Viewcharts: A Behavioral Specification Language for Complex Systems

Ayaz Isazadeh David A. Lamb Glenn H. MacEwen

October 31, 1995
External Technical Report
ISSN-0836-0227-
95-388

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

Abstract

This paper introduces a formalism, called Viewcharts, for specification and composition of software behavioral views. The objective is software behavioral requirements specification independent of implementation. The paper claims that behavioral requirements of large-scale and complex systems can be described formally as compositions of simple behavioral views. The Viewcharts formalism is presented to demonstrate the behavioral views and support the claim.

Keywords: Formal Methods, Statecharts, Specification Languages

Contents

List of Figures

1 Introduction

Large-scale real-time distributed systems, can be complex to describe, construct, manage, understand, and maintain. Current research approaches to reducing this complexity separate software structural and behavioral descriptions [2, 11, 14, 15]. It is, therefore, important to identify and formally describe software *behavioral views*[1] and structural patterns; these patterns and/or views, then, can be used in software systems analysis, specifications, design, and configurations. Much research has been done on software structures and their patterns, characterizations, and classifications. Little or no attention, however, has been given to the identification and analysis of software behavioral views that can be used as a basis for software behavioral requirements engineering.

This paper introduces the concept of software behavioral views, and presents a formal notation for their specification and composition. The objective is software behavioral requirements specification independent of design and implementation. The paper claims that software behavioral requirements can be described formally in terms of behavioral views. To establish this claim, the paper introduces a notation, called Viewcharts, which is based on David Harel's Statecharts [7]. The Viewcharts notation 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.

1.1 Application Domain

The Viewcharts formalism can be viewed as a high-level specification language that uses Statecharts; and Statecharts is designed for real-time event-driven reactive systems. Furthermore, Viewcharts extends Statecharts to include behavioral views and their composition; consequently, it describes behavioral requirements of large-scale complex systems as a composition of views. The domain of the Viewcharts formalism, therefore, is behavioral specification of large-scale complex real-time event-driven reactive systems.

1.2 Previous Work

There is a body of work on state-transition based formal methods and software behavioral requirements specifications using these methods. This work includes Statecharts of Harel [7, 9], Modechart of Jahanian and Mok [12], Communicating Real-Time State Machines (CRSM's) of Shaw [19], Augmented

[1] In the literature, the term *behavioral view* (in contrast to structural or functional views) generally refers to the behavioral aspect of a system. In this paper, however, we talk about different behavioral views of the same system. See Section 2 for our definition of the term.

State Transition Diagram of Hendricksen (ASTD) [10], and Requirements State Machine Language (RSML) of Leveson [13]. In object-oriented analysis and design methods also Rumbaugh [17], Selic [18], Walters [20], and others [3] have used state-transition based methods to describe software behavior.

To our knowledge, however, no one has ever attempted to formally specify software behavioral requirements in terms of behavioral views.

2 Viewcharts

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.[2] Statecharts, however, has no concept of behavioral views. Viewcharts extends Statecharts to include views and their compositions.

Statecharts is often criticized for not being practical for large-scale systems specifications. Specifically, the issue of scale in Statecharts leads to the following problems:

1. *Explosion of States*: In conventional finite state machines the number of states grows exponentially as the scale of the system grows linearly. Statecharts reduces this blow-up, but for a large-scale system this is still a problem.

2. *Global Name Space*: All names, in Statecharts, are defined globally. A given event, for example, can be sensed throughout the system and, therefore, it must have a unique name. Managing the name space is difficult in large-scale systems.

Viewcharts attempts to eliminate these problems.

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.

[2]The reader is assumed to be familiar with the Statecharts formalism; relevant references are given in Section 1.2.

Notice that the statecharts describing views, in a viewchart, are independent. In other words, the scope of broadcast communications of Statecharts is limited to the views and does not cover the entire viewchart. (See Section 2.2 for extended scopes.)

Notice also that a statechart describing a view, in a viewchart, describes only a behavioral view of the system or, more importantly, its component. In other words, the number of states and the size of such a statechart are affected only by the scale of the behavioral view. Consequently, considering that a large-scale system behavior can be described in terms of simple behavioral views, Viewcharts is expected to eliminate the issue of scale.

