The Semantics of Viewcharts

Ayaz Isazadeh David A. Lamb Glenn H. MacEwen

December 1995
External Technical Report
ISSN-0836-0227-
95-395

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Version 1.1
Document prepared December 21, 1995

Abstract

This paper presents the semantics of Viewcharts. Viewcharts is a formalism designed for specifying the behavioral requirements of large-scale complex systems independent of implementation. The formalism is based on Harel's Statecharts. Therefore, we establish the semantic basis of Viewcharts via translation to Statecharts.

Keywords: Formal Methods, Statecharts, Specification Languages

Contents

List of Figures

1 Introduction

Large-scale software systems, distributed or otherwise, can be complex to describe, construct, manage, understand, and maintain. Current research approaches to reducing this complexity separate software structural and behavioral descriptions [1, 11, 15, 16]. It is, therefore, important to study and analyze the behavioral as well as structural aspects of software systems. Much research has been done on software structures and their patterns, characterizations, and classifications [4, 5, 6]. Special languages, called *configuration languages*, supported by *configuration management systems,*[1] are designed for describing the structure of a software system [12]. Current research on the behavioral aspect of software systems includes using formal notations (e.g., Statecharts [7, 8, 10], ESTEREL [2], Z [3, 18], VDM [14], etc.) for specifying software behaviors and possibly refining the specifications to design and implementation. Large formal specifications can be difficult to create and to understand; more research is needed into methods for assisting software requirements engineers in reducing these difficulties.

Elsewhere, we have introduced the idea of structuring a requirements specification around software behavioral views. We have introduced a notation, called Viewcharts [13], which is based on David Harel's Statecharts. Viewcharts extends Statecharts to include behavioral views and their compositions, limits the scope of broadcast communications and, consequently, reduces the complexity of scale that Statecharts faces in behavioral specifications of large systems. In this paper we provide an overview of the Viewcharts notation and present its semantics.

2 Overview of Viewcharts

This section provides a brief overview of the Viewcharts formalism; a separate paper [13] describes it in more detail. A *(behavioral) view* of a software system is the behavior of the 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. This behavior, of course, may differ from the behavior that the server exhibits to another client. A server, therefore, may have several behavioral views. The caller view of a telephone set and the telephone set's view of a switching system are also examples of behavioral views.

The Viewcharts notation is based on Statecharts [7, 10]. Statecharts, however, has no concept of behavioral views. Viewcharts extends Statecharts to

[1] The term *configuration management systems*, in this context, refers to software interconnection systems which are used in configurable distributed systems to integrate software components using configuration languages; it should not be mistaken for software version control and management systems.

include views and their compositions.

A viewchart consists of a hierarchical composition of *views*. The leaves of the hierarchy, described 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. Views are represented just like states, except that the arc-boxes representing views have thicker borders than those of states.

Note that the statecharts describing the views at the leaves of a viewchart hierarchy are independent. In other words, the scope of broadcast communications of Statecharts is limited to the views and does not cover the entire viewchart. (Section 2.3 discusses extended scopes.)

2.1 Ownership of Elements

The Viewcharts notation limits the scope of broadcast communications. In other words, the scope of an element (event, action, or variable) in a given view is limited to the view. 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. 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". (Section 2.2 discusses the triggering concept.) 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 event may have multiple owners; in other words an event can be triggered by more than one view. An event may also have no owner, in which case the event can never occur. The Viewcharts notation allows this case, because of the possibility of further composition of the viewchart with additional views, which may affect the event. This is exactly analogous to the notion of free variables in program fragments, which can be be bound in a larger context.

The notion of ownership for events is a natural consequence of composing views, while the scopes of events are limited. This notion also applies for actions. However, actions are implicitly declared: an action belongs to the view (or views) that generates (or generate) the action. There is no need, therefore, for explicit declaration of actions. However, an action may also be owned as an event by some other views, in which case it must be declared accordingly.

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. If a variable x is

declared by a view V and redeclared by another view V1 within the scope of x, then Viewcharts recognizes two different variables which can be referenced by their *qualified names*, V.x and V1.x. In a view that is covered by the scopes of both variables, the *base name* x refers to V1.x. In the case of events, on the other hand, there is no need to specify them by their qualified names; Viewcharts determines the effect of each event occurrence based on the ownership and scoping rules. However, an event occurrence may still be specified by its qualified name, if it does not violate these rules.

