Behavioral Views for Software Requirements Engineering

by

Ayaz Isazadeh

A thesis submitted to the
Department of Computing and Information Science
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
September 1996

Copyright © Ayaz Isazadeh, 1996

بیاد پدرم
تقدیم به مادرم
به فرزندانم سارا، سیروس، شیلا
و برای همسرم نسرین

In memory of my father
To my mother
To my children Sara, Cyrus, Sheila
And for my wife Nasrin

بازی بودم پریده از عالم راز   تا بلکه رسم من از نشیبی بفراز
اینجا چو نیافتم کسی محرم راز   زان در که درآمدم برون رفتم باز

خیّام

I have flown like a sparrow-hawk forth from this world of mysteries, in the hope of reaching a higher sphere. But, fallen again to the earth and finding none worthy of sharing the hidden thoughts of my heart, I have gone forth again by the door through which I came.

Khayam
Translated by McCarthy

Abstract

Large-scale software systems, distributed or otherwise, are generally complex to describe, construct, manage, understand, and maintain. Current research approaches to reducing this complexity separate software structural and behavioral descriptions. It is important to study and analyze the behavioral as well as structural aspects of software systems. Much research continues on software structures and their patterns, characterizations, and classifications. Currently, research on the behavioral aspect of software systems includes using formal notations for specifying software behaviors and possibly refining the specifications to design and implementation. Large formal specifications, however, can be difficult to create and to understand; more research is needed into methods for assisting software requirements engineers in reducing these difficulties.

This dissertation introduces the idea of a software *behavioral view*: intuitively, this is a complete description of the behavior of the system observable from a specific point of view. We believe that a fully-developed methodology based on views would significantly reduce the complexity of creating and understanding software requirements. In this dissertation we take the first steps towards such a methodology. We define a formal notation, Viewcharts, with a well-defined semantics based on Statecharts. Viewcharts gives a means for precisely describing views and their compositions. We show that Viewcharts reasonably capture the informal idea of a view by giving two examples: a manufacturing control system, and a plain old telephone system. We show that Viewcharts have some advantages over Statecharts; in particular, Viewcharts adds name space control to limit the scope of broadcast communication, solving a problem with Statecharts presented by Harel.

Acknowledgments

I had the honor of working with Dr. David Lamb, as my supervisor, who let me work at my own pace on the topic and in the direction of my own choice. I thank him for his patience, support, and encouragement. I would like to also thank Dr. Glenn MacEwen and Dr. Andrew Malton for their supervision during the earlier stages of my Ph.D. program.

My special thanks are to Dr. James Cordy, a member of my supervisory committee, for his bold criticism and inspiring encouragement. I would like to also thank Dr. Terry Shepard, another member of my supervisory committee, for his high expectations in accuracy, clarity, and soundness, keeping me on my toes at all times.

I would like to thank members of my examining committee, Dr. Joanne Atlee of University of Waterloo, Dr. Karen Rudie of Queen's Electrical and Computer Engineering Department, Dr. Terry Shepard of Royal Military College of Canada, Dr. James Cordy, Dr. David Lamb, and the chair of the committee Dr. Ole Nielsen of Queen's Mathematics Department. I thank them for their careful review of this research, for their comments, and for giving me such a memorable defense.

I had numerous communications with Dr. David Parnas and Dr. David Harel, who have patiently replied to my mails and answered my questions; I thank them for their cooperation. I would like to also thank Dr. Thomas Marlowe of Seton Hall University, New Jersey, for reading the first draft of this dissertation and providing me with his comments.

I thank Dr. Dorothea Blostein, Dr. Patrick Martin, and Dr Robert Tennent for their advice and comments over the years.

I thank Homayoun Dayani-Fard, Hosein Isazadeh, Laurie Ricker, Medha Shukla Sarkar, Xiaobing Zhang, and all the members of our Software Technology Lab for listening to my half baked ideas, and encouraging me with their comments and suggestions. Special thanks are to Dr. Cordy, once gain, and Dr. Donald Jardine for shredding my presentations in our lab meetings and reshaping them according to academic standards.

I thank Irene LaFleche and the staff of our department for their support.

Finally, I would like to thank my wife, Nasrin, for her support and understanding.

Financial support for this research is provided in part by an Ontario Center of Excellence, the Information Technology Research Center (ITRC).

Contents

1 Introduction **1**

1.1 Terminology . . . 3

1.2 Problem and Solution Characteristics . . . 4

1.3 Motivation and Further Context . . . 5

1.4 Dissertation Outline . . . 6

2 Literature Review **8**

2.1 Views . . . 8

2.1.1 Views in DBMS . . . 9

2.1.2 The ViewPoint of Finkelstein . . . 10

2.1.3 Views in OSA Models . . . 11

2.1.4 MVC Paradigm of Smalltalk . . . 12

2.2 Scenarios and Use Cases . . . 13

2.3 Statecharts . . . 13

2.3.1 Overview of Statecharts . . . 14

2.3.2 Semantics of Statecharts . . . 18

2.3.3 Complexity of Scale in Statecharts . . . 21

2.4 Other Extensions of State Machines . . . 22

2.5 OOAD Methods 24

3 Viewcharts 26

3.1 Software Behavioral Views 26

3.2 Informal Description of Viewcharts 28

3.2.1 Ownership of Elements 29

3.2.2 Ownership and Triggering 31

3.2.3 Composing Behavioral Views 33

3.2.4 A General Example 39

3.2.5 History Transitions 41

3.2.6 Timing Issues 41

3.3 Conflicts 42

4 Formal Definitions and Semantics 45

4.1 Primitives 48

4.2 Definitions 49

4.2.1 Hierarchy of Views 50

4.2.2 Basic Views 51

4.2.3 Transitions 52

4.2.4 Composing Views 53

4.2.5 Ownership and Scoping 54

4.2.6 Initial Configuration 55

4.3 Viewcharts Equivalence to Statecharts 55

5 Examples 60

5.1 Manufacturing Control System 60

5.1.1 Specifying Behavioral Views 63
5.1.2 Composing Behavioral Views 66
5.1.3 Discussion 67
5.2 Telephone System 70
5.2.1 Specifying Behavioral Views 73
5.2.2 Composing Behavioral Views 77
5.2.3 Discussion 77