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 composing 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 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 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 *full names*, $\mathsf{V}.x$ and $\mathsf{V1}.x$. In a view that is covered by the scopes of both variables, the *base name* x refers to $\mathsf{V1}.x$. In the case of events, on the other hand, there is no need to specify them by there full names; Viewcharts determines the affect of each event occurrence based on the ownership and scoping rules. However, an event occurrence may still be specified by its full name, provided that 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 *full name*, which is a full path from the root to the view, consisting 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 Composing Behavioral Views

Views can be composed in four ways: SEPARATE, OR, AND, and HIERARCHICAL compositions.

2.2.1 SEPARATE Composition of Views

In a SEPARATE composition of views, all the views are active;[3] 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 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

[3]A view is *active* if, and for a period of time during which, the system is in a state of the view.

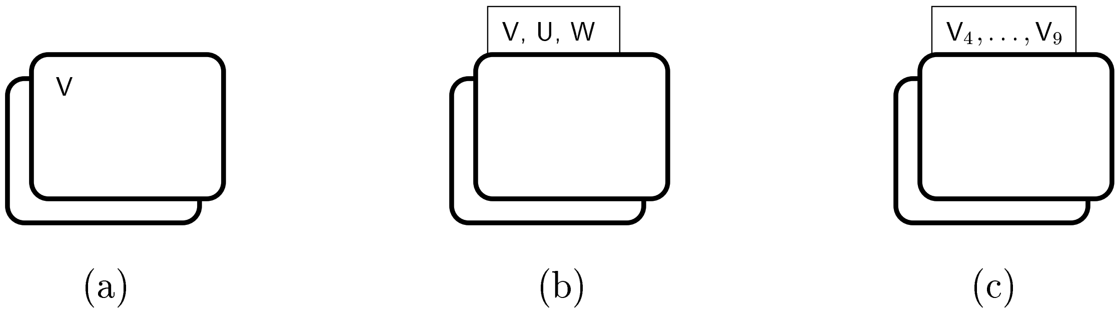

Figure 1: Visual representation of SEPARATE compositions.

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.

2.2.2 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. In Figure 2, for example, the view V consists of an OR composition of V1 and V2.

2.2.3 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 full names. The view V7 of Figure 2, for example, is ANDed with a SEPARATE composition of V5 and V6.

2.2.4 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.

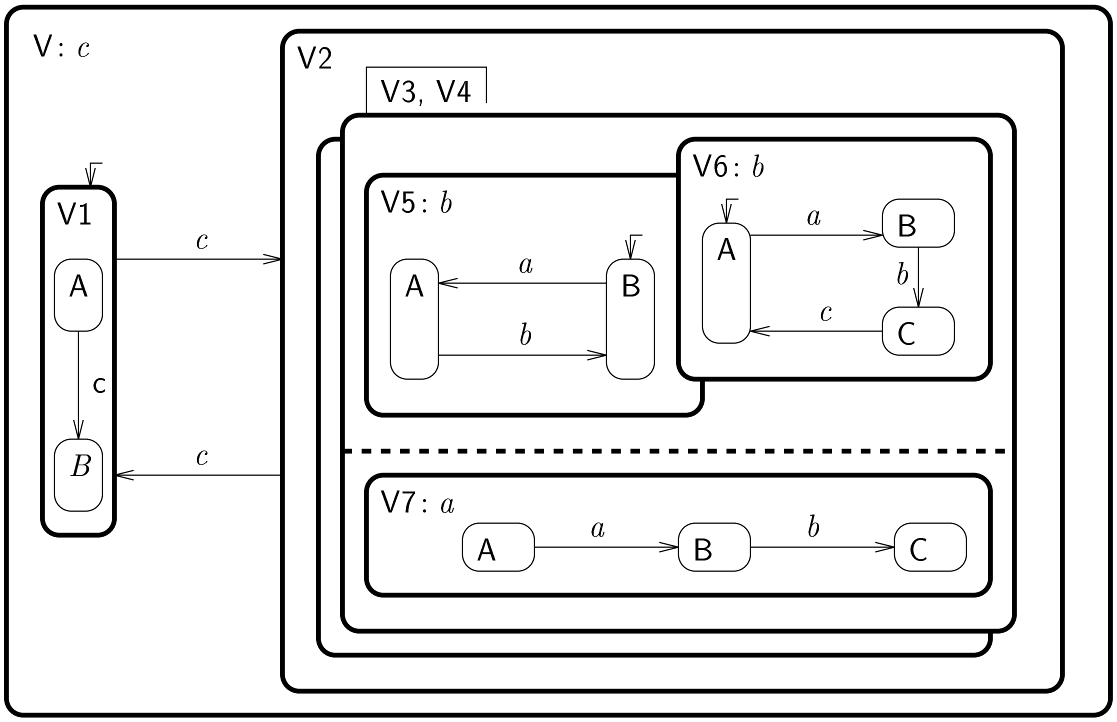

Figure 2: A HIERARCHICAL composition of views.