Consequently, unlike events and actions, variables cannot have multiple owners. On the other hand, if a variable is used in a view, but not declared by the view or any of its superviews (i.e., it has no owner), then the variable, by default, belongs to the top view of the corresponding viewchart.

Syntactically, elements owned by a view can be declared by listing them following the name of the view either in the viewchart, as in Figure 2, or out of it as a separate text. In referencing a view by its name, however, it should be noted that the view must be uniquely identified. 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.

In a viewchart, if the triggering view of an element is obvious and there is no ambiguity in the ownership of the element, then there is no need for explicit declaration of the element.

2.2 Ownership and Triggering

In a statechart, which describes the behavior of a system, the events generated by the system are called *internal* events; all other events (generated by the environment) are *external*. This is also true for Viewcharts. Since a view is considered as a stand-alone system, an event that affects a view is either internal or external to the view. An event may not affect a view at all, being neither internal nor external; the view is independent of such an event.

An internal event of a view (i.e., an event generated by the view) is owned by the view. An event can be internal with respect to more than one view (i.e., it can be generated by more than one view); therefore, it can be owned by more than one view.

An external event of a view (i.e., an event, generated by the environment or other views, that affects the view) may or may not be owned by the view:

- If the event is generated by other views, then it is not owned by the view; it is, in fact, an internal event of the views that generate it and, therefore, owned by them.

- If the event is generated by the environment and affects the view through

a composition and a consequent extended scope, then it is not owned by the view. (Section 2.3 discusses compositions and extended scopes.)

- If the event is generated by the environment and affects the view directly, independent of any composition, then it is owned by the view.

An event, generated by the environment, can have direct affect on more than one view; therefore, it can be owned by more than one view.

In a statechart, when an event occurs, it is sensed throughout the statechart and, therefore, all occurrences of the event within the statechart are affected. In a viewchart, however, when an event occurs it is sensed only in some views and, therefore, only some occurrences of the event are affected. To determine the scope of the event (i.e., to determine which occurrences of the event are affected) we must know which view *triggers* the event. The scope of the event then is the view that triggers the event (and possibly some other views as described in Section 2.3).

In Viewcharts, therefore, an event does not just occur, it is triggered by a view. The view that triggers the event must be clearly specified if it is not obvious. For example, we may specify that "$\mathsf{V}.e$ occurs", where V is the name of the view that triggers the event e; to state that "e occurs" is ambiguous.

2.3 Composing Behavioral Views

Views can be composed in four ways: SEPARATE, OR, AND, and HIERARCHICAL compositions. Except for the affect of ownership and scoping restrictions, the OR, AND, and HIERARCHICAL compositions of views, in Viewcharts, are similar to the OR, AND, and HIERARCHICAL compositions of states, in Statecharts, respectively. The SEPARATE composition of views, however, is specific to Viewcharts.

2.3.1 SEPARATE Composition of Views

In a SEPARATE composition of views, all the views are active; (A view is *active* whenever the system is in a state of the view.) 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. No view is, in fact, aware of any other view in the composition. Visually, the views involved in a SEPARATE composition are drawn on the top of each other, as shown in Figure 1, giving the impression that they are located on different planes and, consequently, are hidden from each other.

The representation (a), in this figure, specifies a SEPARATE composition of the view V with a finite number of other views; (b) specifies a SEPARATE composition of V, U, and W; and (c) specifies a SEPARATE composition of five views $\mathsf{V}_4, \ldots, \mathsf{V}_9$. In all these cases, the behavior of the first view, the