6 Conclusions **80**

6.1 Contributions 81
6.2 Demonstration of Claims 82
6.3 Discussion 84
6.4 Future Directions 86

Bibliography **89**

Vita **99**

List of Figures

2.1 Depth or hierarchy of states. 14
2.2 Orthogonality or AND-states. 15
2.3 An OR-state. 16
2.4 History transitions. 17
2.5 Timeout events and scheduled actions. 17
2.6 A possible infinite loop in a chain of events. 19
2.7 An instantaneous state. 20
2.8 Negated events. 20

3.1 Visual representation of SEPARATE compositions. 34
3.2 A SEPARATE composition of views in a viewchart. 35
3.3 An OR composition of views. 37
3.4 An AND composition of views. 38
3.5 Composition of views in a viewchart. 40
3.6 Consistent views in a viewchart. 43
3.7 Conflicting views in a viewchart. 44

4.1 A Statecharts translation of the viewchart shown in Figure 3.2. 46

4.2 A Statecharts translation of the viewchart shown in Figure 3.4. 47
4.3 A Statecharts translation of the viewchart shown in Figure 3.5. 48

5.1 Product and information flow in a manufacturing line. 61
5.2 A viewchart for the Serialization Workstation. 64
5.3 The workstations' views of the system. 65
5.4 A viewchart for a manufacturing line. 66
5.5 An attempt to specifying MCS in Statecharts. 68
5.6 A high level scenario for establishing a telephone connection 71
5.7 A detailed scenario for establishing a telephone connection 72
5.8 The caller view of a telephone set. 73
5.9 The called view of a telephone set. 74
5.10 A telephone set's view of POTS. 75
5.11 A viewchart for the telephone service provided by POTS. 76
5.12 A portion of a statechart specifying POTS. 78

Chapter 1

Introduction

Large-scale software systems, distributed or otherwise, are generally complex to describe, construct, manage, understand, and maintain. Current research approaches to reducing this complexity separate software structural and behavioral descriptions [5, 40, 54, 56]. It is important to study and analyze the behavioral as well as structural aspects of software systems. Much research continues on software structures and their patterns, characterizations, and classifications [13, 14, 24]. Special languages, called *configuration languages*, supported by *configuration management systems,*[1] are designed for describing the structure of a software system [45]. Currently, research on the behavioral aspect of software systems includes using formal notations (e.g., Statecharts [27, 28, 32], ESTEREL [7], Z [9, 70], VDM [50], etc.) for specifying software behaviors and possibly refining the specifications to design and implementation. Large formal specifications, however, can be difficult to create and to understand;

[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.

more research is needed into methods for assisting software requirements engineers in reducing these difficulties.

A possible approach to reducing the complexity of creating and understanding requirements specifications is based on the following ideas. A common approach to reducing complexity is to apply divide-and-conquer: divide a large task into smaller ones, each simpler than the original, whose combination solves the larger problem. In the requirements area, customers commonly express their requirements using *scenarios*: descriptions of particular detailed interactions between the system and its environment, usually focusing on combinations of a subset of the possible boundary-crossing events. Such scenarios are typically small (and thus relatively easy to understand). Can we find an effective way to capture and combine such scenarios?

Scenarios typically present two problems.

- First, they are *incomplete.* That is, they pick some portion of the requirements and describe *one* or a *few* possible interactions, but neglect other possible events (especially failure conditions) that could arise during the same interaction. Thus a book-borrowing scenario might fail to consider what happens when a library patron tries to borrow too many books at one time. Can we give a *complete* description of something akin to a scenario, while still focusing on only a subset of the possible events?

- Second, they are *unstructured.* Informal approaches to handling scenarios fail to show how to combine scenarios to give a complete requirements specification. This is left to the analyst, who integrates the scenarios into a requirements specification, in which the original scenarios may not be readily visible.

This dissertation introduces the idea of a software *behavioral view*: intuitively,

this is a complete description of the behavior of the system observable from a specific point of view (Chapter 4 defines this formally). We will also refer to a behavioral view by just "view". We believe that a fully-developed methodology based on views would significantly reduce the complexity of creating and understanding software requirements. In this dissertation we take the first steps towards such a methodology. We define a formal notation, Viewcharts, with a well-defined semantics based on Statecharts. Viewcharts gives a means for precisely describing views and their compositions. We show that Viewcharts reasonably capture the informal idea of a view by giving two examples: a manufacturing control system, and a plain old telephone system. We show that Viewcharts have some advantages over Statecharts; in particular, Viewcharts adds name space control to limit the scope of broadcast communication, solving a problem with Statecharts presented by Harel [27, 28, 31, 32].

1.1 Terminology

Configuration of a System: If we represent a system by a single 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 state of the FSM. However, if we represent the system by an extended FSM, such as Statecharts or Viewcharts (where the system can be in many states of these machines at any instant in time), then we have to distinguish the notion of state in these machines from the state of the system. To avoid any confusion caused by the overloaded use of the term "state", we will refer to the state of a system as the *configuration* of the system. A system configuration, therefore, can be represented by a set of states in Statecharts or Viewcharts. To be accurate, this set describes the *basic configuration* of the system; however, until we define the full

configuration of the system (in Chapter 4) we will omit the term “basic”.