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

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

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

3 Example: Manufacturing Control System

This section presents a Viewcharts specification of a Manufacturing Control System (MCS), demonstrating the way in which the behavioral requirements of a software system can be specified as a composition of its behavioral views. An informal, but detailed, description of a similar system is given by Dunietz and others [4].

Notice that a system can be specified at different levels of abstractions. We need 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. Therefore, we will keep the specifications at an appropriate level of abstraction.

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

[4]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.

[5]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.

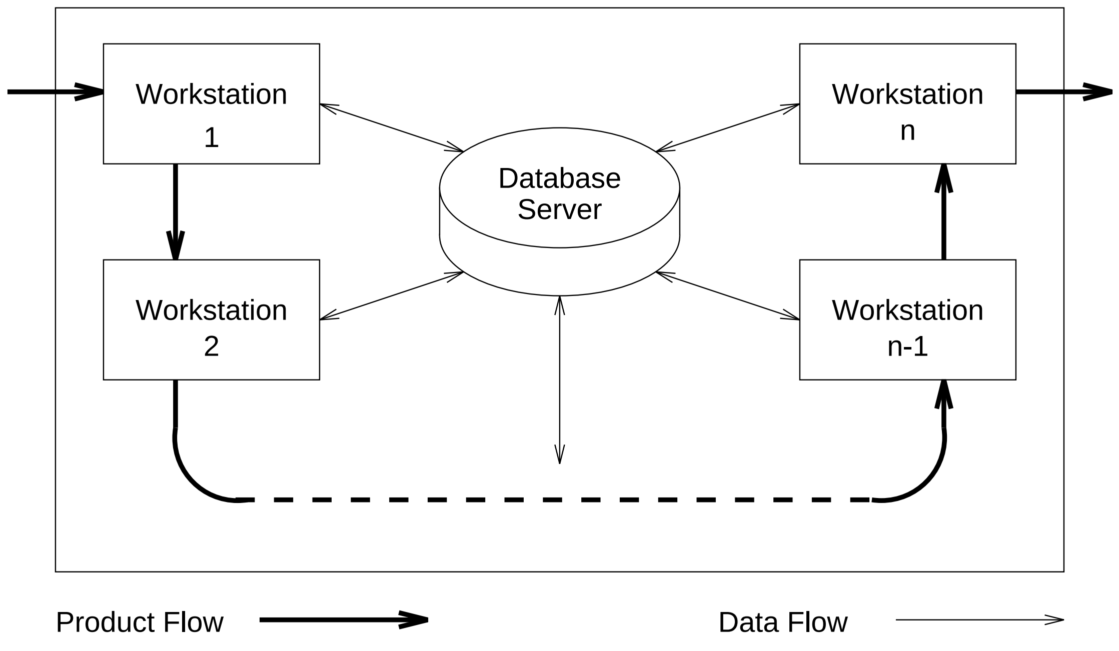

Figure 3: Product and information flow in a manufacturing line.

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 workstations, 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 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; therefore, 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.

3.1 Specifying Behavioral Views

The viewchart WS_1, shown in Figure 4, describes the behavior of the system observable at the Serialization Workstation, which is the first workstation in the manufacturing line. It consists of two ANDed views: WS, which describes the behavior of the workstation, and DBS, which describes the workstation's view of DBS.

The declaration "$\mathsf{WS}_1 : prec$", in this figure, shows that *prec* is owned by WS_1 and, therefore, its scope is WS_1. *db* is not declared anywhere in WS_1 and, therefore, it is global to WS_1. All other elements in WS_1 are implicitly declared; they belong either to WS or DBS and their scope is WS_1. the event **ad**(*db.prec*), for example, 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 WS_1.

WS specifies that the workstation by default is in the state of SERIALIZING, 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* which, 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.