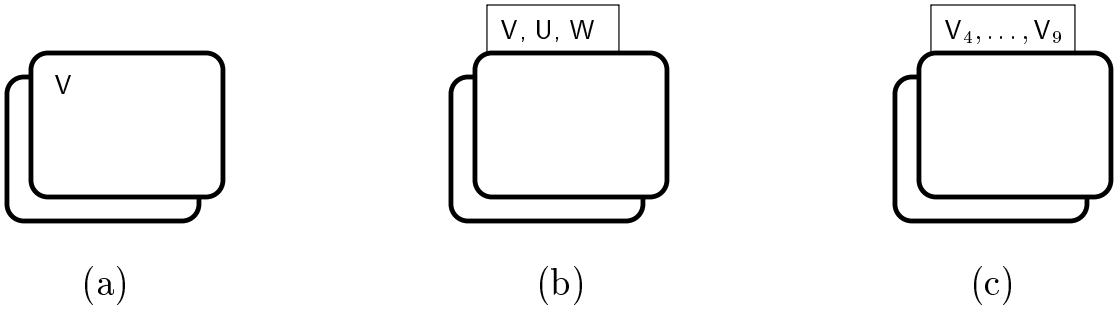

Figure 1: Visual representation of SEPARATE compositions.

one located on the top, can be specified. By default all the other views are identical to the first one. Exceptions are represented by specifying the others separately and referencing them in the composition by their names using the representations (b) or (c). If only a small number of views are involved in the composition, then it may be practical to give them enough space to show their behaviors. An example of this representation is given in Figure 2, which includes a SEPARATE composition of views V5 and V6.

2.3.2 OR Composition of Views

The OR and SEPARATE compositions are similar, except that in an OR composition, only one view can be active and there can be transitions between the views. Like the SEPARATE composition, any subview or state in one view is hidden from (i.e., cannot be referenced by) the other views. In Figure 2, for example, the view V consists of an OR composition of V1 and V2.

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.

2.3.3 AND Composition of Views

In an AND 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; variables, however, must be referenced by their qualified names. The view V7 of Figure 2, for example, is ANDed with a SEPARATE composition of V5 and V6.

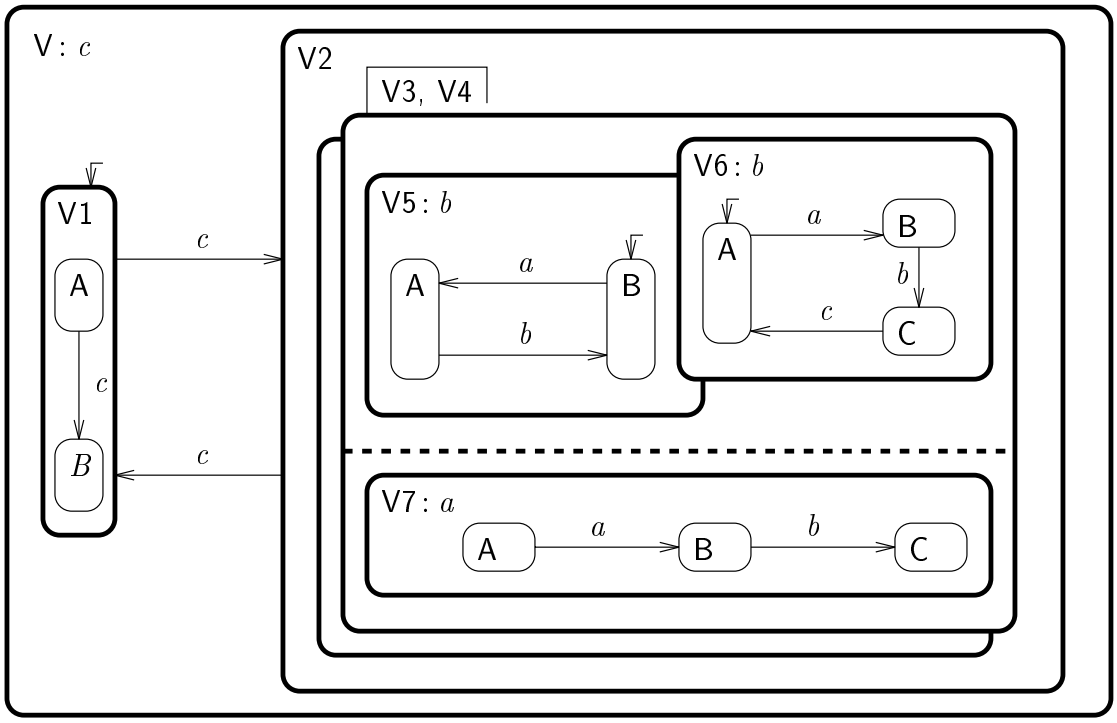

Figure 2: Composition of views in a viewchart.