Specification-in-the-large versus specification-in-the-small: Hoffman and Strooper [39] define *software engineering* as multi-person multi-version software systems development; such systems are generally referred to as *large* systems. (The definition is attributed to David Parnas.)

Analogous to programming-in-the-large versus programming-in-the-small [14], we further define *specification-in-the-large* versus *specification-in-the-small*: If we express the specification of a system as a composition of other specifications, then we have a specification-in-the-large, otherwise it is a specification-in-the-small. Specification-in-the-small may not be practical for large software systems. We believe that requirements specification of large systems requires methods of specification-in-the-large.

1.2 Problem and Solution Characteristics

The problem addressed by this dissertation is to discover whether behavioral views can serve as the basis for a software development methodology that could plausibly be expected to reduce the complexity of formal specification of large-scale software systems.

A solution to this problem must have the following necessary characteristics:

1. Give a precise meaning to the term “behavioral view”.

2. Provide a way to record behavioral views.

3. Provide a means of composing views to form a complete system specification.

4. Traceability: the composition (item 3) should preserve the identity of the composed views so that a given requirement can be traced back to its originating

view.

5. Practicality: at least as expressive as notations accepted as practical.

It should also, where possible, have the following desirable characteristics:

6. Independence from design: it should not force the specifier to make design decisions.

7. Modularity: it should allow modular specifications of software behavioral requirements so that they can be reused in composing specifications. The advantages of achieving a high degree of modularity in composing specifications have long been recognized [1, 72].

The central claim (thesis) of this dissertation is that the Viewcharts notation introduced in Chapter 3 exhibits properties 1-5, and thus is a plausible basis for a software development methodology oriented around behavioral views.

1.3 Motivation and Further Context

The following are two major motivating factors, for this dissertation, that leads to the Viewcharts notation:

1. *Most of the defects in software systems can be traced back to the requirements phase.* Specifically, studies in Bell Labs and IBM have shown that 80% of all defects in software systems are inserted in the requirements phase [68].

   Accurate requirements specification of software systems, therefore, will improve quality and increase reliability of the software. Accurate requirements specifications, in turn, requires a formal method.

2. *Formal methods have not been practical for large-scale complex systems.* Based on experience with the A-7 project [36], John Guttag and others [26] conclude that one problem with formal methods is size. The difficulties of managing a large volume of formal specifications have made formal methods impractical for large systems.

We have chosen Statecharts as a basis for Viewcharts. This is due to the fact that Statecharts has many desirable characteristics: It is a formal and visual notation; and its notions of depth, broadcasting, and orthogonality, as described in Section 2.3, make it a useful notation for industrial applications.

Statecharts is designed for real-time event-driven reactive systems. Viewcharts extends Statecharts to include behavioral views and their composition; consequently, it describes behavioral requirements of large systems as compositions of views. The domain of the Viewcharts formalism, therefore, is behavioral specification of large-scale real-time event-driven reactive systems.

1.4 Dissertation Outline

This dissertation consists of five chapters. This section concludes Chapter One, where we stated the problem and provided a list of desirable characteristics of the proposed solution. The list characterizes a good solution, basically, as a formal notation capable of reducing the complexity of behavioral requirements specifications of large-scale systems. We will use this capability as a basis of our literature review, discussed in Chapter Two. Chapter Three describes the proposed solution to the problem by an informal introduction of software behavioral views and the Viewcharts notation. The formal definitions and semantics of Viewcharts are provided in Chapter Four. Chapter

Five presents some examples, demonstrating the way in which software behavioral requirements can be engineered using Viewcharts. Finally, Chapter Six returns to the list of desirable characteristics of the proposed solution, presents a discussion of the Viewcharts notation with respect to the list, and concludes the dissertation with a summary of the results and future directions.

Chapter 2

Literature Review

The notion of view has been used in the literature with different meanings and for different purposes. This chapter presents a review of views as they are used in the literature.

There is also a body of work on Statecharts, other state-transition based formal methods, and software behavioral requirements specifications using these methods. Considering that Viewcharts is based on Statecharts, a review this work is also presented.

2.1 Views

In the literature, generally, the term "view" refers to two different notions:

- The notion of view as a (formal or informal) partial description of an object, model, or system [6].

- The notion of view as the behavior of an existing object, model, or system

observable from a specific point of view [4, 20, 25, 60].

This section presents a review of these notions, points out their differences with our notion of view, and discusses their strength and limitations in reducing the complexity of software behavioral requirements specification.

2.1.1 Views in DBMS

In existing database management systems, a *view* refers to a subset of the database; it may also include "virtual" data, which is derived from the database, but is not explicitly stored [10, 19, 64]. A database view, therefore, is a convenient way of providing the data of interest to a user or a group of users. The mechanism of views, in a database system, can also be used for authorizing a user to access selected data and hiding the rest of the database.

In a database design process, Batini and others [6] use the term "view" to refer to the database requirements of an application as seen by a user or a group of users. They describe a conceptual approach to database design, where different views of a database can be designed and integrated to form a global conceptual schema. The conceptual schema then leads to a logical schema which, in turn, is used as a basis for the physical design of the database.

A database view, therefore, either provides a user's view of the database or facilitates the process of designing a conceptual schema for the database. In any case, it is not designed for (and does not help in) reducing the complexity of behavioral requirements specification of the database system. A database view, as defined by Batini and others [6], is similar to the notion of view introduced by this dissertation; and it can be considered as a formal requirements specification of a user's view of

the database. However, their objective is design and implementation and they use database views as a design mechanism. Consequently, the views are influenced by the implications of implementation. Finally, they neither claim nor offer any mechanism in reducing the the complexity of behavioral requirements specification of the system under design.