Other workstations, at the abstraction level of this specification, have identical behaviors. Therefore, a SEPARATE composition of $n-1$ behavioral views, as shown in Figure 5, can specify the behavior of the system observable at these workstations. Each views, of course, can be further refined to describe the specific and detailed behavior at the corresponding workstation.

The declaration "$\mathsf{WS}_2, \ldots, \mathsf{WS}_n : prec, pid$", in this figure, shows that for each view WS_i ($i = 2, \ldots, n$), the elements *prec* and *pid* belong to WS_i and, therefore, their scope is WS_i. *db* is not declared anywhere in WS_i and, there-

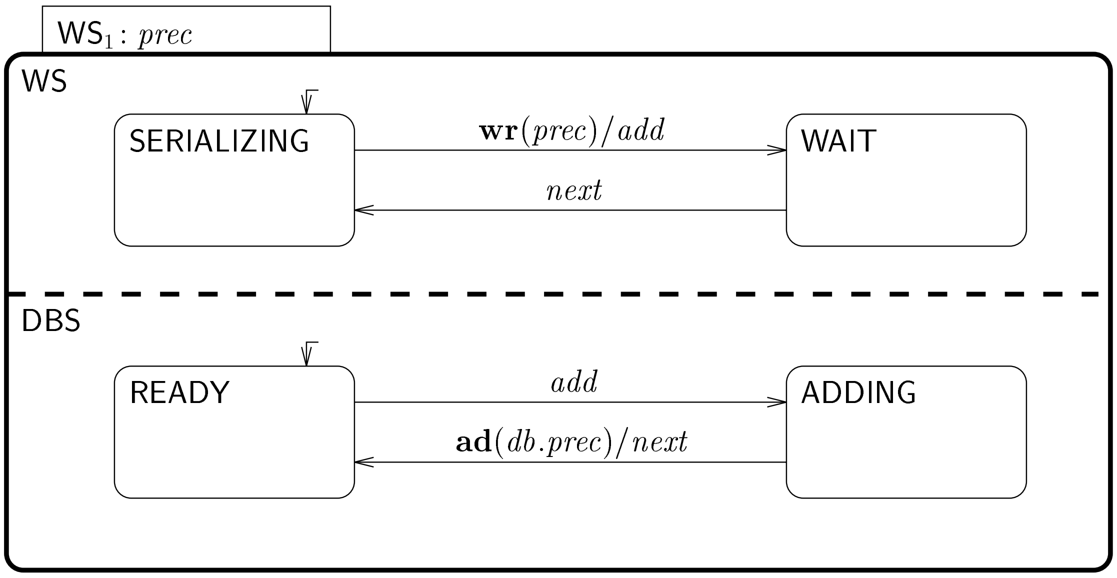

Figure 4: A viewchart for the Serialization Workstation.

fore, it is global to WS_i. All other elements in WS_i are implicitly declared; they belong either to WS_i.WS or WS_i.DBS and their scope is WS_i.

The event $\mathbf{rd}(pid)$, which is an abbreviation for $\mathbf{read}(pid)$, 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[6] 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 are included in the product record and its affect on the system behavior can be specified by refining the PROCESSING and other affected states.

Figure 5 should now be self explanatory.

[6]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.