2.3.4 HIERARCHICAL Composition of Views

In a HIERARCHICAL composition of views, some views form a superview; all the subviews and states in a superview are visible to the superview; and the scopes of the elements owned by a superview covers all its subviews.

The viewchart of Figure 2, for example, is composed of a SEPARATE composition of V5 and V6, which in turn is ANDed with V7 forming V3. A SEPARATE composition of two identical views V3 and V4 forms V2. The full view V is an OR composition of V1 and V2.

2.4 Effect on Transitions

The following examples demonstrate the way in which the compositions affect transitions with the same label. A possible configuration of the system described in Figure 2 is {V3.V5.A, V3.V6.B, V3.V7.B, V4.V5.B, V4.V6.C, V4.V7.A}.

Recall (Section 2.1) that a view can trigger the events it owns. Assuming that the system is in sub-configuration {V3.V5.B, V3.V6.A, V3.V7.A},

- if the view V3.V7 triggers a, then the sub-configuration will change to {V3.V5.A, V3.V6.B, V3.V7.B};

- if the view V triggers c, nondeterministically, then the entire system configuration will change to either {V1.A} or {V1.B};
- no other event can change the sub-configuration.

Assuming that the system is in sub-configuration
{V3.V5.A, V3.V6.B, V3.V7.B},

- if the view V3.V5 triggers b, then the sub-configuration will change to {V3.V5.B, V3.V6.B, V3.V7.C};
- if the view V3.V6 triggers b, then the sub-configuration will change to {V3.V5.A, V3.V6.C, V3.V7.C};
- if the view V triggers c, nondeterministically, then the entire system configuration will change to either {V1.A} or {V1.B};
- no other event can change the sub-configuration.

Assuming that the system is in sub-configuration {V3.V6.C},

- if the view V triggers c, nondeterministically, then the sub-configuration will change to {V3.V6.A} or the entire system configuration will change to either {V1.A} or {V1.B};
- no other event can change the sub-configuration.

2.5 History Transitions

Considering that the leaves of a viewchart hierarchy are stand-alone statecharts, history (H) as well as deep history (H*) transitions can occur within the leaves; and Viewcharts handles them in exactly the same way as Statecharts does. Viewcharts also allows these transitions to occur in the higher level views of a viewchart hierarchy as long as they do not cross view boundaries. A history transition that crosses view boundaries does not affect the semantics of Viewcharts. In other words, a viewchart that contains such transitions is still legal and can be translated to a legal statechart (See Section 3). However, allowing the transitions to cross view boundaries violates the independence of views.

2.6 Timing Issues

Viewcharts adopts Harel's *synchrony* hypothesis that events are instantaneous. Specifically, events, actions, and checking the value of a condition expression ideally take no time; therefore, transitions are also instantaneous. A time consuming task is considered an *activity* and is performed in a state. However,

a point in time when an activity starts or ends can be marked by the occurrence of an event. For example, we may define **retrieved**($db.prec(pid)$) as an event that occurs at a point in time when the activity of retrieving a record from a database is completed.

Statecharts allows specification of concurrent events; $a \wedge b$, for example, can be considered as an event that occurs when the two events a and b occur simultaneously. Harel, however, describes that while a and b occur at one *step*, they occur at different *micro-steps* within the step [9]. Therefore, there is an order of occurrence between them. Another approach to specifying concurrent events is based on the *single-event* hypothesis, where the events occur in a nondeterministic order at consecutive steps. Viewcharts allows either approach provided that it is supported by the semantics chosen for the underlying Statecharts.

Viewcharts also allows the timeout and scheduled transitions of Statecharts.

3 Semantics

We now establish a semantic basis for Viewcharts via translation to Statecharts. Recall that the leaves of a viewchart hierarchy are independent statecharts. The Viewcharts formalism, therefore, can be viewed as a high-level notation that uses (but does not change) the Statecharts notation. Statecharts, however, has a variety of different semantics, each of which makes certain assumptions or imposes certain restrictions on the notation, resulting in some variations in Statecharts [9, 17]. The Viewcharts notation encapsulates these variations (within the leaves of Viewcharts hierarchies) and, therefore, is not restricted to a particular variation of Statecharts. The encapsulation, furthermore, allows Viewcharts to benefit from different variations of Statecharts and their semantics, available tools, and further extensions and evolutions.

