∅ ∧ priority(t)
                  ∩enabled = ∅ ∧ conflict_views = ∅)
6.         firable := firable + t
7. endloop
8.
9. do forall t ∈ firable:
10.    exit(ms(t))
11.    exec(eff(t))
12.    enter(mt(t), dst(t)):
13. genCmplevt()
```

Fig. 9. Firing enabled transitions.

```
1. exit(ms) ≡ begin
2. set φ = △ ∩ cover′(ms) :
3. setHistory(ms)
4. loop
5.     do forall s ∈ φ with subs(s) ∩ △ = ∅ :
6.         △ := △ − {s}, φ = φ − {s}
7.         if s ≠ ms then φ = φ + {cntr(s)}
```

Fig. 10. Exiting the main source.

Entering the *main target* results in views and states including the main target and some views or states covered by it to be entered in an outside-in order. The entering procedure is similar to the entering of the top state in the initialization rule. The main difference is that if the main source or some target state is a history pseudostate s, the states recorded by $hc(\mathsf{s})$ needs to be entered rather than s. Details are given in Fig. 12. Given a main target mt and a set TS of target states, the entering procedure starts with mt, which is initially a pending view or state in φ For every $\mathsf{v}, \mathsf{s} \in \varphi$, If v or s is a history, it is removed from φ, a specific state q is added to φ, and all states in $hc(\mathsf{s})$ are added to AT, an auxiliary set initially consisting of all views or states in TS. If v or s is a shallow history, then q is the unique view or state recorded in $hc(\mathsf{s})$. Or if s is a deep history, then q is the view in $hc(\mathsf{s})$ at the highest level of view hierarchy. Furthermore, if s is a view or state, it is entered. Selecting the subviews or substates to enter is similar to the initialization rule shown in Fig. 6.

1. $setHistory(ms) \equiv$
2. **let** $H = \{\mathsf{h} \in \mathsf{P}_h | cntr(\mathsf{h}) \in \triangle \cap cover^{'}(ms)\}$:
3. **do forall** $\mathsf{h} \in \mathsf{P}_{sh} \cap H$:
4. $hc(\mathsf{h}) := subs(cntr(\mathsf{h})) \cap \triangle$
5. **do forall** $\mathsf{h} \in \mathsf{P}_{dh} \cap H$:
6. $hc(\mathsf{h}) := cover(cntr(\mathsf{h})) \cap \triangle$

Fig. 11. History update.

1. $enter(mt, TS) \equiv$ **begin**
2. **set** $\varphi := \{mt\}, AT := TS$:
3. **loop**
4. **do forall** $\mathsf{v} \in \varphi$
5. $\varphi = \varphi - \{\mathsf{v}\}$
6. **if** $\mathsf{v} \in P_{sh}$ **then** $\varphi = \varphi + hc(\mathsf{v}), AT := AT + hc(\mathsf{v})\}$
7. **elseif** $\mathsf{v} \in P_{dh}$ **then**
8. $\varphi := \{\mathsf{q} \in hc(\mathsf{v}) | \forall \mathsf{q}^{'} \in (\mathsf{v}), cover^{'}(\mathsf{q}, \mathsf{q}^{'})\}, AT := AT + hc(\mathsf{v})$
9. **else**
10. $\triangle := \triangle + \{\mathsf{v}\}$,
11. **if** $\mathsf{v} \in \mathsf{V}_{cc}$ **then** $\varphi = \varphi + subs(\mathsf{v})$
12. if $\mathsf{v} \in \mathsf{V}_{sc}$ then $\varphi := \varphi + \{default(\mathsf{v})\}$,.
13. if $\mathsf{v} \in \mathsf{V}_{se}$ then $\varphi := \varphi + subs(\mathsf{v})\}$;.
14. if $\mathsf{v} \in \mathsf{V}_b$ then $\varphi := \varphi + \{default(\mathsf{v})\}$;.

Fig. 12. Entering the main target.

3.2.5. *Completion Events*

In viewcharts a completed event is generated when sum view is completed. A completed view v must satisfy one of the following conditions:

- v is a sequential composite view but it is not a region of a concurrent view and some final substate in basic view of v is active.
- v is a concurrent composite view and all its regions have active final substates in their basic view.
- v is a SEPARATE composite view and all its subview have been completed.