2.1.2 The ViewPoint of Finkelstein

Finkelstein and others [22, 23, 57] define *ViewPoints* as "loosely coupled, locally managed, distributed objects encapsulating partial representation knowledge, development process knowledge and specification knowledge, about a system and its domain". Each ViewPoint is associated with:

- a *style*, which describes the notation used by the ViewPoint,
- a *work plan*, which describes the development plan of the ViewPoint,
- a *domain*, which describes the area of concern of the ViewPoint,
- a *specification*, which describes the *domain* using the *style* according to the *work plan*, and
- a *work record*, which describes the current state and history of the *specification*.

A ViewPoint represents a partial specification of a system. Different ViewPoints can be developed using different notations. The overall system specification then is described as a configuration of different ViewPoints.

ViewPoints are allowed to overlap (and possibly contradict each other), resulting in inconsistencies within the specification. The creators of the ViewPoint framework,

therefore, have devoted a considerable amount of work on the issue of inconsistency management [18]. Furthermore, the ViewPoint framework is basically a software development framework. It is designed to cover all phases of software development process, from requirements to evolution, and to facilitate distributed development of different ViewPoints by different developers. There is a correlation between the configuration of ViewPoints used in the software process and the resulting software structure. Consequently, the requirements specification phase of the ViewPoint framework violates the independence of specification from design, item 6 of the desirable characteristic of our proposed solution (presented in Section 1.2).

2.1.3 Views in OSA Models

Embley and others [20] introduce their notion of view, which is an abstraction mechanism for reducing complexity in large Object-Oriented Systems Analysis (OSA) models. They provide multiple levels of views for every type of OSA construct (object classes, relationship sets, states, and transitions). A view can contain other views and, consequently, there can be a hierarchy of views. Their notion of view, however, is similar to the concept of depth in Harel's Statecharts. Consider a superstate, in Statecharts, which includes some substates; the superstate (an abstraction of the substates) corresponds to their notion of a view. Furthermore, they use a bottom-up approach to defining views: one has to construct the detailed model, then apply the abstraction mechanism. This mechanism provides a convenient way to refer to parts of an OSA model, in different levels of abstractions, by the corresponding views. The mechanism, therefore, helps one to understand the model and communicate about it, but does not help reduce the complexity of specifying or building the model. A view

is, in fact, a grouping of some entities in the OSA models. A top-down approach, using this notion of view, where one can start with an abstract model and refine it down to the detailed levels, also will not reduce the complexity of describing the model. In any case, the detailed model is the objective; and their notion of view can help construct it (in a top-down approach) or understand and communicate about it (in a bottom-up approach), but does not affect the complexity of specifying the model.

2.1.4 MVC Paradigm of Smalltalk

Adele Goldberg [25] and Kenneth Ayers [4] talk about views within the Model-View-Controller (MVC) paradigm of the Smalltalk community. The MVC paradigm distinguishes three types of components in a Smalltalk application: model, view, and controller. The model describes the underlying information and computation of the application; the view refers to the method of presenting the information contained in the model; and the controller controls the interaction between the model and its views. Views in MVC help create (graphical) user interfaces, reuse interface design across different applications, and accommodate diverse user preference in interfacing with a model. However, they do not reduce the complexity of describing the model.

James Rumbaugh [60] presents a slightly more general discussion of the views within the MVC paradigm. However, in regard to reducing the complexity of describing the model (or in his terminology "subject"), he neither makes any claim nor offer any mechanism.

2.2 Scenarios and Use Cases

Ivar Jacobson [47] defines a *use case* as a behaviorally related sequence of transactions that a "user" performs in a dialogue with a system. The *user* is regarded as an instance of a class called *actor* and the set of all use case descriptions (classes) specifies the complete functionality of the system. The environment of a system, therefore, is characterized (informally) by actors and the transactions between the system and its environment are specified by the use case descriptions.

Jacobson's notion of use case is analogous to our notion of view. Use cases, as views, describe partial specifications; therefore, there can be inconsistencies in use cases, as in the views. Use cases, however, are defined and manipulated informally, while views, as described in Chapter 3, are defined and composed formally.

2.3 Statecharts

Since the Viewcharts notation is based on Statecharts, therefore, this section presents a more comprehensive review of Statecharts.

Introduced by David Harel [27, 28, 31, 32], Statecharts is an extension of Finite State Machines (FSM) designed as a formal method for behavior specification of complex systems. The term *Statecharts* refers to the specification method, while a specification written in Statecharts is called a *statechart*.

Furthermore, Harel and others [30] have developed a set of tools called STATEMATE for the specification, design, analysis, and documentation of large and complex reactive systems. STATEMATE uses Statecharts for behavior specification of a system