To provide a semantic basis for Viewcharts, therefore, we need to show:

> *Given a viewchart, there is a statechart that describes the same behavior as the viewchart does.*

An examples of such translations is shown in Figure 3, which is the Statecharts translation of the viewchart shown in Figure 2.

Views, in Viewcharts, are similar to states; and states, in Statecharts, are uniquely identified by their full paths. Therefore, transforming views to states and SEPARATE composition of views to AND composition of states, in a viewchart, transforms the viewchart to a statechart. The resulting statechart, however, may not preserve the behavior described by the corresponding viewchart (because the viewchart limits the scopes of its elements, while the statechart does not.) To ensure that the transformation does not change the

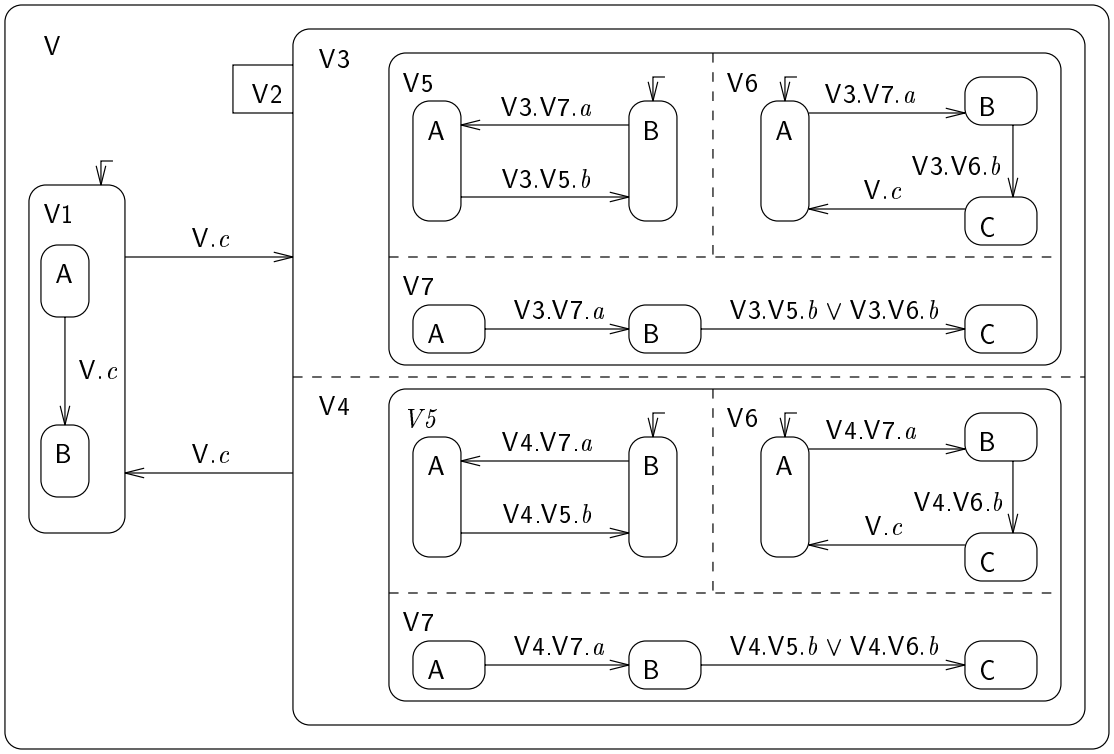

Figure 3: A Statecharts translation of the viewchart shown in Figure 2.

behavior description, the elements must be renamed, before making the transformation, such that a given element does not occur beyond its scope. The translation, therefore, can be summarized as follows:

- Rename the elements that occur in each view, such that they can be uniquely identified within the viewchart, while the behavior described by the viewchart is preserved.
- Transform the SEPARATE compositions of views, which do not exist in statecharts, to AND composition of states.
- Transform views to states.

In general, the resulting statechart is more complex than the original viewchart.