Fig. 13 defines the set of completed Views by a macro **completed**. In addition to completed, two macros for generating and processing completion events are also defined. Firstly, **genCmplevt** generates a completion event by setting $cmplevt$ to true. The second macro **handleCmplevt** processes a pending completion event. It terminates the running viewcharts instance using **stop (this)** if the top view is completed, where this

1. $\mathbf{Completed} \equiv$
   $\{\mathsf{v} \in \mathsf{V}_{sc} | cntr(\mathsf{v}) \notin \mathsf{V}_{cc} \cup \mathsf{V}_{se} \wedge \mathsf{s} \in \mathsf{V}_b, subs(\mathsf{s}) \cap \triangle \cap \mathsf{S}_f \neq \emptyset\}$
   $\{\mathsf{v} \in \mathsf{V}_{cc} | \forall \mathsf{r} \in subs(\mathsf{v}), subs(\mathsf{r}) \cap \triangle \cap \mathsf{S}_f \neq \emptyset\}$
   $\{\mathsf{v} \in \mathsf{V}_{se} | \forall \mathsf{r} \in subs(\mathsf{v}), \mathsf{r} \notin Completed$
2. $\mathbf{genCmplevt} \equiv cmplevt\{:= TRUE\ if\ Completed \neq \emptyset\}$
3. $\mathbf{handleCmplevt} \equiv$
4. **if** $top \in completed$ **then stop(this)**
5. **else** $enabled = \{t \in T | trg(t) = \perp$
   $\wedge src(t) \subseteq \mathbf{Completed} \wedge \mathbf{eval}(grd(t), \triangle)\}$
6. **if** $enabled = \emptyset$ **then** $cmplevt := \mathbf{FALSE}$
7. **else** $\mathbf{firing}(enabled)$

Fig. 13. Completion events.

refers to the instance itself. If the top view is not completed, it first computes the enabled completion transitions with completed sources and then fires them using **firing**. If no such transition exists, **handleCmplevt** resets $cmplevt$.

4. Conclusion

We have presented a method for describing the syntax and semantics of viewcharts. We have picked viewcharts, changed its basis from statecharts to an Extended Finite State Machine (EFSM), and presented as our solution to the overall and general problem discussed in Section 1.2. As described in Section 2.1, the viewcharts formalism fulfils the first, second, and third desirable characteristics of the solution. The precise and sound syntax and semantics of viewcharts presented in Section 3 satisfies the forth characteristic.

Using an independent graph definition language GTDL, not only we have defined the syntax of viewcharts, but also we have described its static semantics. Furthermore, our choice of the formal object mapping automata language has enabled us to describe the operational semantics of viewcharts, covering many important constructs of viewcharts, including hierarchy of views, ownership of elements, scope, and composition of Views.

4.1. *Future Work*

The work presented in this paper has presented a method for describing the syntax and semantics of viewcharts. Along this, a number of directions can be explored to extend and improve this work. Firstly, this method for describing the syntax and semantics of viewcharts makes it suitable for a variety of tool support and, thereby, setting the stage ready for our future work on the last two characteristics of the solution in Section 1.2. Secondly, it my be possible synthesizing viewchart models from scenario-based requirements like

UML 2.0 Sequence Diagram (UML, 2006). The synthesis problem is an interesting and crucial problem in the development of complex object oriented and component based systems. Since sequence diagrams serve to instantiate use cases. If we can synthesize viewcharts from them, we will generate running code directly from UML 2.0 use cases.

References

Beeck, V. (2002). A structured operational semantics for uml-statecharts. *Software and System Modeling*, **1**(2), 130–141.

Borger, E., A. Cavarra and E. Riccobene (2000). Modeling the dynamics of uml state machines. *International Workshop on Abstract State Machines – Theory and Applications*, vol. 1912 of *Lecture Notes in Computer Science*. Springer-Verlag, Monte Verità, Switzerland. pp. 223–241.

Davies, J. (1993). *Specification and Proof in Real-Time CSP*. Distinguished Dissertations in Computer Science. University of Cambridge Press, Cambridge.

Drusinsky, D., and D. Harel (1988). On the power of cooperative concurrency. In *Proceedings of International Conference on Concurrency (CONCURRENCY'88)*, vol. 335 of *Lecture Notes in Computer Science*. Springer-Verlag, New York. pp. 74–103.

Drusinsky, D., and D. Harel (1994). On the power of bounded concurrency I: Finite automata. *Journal of the Association for Computing Machine* (*ACM*), **41**(3), 517–539.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8**, 231–274.

Harel, D., and A. Naamad (1995). *The* STATEMATE *Semantics of Statecharts* (Technical report). 22 Third Avenue, Burlington, Mass.: i-Logix, Inc.

Hoare, C. (1978). Communicating sequential processes. *Communications of ACM*, **8**(21), 666–677.

Isazadeh, A. (1996). *Behavioral Views for Software Requirements Engineering*. PhD thesis, Department of Computing and Information Science, Queen's University, Kingston, Canada.

Isazadeh, A., and D.A. Lamb (1996). An algorithmic semantics for viewcharts. In *Proceedings of IEEE International Conference on Engineering of Complex Computer Systems* (*ICECCS'96*). IEEE Computer Society Press, Montreal, Canada. pp. 293–296.