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

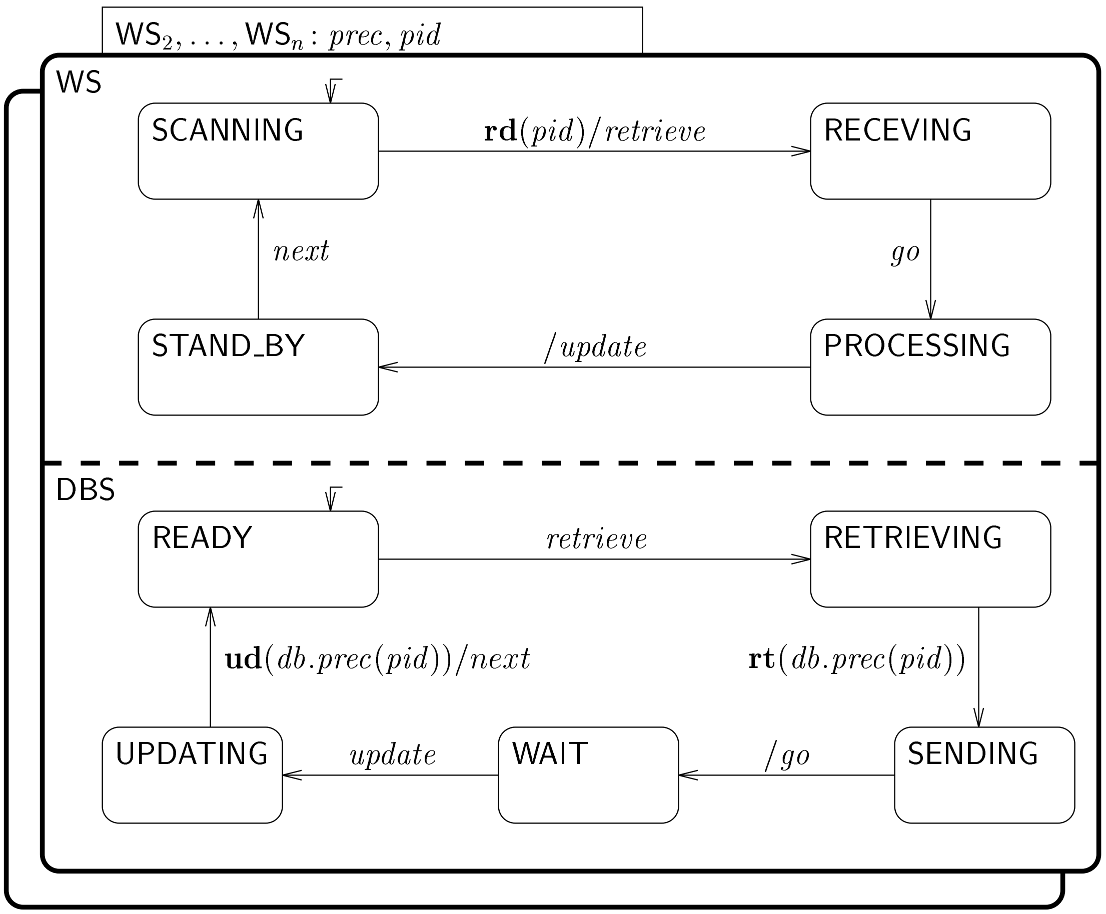

Figure 5: The workstations' views of the system.

3.2 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 6 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 "WS$_1$: *prec*" of Figure 4, "WS$_2, \ldots,$ WS$_n$: *prec, pid*" of Figure 5, and "MCS : *db*" of Figure 6. These declarations mean that each view WS$_i$ ($i = 2, \ldots, n$) has its own variables *pid* and *prec*, WS$_1$ has its own variable *prec*, and *db* is global to all WS$_i$ ($i = 1, \ldots, n$). All the views, therefore, 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.

Notice that a Statecharts specification of this example would consist of $n + 1$ orthogonal components: one for each workstation and another one for

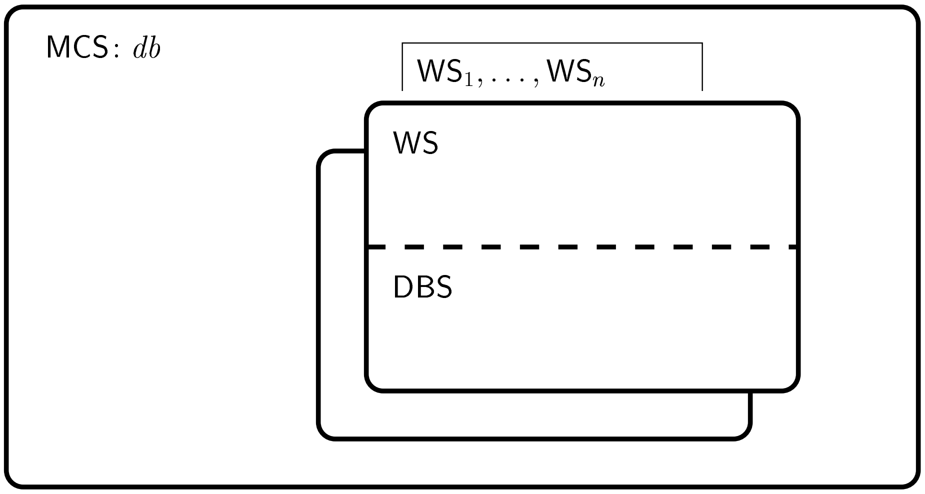

Figure 6: A viewchart for a manufacturing line.