Formally, a given viewchart can be translated to its equivalent statechart as follows:

Let w denote the given viewchart;
U denote the set of all the views in w, where a view is

uniquely identified by a fully or partially qualified name;
V denote the set of all the variables in w;
E denote the set of all the events and actions in w;
$\mathrm{sup}(x, y)$ denote "x is a direct or indirect superview of y", where $x \in U \wedge y \in U$;
$\mathrm{and}(x, y)$ denote "x is ANDed with y", where $x \in U \wedge y \in U$;
$\mathrm{own}(x, y)$ denote "x owns y", where $x \in U \wedge y \in E \cup V$;
$\mathrm{bas}(x)$ denote "x is a base name", where $x \in E \cup V$;
$\mathrm{occ}(x, y)$ denote "x occurs in y", where $x \in E \cup V \wedge y \in U$;

$S'_u = \{x | \mathrm{sup}(x, u)\}$,
the set of all superviews of u;

$S_u = \{u\} \cup S'_u$

$A_u = \{x | \exists y \in S_u \cdot \mathrm{and}(x, y)\}$,
views that are ANDed with u or with any superview of u;

$O_e = \{x | \mathrm{own}(x, e) \wedge e \in E\}$,
views that own e;

$T_{e,u} = O_e \cap (S_u \cup A_u)$,
views that can trigger e in u;

$O''_v = \{x | \mathrm{own}(x, v) \wedge v \in V\}$,
views that declare v;

$O'_{v,u} = O''_v \cap S_u$

$O_{v,u} = \{x | x \in O'_{v,u} \wedge \forall y \in O'_{v,u} \setminus \{x\} \cdot \mathrm{sup}(y, x)\}$,
this set is either empty or has only one element, namely the owner of v that occurs in u;

$E_u = \{x | \mathrm{occ}(x, u) \wedge \mathrm{bas}(x) \wedge x \in E\}$,
not yet prefixed events and actions that occur in u;

$V_u = \{x | \mathrm{occ}(x, u) \wedge \mathrm{bas}(x) \wedge x \in V\}$,
not yet prefixed variables that occur in u.

Figure 4 then presents a simple algorithm for the translation. The first loop, line 1–5, makes sure that each occurrence of all the state and view names in the viewchart is specified by a uniquely identifiable partial or full name. The state and view names can occur in a condition expression; also recall (Section 2.1) that an element can be specified by a fully or partially qualified name. A qualified name for an element is of the form *viewname.elementname*, where *viewname* uniquely identifies a view that owns the element. A partially