Isazadeh, A., D.A. Lamb and T. Shepard (1999). Behavioural views for software requirements engineering. *Requirements Engineering Journal*, **4**(1), 19–37.

Janneck, J.W. (1997). *Moses Project*. Computer Engineering and Communications Laboratory, ETH Zurich. `http://www.tik.ee.ethz.ch/moses`

Janneck, J.W. (1998a). *Graph Type Definition Language* (Technical report). Computer Engineering and Communications Laboratory, Swiss Federal Institue of Technology, ETH Zurich.

Janneck, J.W. (1998b). *Object-Based Mapping Automata, Refrence Manual, ver 0.3a* (Technical report). Computer Engineering and Networks Laboratory, Swiss Federal Institue of Technology, ETH Zurich.

Janneck, J.W. (2000). *Syntax and Semantics of Graphs*. PhD thesis. ETH Zurich, Switzerland.

Jin, Y., R. Esser and J.W. Janneck (2004). A method for describing the syntax and semantics of uml statecharts. *Software and Systems Modeling* (*SoSyM*), **3**(2), 150–163.

Kaplan, A., and J.C. Wileden (1995). Formalization and application of a unifying model for name management. In *Proceedings of ACM SIGSOFT'95*. Washington, D.C. pp. 161–172.

Michelle, L., C. Dingel and C. Juergen (2005). *On the Semantics of UML State Machines: Categorization and Comparison* (Technical report). School of Computing, Queen's University, Kingston, Ontario, Canada.

Parnas, D.L. (2000). Requirements documentation: Why a formal basis is essential. In *Fourth IEEE International Conference On Requirements Engineering* (*ICRE2000*). Available online at: `http:web.cps.msu.edu/sens/temp/ICRE2000/parnas.pdf`

Peterson, J.L. (1977). Petri Net. *Computing Surveys*, **9**(3), 223–252.

UML (2006). *Documentation of the Unofied Modeling Language (UML)*. Available online from the Object Management Group (OMG), `http://www.omg.org`

Wolf, A.L., L.A. Clarke and J.C. Wileden (1988). A model of visibility control. *IEEE Transactions on Software Engineering*, **14**(4), 512–520.

A. Isazadeh received the BSc degree in mathematics from Tabriz University (Iran) in 1971, the MSE degree in electrical engineering and computer science from Princeton University (USA) in 1978, and the PhD degree in computing and information science from Queen's University (Canada) in 1996. Before returning to graduate studies in 1992, he worked for several years, as a member of technical staff, at AT&T Bell Laboratories (USA) on telecommunication and manufacturing software systems. Dr. Isazadeh is currently an associate professor in the Department of Computer Science at Tabriz University. His current research interests include information technology, software engineering, mathematical foundation of computer science, and formal methods. He is a member of Mathematical Society of Iran, a member of Computer Society of Iran, and was a senior member of IEEE until 2002.

J. Karimpour received the BSc degree in computer science and applied mathematics from Tabriz University (Iran) in 1998, the MSc degree degree, specializing in the computer systems area of applied mathematics, from Tabriz University in 2000. He is currently a PhD student in the Faculty of Mathematical Sciences at Tabriz University, Iran. His research focusses primarily on the formal specification and compositional verification of component-based systems. In addition, he is working on the formal semantics definition of visual modeling languages such as UML 2 and viewcharts. Previously, Jaber Karimpour has worked on mathematical logic, temporal logic, modal logic, neural networks and numerical methods.

Požiūrių diagramos: sintaksė ir semantika

Ayaz ISAZADEH, Jaber KARIMPOUR

Straipsnyje pateiktas naujas požiūrių diagramų sintaksės ir semantikos aprašymo metodas. Požiūrių diagramos – tai vizualizavimo formalizmas, pritaikytas sistemos komponentų dinaminei elgsenai aprašyti. Požiūrių diagramų sintaksę siūloma apibrėžti atributiniais grafais ir toliau, remiantis tais grafais, aprašyti požiūrių diagramų semantiką. Semantiką siūloma aprašyti objektų atvaizdžių automatu. Šis metodas leidžia aprašyti daugumą požiūrių diagramų konstrukcijų, įskaitant požiūrių hierarchijas, elementų nuosavybę, apibrėžties sritį bei SEPARATE, OR ir AND tipo kompozicijas. Taip pat galima aprašyti užbaigtus perėjimus, perėjimus iš lygmens į lygmenį bei perėjimų požiūrių nepriklausomumo nepažeidžiančių perėjimų istorijas. Tradicinės požiūrių diagramos yra grindžiamos būsenų diagramomis. Šiame straipsnyje bazinis formalizmas išplėstas iki baigtinių būsenų mašinų.