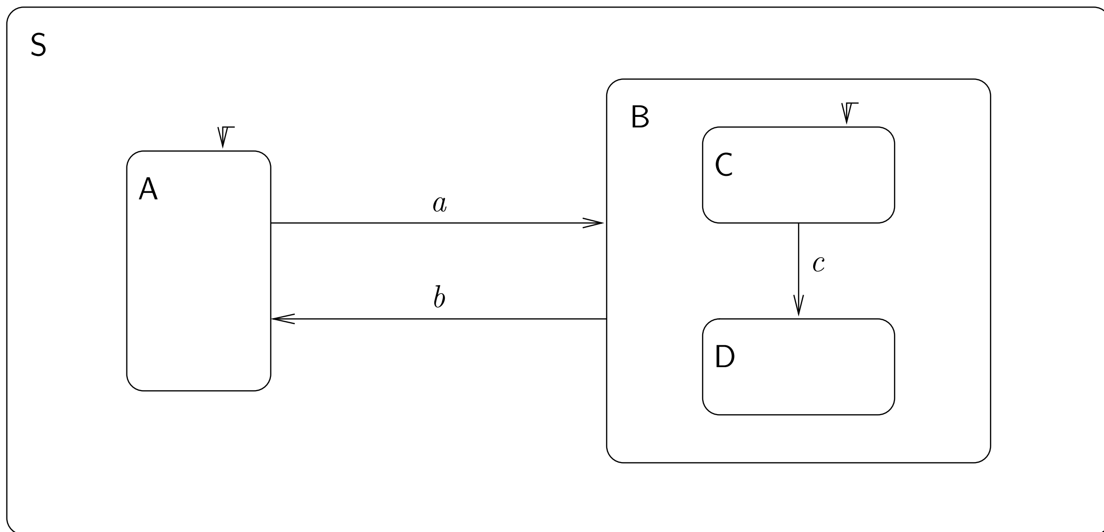

Figure 2.1: Depth or hierarchy of states.

under development. In addition, STATEMATE provides *module-charts* and *activity-charts* for specifying the structural and functional view of the system, respectively.

2.3.1 Overview of Statecharts

Harel describes Statecharts as:

state diagrams + depth + orthogonality + broadcast communication.

Depth refers to a clustering of some states into a superstate and thus forming a hierarchy of states. Figure 2.1 demonstrates the notion of depth in Statecharts. The states C and D, in this figure, are clustered into the superstate B. The complete statechart S is a superstate containing B and A.

Orthogonality refers to a composition of two or more states into a superstate called an AND-state. If a system enters into an AND-state, it must enter in all the AND components (i.e., immediate substates) of the AND-state. Similarly, if a system

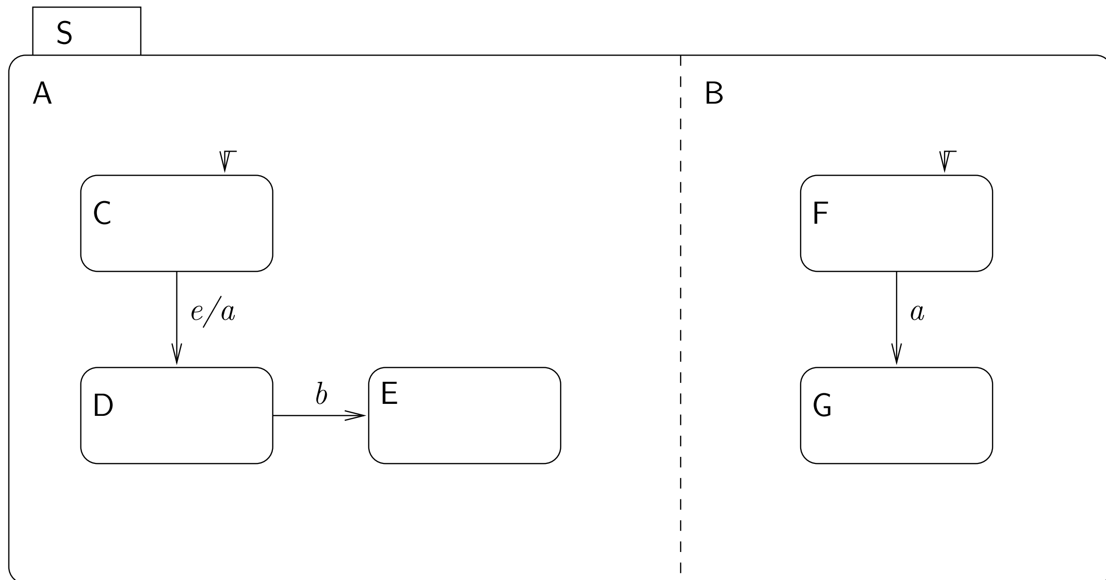

Figure 2.2: Orthogonality or AND-states.

exits from an AND-state, it must exit from all the AND components of the AND-state. Figure 2.2 provides an example of orthogonality in Statecharts. The state S, in this figure, consists of two orthogonal components: A and B. Being in the state S means that the system must be in both A and B. The states C and F are specified as the default states: when the system enters S, it is in both C and F. If S represents a system, then the initial configuration of the system is described by the set {C, F}.

In contrast to AND-states, there are also OR-states in Statecharts. If a system enters into an OR-state, it must enter in only one of the immediate substates of the OR-state. Figure 2.3 provides an example of OR-states in Statecharts. The state S, in this figure, consists of two states: S1 and S2. Being in the state S means that the system must be either in S1 or S2; it cannot be in both states. The state S1 is specified as the default state: when the system enters S, it is in S1. If S represents a system, then the initial configuration of the system is described by the set {S1}.

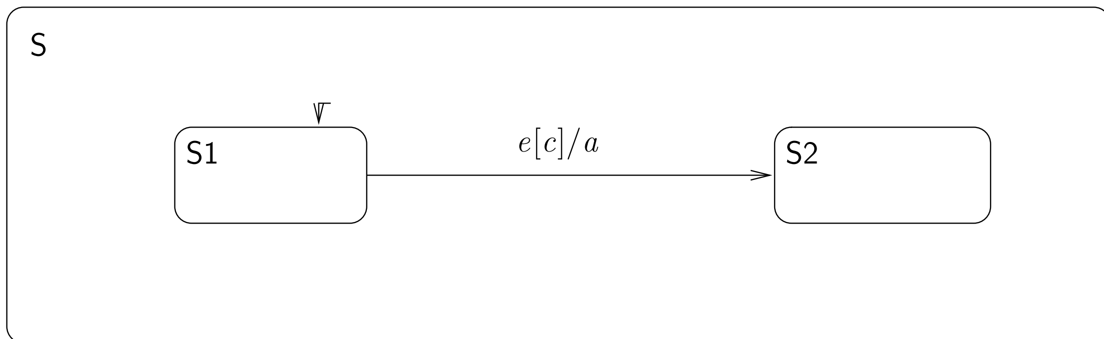

Figure 2.3: An OR-state.