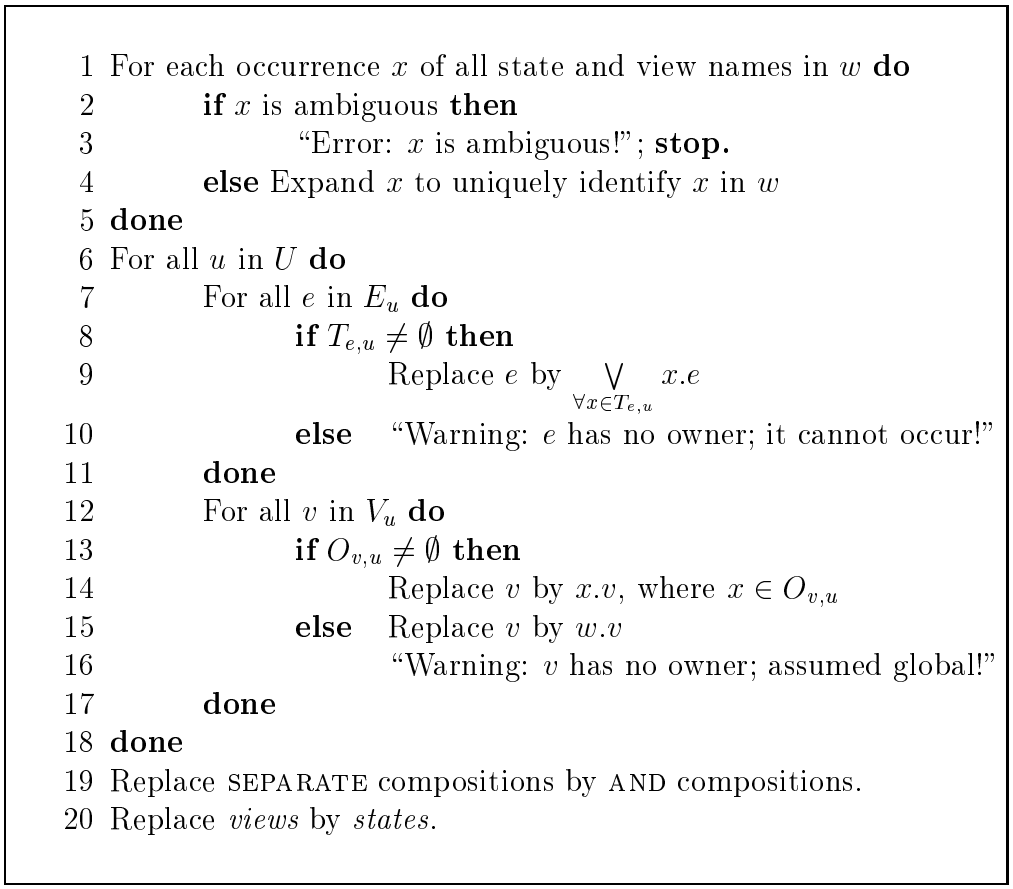

Figure 4: An algorithm translating a viewchart to an equivalent statechart.

qualified name may uniquely identify a view in a viewchart (considering the scoping restrictions); however, it may no longer be unique in the global environment of an equivalent statechart. This loop, therefore, makes sure that if a view or state is specified by a partially qualified name, it is not ambiguous; it also expands the name to make it unique within the entire viewchart.

The second loop, line 6–18, renames the events, actions, and variables. Line 7–11 handles the events and actions. An event, in a viewchart, can have multiple owners; i.e., it can be triggered by multiple views. However, when a view that owns an event triggers the event only some (not necessarily all) occurrences of the event, in accordance with the ownership and scoping rules, take place. Furthermore, more than one view may trigger an event with the same effect on some occurrences of the event. We want to distinguish different occurrences of an event that can take place independent of each other and give them different and unique names. We also want this renaming not to change the behavior specified by the viewchart; i.e., a renamed occurrence of an event to be triggered if and only if the original occurrence of the event is triggered. This is accomplished by line 9 (as discussed in Section 3.1). Notice that x

in line 9 is a uniquely identifiable partially or fully qualified name of a view that owns e and e is a base name. The events that are specified by a qualified name are checked and possibly expanded by the first loop (line 1–5) and left unchanged.

Line 12–17 renames the variables. Unlike events, variables cannot have multiple owners. However, like events, different occurrences of a variable can also be independent of each other. Recall (Section 2.1) that if a variable is declared by a view and redeclared by a subview of the view then the two declarations define two independent variables. Therefore, the owner of a variable v that occur in a view u is the first view in the bottom-up hierarchical sequence of u and its superviews that declares the variable. This is represented by the set $O_{v,u}$, which is either empty or has only one element, the owner. As in the case of events and actions, x in line 14 is a uniquely identifiable partially or fully qualified name of a view that owns v and v is a base name. The variables that are specified by a qualified name are checked and possibly expanded by the first loop (line 1–5) and left unchanged. The algorithm now should be self explanatory.

3.1 Proof of the Correctness

To prove that the algorithm is correct, we have to show that

(a) the translation results in a statechart,

(b) the resulting statechart describes the same behavior as the original viewchart, and

(c) the algorithm terminates.

For part (a), notice that the algorithm replaces the SEPARATE compositions by AND compositions (line 19) and the views by states (line 20); as all other elements of Viewcharts are identical to those in Statecharts, the result is a statechart.

For part (b), an analysis of line 9 shows that the algorithm does preserve the behavior described by the viewchart; the rest of the algorithm is obvious in regard to preserving the behavior. Consider the sets S_u, A_u, and O_e, as defined above; based on the ownership and scoping rules (described in Sections 2.1 and 2.3), then, a subset of O_e, $T_{e,u}$, contains the views that can trigger e with the same affect on all occurrences of e in the view u. Therefore, we replace all occurrences of e in u by $\bigvee_{\forall x \in T_{e,u}} x.e$. Consequently, the transitions labeled e in the view u take place if and only if the corresponding transitions in the resulting statechart take place.