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. 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 define different variables for the product IDs being scanned (concurrently) by different workstations. Similarly, we have to uniquely identify the associated events and actions. Finally, we have to specify the way in which DBS is supposed to handle all these events, actions, and variables. Furthermore, suppose that an action to update a product record is generated by a workstation, while DBS is in a state of retrieving another one. We cannot ignore the action; should we queue it, or specify a method for concurrent processing of database transaction requests? In any case, we are forced to make some design decisions; and doing so not only limits design choices, but also unnecessarily complicates the specification.

4 Conclusions

A large-scale software system may exhibit a combination of many different and identical behavioral views. The Viewcharts notation allows these views to be specified as stand-alone systems and provides a method of composing them to form the overall system behavior specification. It is important, however, to realize that composing behavioral views is different from integrating them. In a composition of views, the views keep their identities and are used as building blocks of the requirements specification; each requirement can be traced back to its originating view. In an integration of views, on the other hand, the views may loose their identities and be replaced by a different mechanism; the requirements cannot be traced back to their originating views. Integration requires making design decisions, while composition does not. Viewcharts leaves the integration of views to designers and implementers.

Furthermore, a statechart describing a view, in a viewchart, describes only a behavioral view of the system or, more importantly, its component. In other words, the number of states and the size of such a statechart are affected only by the scale of the behavioral view. Consequently, considering that a large-scale system behavior can be described in terms of simple behavioral views, Viewcharts simplifies the specification and reduces the complexity of scale that the underlying Statecharts notation faces in behavioral specifications of large systems

Others have also used the term "view", but not exactly for the same concept as we have. Embley and others [5], for example, introduce their notion of view, which is an abstraction mechanism for reducing complexity in large Object-Oriented Systems Analysis (OSA) models. Their notion of a view, however, is a grouping of some entities in the OSA models; it reduces the complexity of understanding a complex OSA model and communicating about it, but does not affect the complexity of specifying or building the model.

Goldberg [6], Ayers [1], and Rumbaugh [16], also describe the notion of view within Model-View-Controller (MVC) paradigm of Smalltalk community. A view, in MVC, refers to the method of presenting the information contained in the underlying model of an application. In regard to reducing the complexity of describing the model they neither make any claim nor offer any mechanism.

4.1 Contributions

The contribution of this paper is the introduction of the Viewcharts notation. The notation 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 world-wide, 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 get even 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 notation allows these views to be specified independent of each other.

4.2 Future Work

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

1. It is necessary to establish a semantic basis for Viewcharts. This is, however, the subject of another paper, in which we prove:

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

   The paper (which is currently under preparation) also provides an algorithm that translates a given viewchart to its equivalent statechart.

2. In a SEPARATE composition of views, Viewcharts does not allow transitions between the views. Because a SEPARATE composition of views in a viewchart is transformed to an AND composition of states in the corresponding statechart; and Statecharts does not allow transitions between ANDed states. However, it would be interesting to extend the semantic basis of Viewcharts to allow such transitions.

3. 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.

4. Finally, modeling viewcharts is another topic of future research. It is possible to produce an executable model of a viewchart using the algorithm mentioned in Item 1 and an available Statecharts tool (e.g., STATEMATE [8]). However, it would be more efficient and practical to provide a method of producing the executable models directly from the Viewcharts notation.

References

[1] K. E. Ayers. The MVC paradigm in Smalltak4. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 15(7):168–175, November 1990.

[2] 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.

[3] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or how to use Statecharts in object oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[4] 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.

[5] D. W. Embley, B. D. Kurtz, and S. N. Woodfield. *Object-Oriented Systems Analysis, A Model-Driven Approach.* Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[6] A. Goldberg. Information models, views, and controllers. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 15(7):54–61, July 1990.

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

[8] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[9] 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.

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

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

[12] F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):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 Transactions on Software Engineering*, 20(9):684–707, September 1994.

[14] 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, Mar 1994.

[15] 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.

[16] J. Rumbaugh. Modeling models and viewing views: A look at the model-view-controller framework. *Journal of Object-Oriented Programming*, pages 14–, May 1994.

[17] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[18] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling.* Wiley, New York, 1994.

[19] A. C. Shaw. Communicating real-time state machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.

[20] N. Walters. Using Harel Statecharts to model object-oriented behavior. *ACM SIGSOFT Software Engineering Notes*, 17(4):28–31, October 1992.