A transition label, in Statecharts, can have three components: *event, condition,* and *action.* In Figure 2.3, for example, if the system is in the state S1 and the event e occurs, while the condition expression c evaluates to true, then the transition from S1 to S2 takes place and the action a is generated. An action generated by a transition can be used as an event in another transition. For example, in Figure 2.2, if the system is in configuration $\{C, F\}$ and the event e occurs, then the transition from C to D takes place, generating the action a which, in turn, triggers the transition from F to G. Notice that in this example an action (a) is generated by one orthogonal component (A) and used by another one (B). The action must be communicated between the components. The communication mechanism used in Statecharts is broadcasting. For example, when an event occurs or an action is generated, in a statechart, it is sensed throughout the statechart.

Statecharts supports *history* and *deep history* transitions, whose targets are history connectors marked H and H*, respectively. The connectors H and H* must reside somewhere within the area of a state. If we call this state B, then an H-transition is directed to the most recently visited immediate substate of B and an H*-transition is directed to the most recently visited lowest level substate(s) of B. An example of

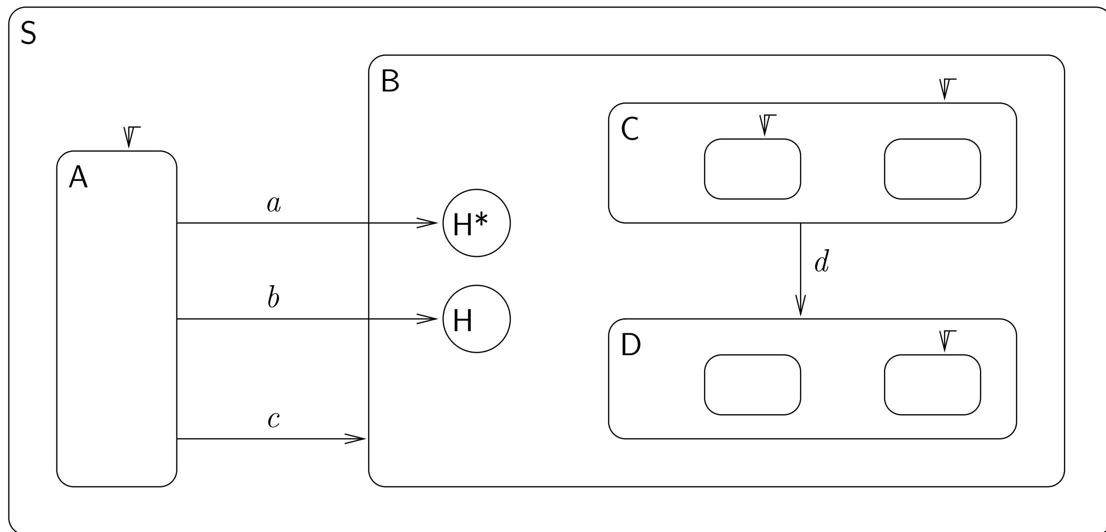

Figure 2.4: History transitions.

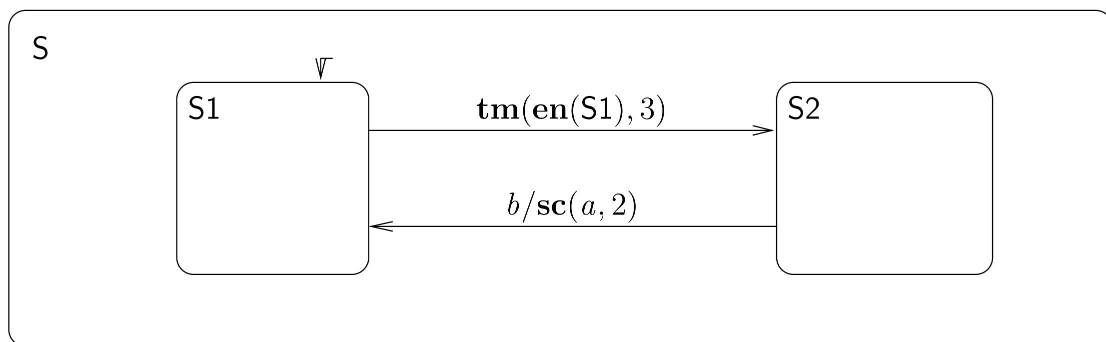

Figure 2.5: Timeout events and scheduled actions.

these transitions are shown in Figure 2.4. Notice that the connectors H or H* must reside somewhere within the area of a state. Since an AND-state has no area of its own, therefore, B cannot be an AND-state.

Statecharts also supports *timeout* events and *scheduled* actions. A timeout event, specified as $\mathbf{timeout}(e, \mathrm{t})$, abbreviated as $\mathbf{tm}(e, \mathrm{t})$, occurs at exactly t time units after the last occurrence of the event e. For example, in Figure 2.5, the event $\mathbf{tm}(\mathbf{en}(\mathsf{S1}), 3)$ occurs at exactly 3 time units after the system enters the state S1. ($\mathbf{en}(\mathsf{S1})$ is an event

that occurs at a point in time that the system enters the state S1.)