Notice that the translation does not affect transitions (including interrupt and history transitions). The transition labels may change; however, the transitions are preserved as they are.

For part (c), notice that all the sets and loops in the algorithm are finite; therefore, the algorithm terminates with w transformed to an equivalent statechart.

4 Conclusion and Future Work

We have provided a semantic basis for the Viewcharts formalism via translation to Statecharts. Our semantics is based on the semantics of the underlying Statecharts notation. This allows allows Viewcharts to benefit from different variations of Statecharts and their semantics, available tools, and further extensions and evolutions.

The following is a list of further extensions that would make Viewcharts richer and more expressive:

- *Transition between* SEPARATE *views*: A SEPARATE composition of views, in a viewchart, is transformed to an AND composition of states, in the equivalent statechart, by the algorithm described in Section 3. Since Statecharts does not allow transitions between ANDed states; therefore, no transition is allowed between the views in a SEPARATE composition. However, it would be interesting to extend the semantic basis of Viewcharts to allow such transitions.

- *Exporting and importing elements*: In a HIERARCHICAL composition of views, the scope of an element owned by a view covers the view and all its subviews. There are, however, other alternative which should be explored. It may, for example, better encapsulate the views, in this composition, to limit the scope of an elements, to the view that owns the element, but instead introduce the notion of *export* and *import*. A view then can export an element to its immediate superview and thereby extend the scope of the element to the exported view. Similarly, a view may import an element from its immediate superview.

- *General history transitions*: Further research is required to provide more general history transitions without violating the independence of views. (See Section 3.)

- *Modeling*: Finally, modeling viewcharts is another topic of future research. The algorithm provided in Section 3 translates a given viewchart to its equivalent statechart. It is possible to produce an executable model of a viewchart using this algorithm and an available Statecharts

tool (e.g., STATEMATE). However, it would be more efficient and practical to provide a method of producing the executable models directly from the Viewcharts notation.

References

[1] M. R. Barbacci, C. B. Weinstock, D. L. Doubleday, M. J. Gardner, and R. W. Lichota. Durra: A structure description language for developing distributed applications. *IEE Software Engineering Journal*, 8(2):83–94, Mar 1993.

[2] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448, New York, June 1984. Springer-Verlag.

[3] B. P. Collins, J. E. Nicholls, and I. H. Sorensen. Introducing formal methods: The CICS experience with Z. Technical Report TR12.260, IBM Hursley Park, December 1987.

[4] T. R. Dean. *Characterizing Software Structure Using Connectivity*. PhD thesis, Queen's University, Department of Computing and Information Science, 1993.

[5] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2):114–121, June 1976.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1994.

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

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

[9] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. Technical report, i-Logix, Inc., 22 Third Avenue, Burlington, Mass. 01803, USA, November 1995.

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

[11] C. Hofmeister, E. White, and J. Purtilo. Surgeon: A packager for dynamically reconfigurable distributed applications. *IEE Software Engineering Journal*, 8(2):95–101, March 1993.

[12] A. Isazadeh. Configuration languages for distributed software systems. Ph.D. Depth Paper, Queen's University, Department of Computing and Information Science, August 1994. Also to appear in IEEE Transactions on Software Engineering Journal.

[13] A. Isazadeh, D. A. Lamb, and G. H. MacEwen. Viewcharts: A behavioral specification language for complex systems. External Technical Report ISSN-0836-0227-95-388, Queen's University, Department of Computing and Information Science, October 1995.

[14] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[15] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 4–14, Pittsburgh, Pennsylvania, March 1994.

[16] J. Nehmer, D. Haban, F. Mattern, D. Wybraniertz, and D. Rombach. Key concepts of the INCAS multicomputer project. *IEEE Transactions on Software Engineering*, SE1-13(8):913–923, August 1987.

[17] M. von der Beek. A comparison of statechart variants. In *Formal Techniques in Real-Time and Fault-Tolarent Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, New York, 1994. Springer-Verlag.

[18] J. B. Wordsworth. *Software Development with Z.* International Computer Science Series. Addison-Wesley, 1992.