A scheduled action a, specified as **schedule**(a, t), abbreviated as **sc**(a, t), is triggered at exactly t time units from the present instance. For example, in Figure 2.5, assuming that the system is in the state S2, the action a is triggered at exactly 2 time units after the event b occurs.

2.3.2 Semantics of Statecharts

There are a variety of issues regarding the semantics of Statecharts. Different resolutions of these issues have resulted in different variations of Statecharts semantics [12, 31, 33, 41, 42, 43, 44, 52, 53, 55, 59, 66]. The Viewcharts notation, as described in Chapter 3, encapsulates these variations and, consequently, is not restricted to a particular variation of Statecharts. Furthermore, Viewcharts is not concerned with the way in which the semantics of the different versions of Statecharts resolve the related issues. We will, therefore, outline the issues, but will not get into the details of their resolutions by different semantics of Statecharts:

- **Synchrony Hypothesis:** This hypothesis states that a system reacts to an external event immediately. Therefore, events, actions, and checking the value of a condition expression ideally take no time; and transitions are instantaneous. Most Statecharts semantics, including Harel's version [31], are based on this hypothesis; exceptions include [12, 52].

- **Macro and Micro Steps:** A *(macro) step*, also called *super step*, is initiated by an external event, causing a chain of internal events. The step is completed when the system stabilizes; i.e., when no more internal events are generated or no more transitions are triggered by the chain of events. Based on the synchrony

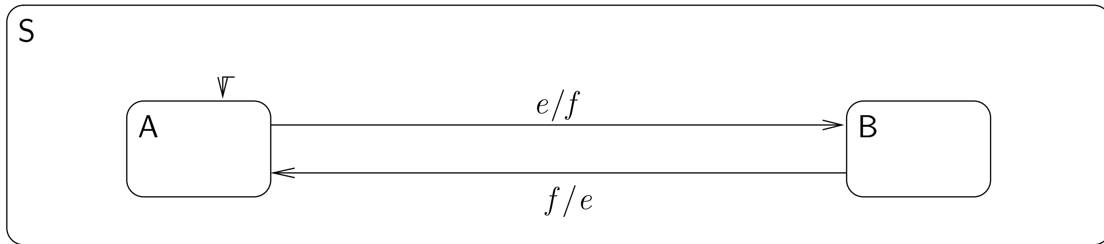

Figure 2.6: A possible infinite loop in a chain of events.

hypothesis, a step is instantaneous and, therefore, all the events within a step occur instantaneously. Most Statecharts semantics, including Harel's version [31], distinguish an order of occurrence between the events within a step. These events occur at different *micro steps* within the macro step. Exceptions include the semantics type D of [43], where all the events occur at the same time.

- **Causality:** A Statecharts semantics is *causal* if each executed transition can be traced back to the occurrence of an external event, where the chain of events starting from the external event leading to the transition has no cycle. In the statechart of Figure 2.6, for example, a chain of events can form an infinite loop, if the underlying semantics is not causal. Most Statecharts semantics respect causality. Exceptions include the models where no order of occurrence is distinguished between the events occuring within a macro step, e.g., the semantics type D of [43].

- **Instantaneous States:** The issue here is whether or not a system is allowed to simultaneously enter and exit a state. In the statechart of Figure 2.7, for example, if the event e occurs when the system is in the state A, should the next configuration be {B} or {C}? Some of the Statecharts semantics allow entering and exiting a state simultaneously [31, 53] and others do not [41, 44].

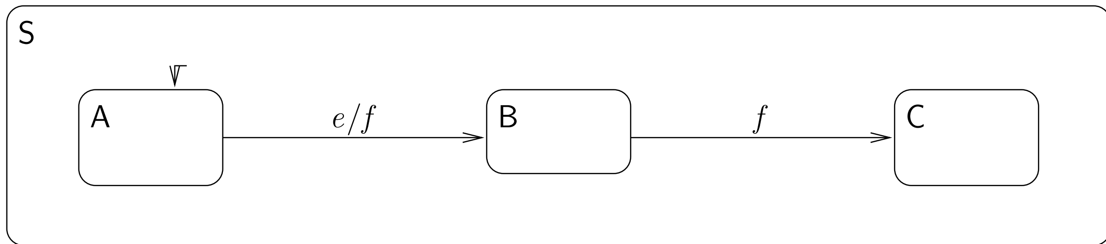

Figure 2.7: An instantaneous state.

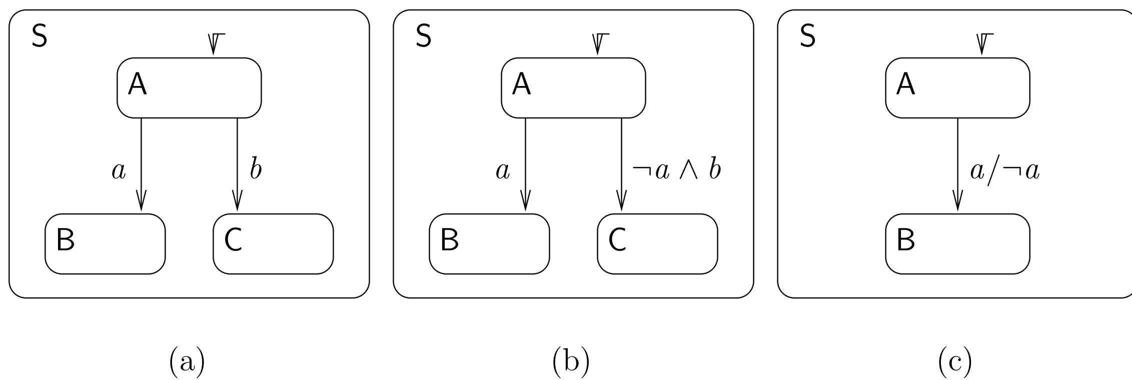

Figure 2.8: Negated events.

- **History Transitions:** Some Statecharts semantics allow specification of history transitions [31, 33] and some others do not [41, 44, 53].

- **Negated Events:** If e is an event, should we allow the negated event $\neg e$ as another event? Doing so helps avoid nondeterminism in transitions. In the statechart of Figure 2.8-(a), for example, if the system is in the state A and the events a and b occurs simultaneously, nondeterministically then the next state of the system will be either B or C. Using a negated event, as shown in Figure 2.8-(b), can avoid this nondeterminism. However, once we allow using negated events, we have to be prepared for situations where a transition negates its cause, as shown in Figure 2.8-(c).

2.3.3 Complexity of Scale in Statecharts

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 [15, 16] prove that Statecharts is exponentially more succinct than finite state machines. The proof is based on the co-operative concurrency mechanism (i.e., orthogonality) of Statecharts and applies for other models that use this mechanism, such as Petri Nets [58] or CSP [11, 37]. 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; in Harel's words "Statecharts represents the scale of the system" [29]. Orthogonality is, indeed, 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 [69].) When an event occurs, it is sensed throughout the system and, therefore, 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 [51]. More research is needed into methods for managing the name space in Statecharts.

2.4 Other Extensions of State Machines

The following is a list of some other extensions of state machines or variations of Statecharts. Leveson's RSML and Selic's ROOMcharts, as described below, provide some mechanisms for reducing the complexity of managing name space. Besides that, none of these machines provide any solutions for the complexity of scale.

- Jahanian and Mok introduce another specification language, called Modechart [49], for real-time systems. They also define the semantics of Modechart in terms of Real Time Logic (RTL) [48]. Modechart is originated based on the mode concept of Parnas [36, 65] and Statecharts of Harel; its emphasis, however, is on the specification of absolute timing properties.

- Alan Shaw [63] describes an executable notation, based on communicating real-time state machines (CRSM's), for specifying concurrent real-time systems. CRSM's are state machines that communicate with each other using a CSP-like synchronous scheme [37, 38]. Shaw also provides an algorithm for simulating CRSM's and some techniques for reasoning about the system behavior.

- Charles Hendricksen [35] describes another extension of finite state machines, called the *Augmented State Transition Diagram* (ASTD), and an associated CASE tool called *State-Graph*. ASTD has been used in the definition, design, and implementation of some applications including a PBX phone system and some complex user interface programs.

- Nancy Leveson and others [53] describe their approach to behavior specification of a real aircraft *Traffic Alert and Collision Avoidance System* (TCAS). The

specification language used for this system is a variation of Statecharts called *Requirements State Machine Language* (RSML). In RSML, physically distinct components are modeled as separate communicating statecharts. The overall system requirements specification can be viewed as a directed graph (not a statechart), where each node represents a component and each edge represents an intercomponent communication channel. The broadcast communication mechanism of Statecharts is used within each component. Intercomponent communication is provided as directed messages transmitted between components over unidirectional channels. An event, therefore, is local to the component in which it occurs. The event does not affect any other component, unless directly transmitted to another one. Therefore, RSML provides a visibility control mechanism that reduces the complexity of name management. The mechanism, however, has a negative consequence: the direct communication method used for intercomponent communications complicates the specifications.

- ROOM [62] is a specialized high-level modeling language,designed for distributed real-time systems and supported by a modeling environment, the ObjectTime toolset. Software behavior in ObjectTime is expressed by ROOMcharts, which is an extension of Statecharts. ROOMcharts replaces the AND-states of Statecharts by encapsulated entities called *actors* (similar to RSML). It also replaces the broadcast communication of statecharts by a port-to-port message-passing mechanism for communications between the actors. As a result, it reduces the complexity of managing name space.

  ROOM starts with the high level design of software components and leads to

their implementations. ROOM is not designed for software requirements specification; it is basically a design notation. However it can be used for requirements analysis at a very high level design phase by prototyping and trying out alternative designs.

The Message Sequence Charts (MSC) of ROOM is similar to our notion of view. MSC can be used for partial specification of the observable behavior of a system. It is, therefore, possible to relate Viewcharts to ROOMcharts via MSC; however, this requires further studies.

ROOMcharts and RSML are further discussed in Section 6.3, Page 84.

2.5 OOAD Methods

Some object-oriented analysis and design (OOAD) methods describe the behavior of objects and classes using variations of extended finite state machines. This includes a considerable amount of work in the area of inheritance and refinement of software behavior. Some of this work is outlined below:

- James Rumbaugh and others [61] use an extension of state diagram, based on Harel's Statecharts, to describe the *dynamic model* of Object Modeling Technique (OMT).

- Neal Walters [67] expands on Rumbaugh's work by providing mechanisms for object collaborations: the invocation of object services and interchange of data between objects. Emphasizing the importance of predicting system behavior for validating completeness and analyzing performance, he describes a method for building dynamic models of object-oriented systems using STATEMATE.

- Derek Coleman and others [8] introduce an extension of Statecharts called *Objectcharts* for object-oriented design. They use Objectcharts to specify the behavior of objects and expect that the future work will provide firm semantics for Objectcharts, enabling the behavior of object-oriented systems to be deduced from the specifications of the corresponding objects.

Chapter 3

Viewcharts

Supporting Items 2-4 of the necessary characteristics of the solution, this chapter introduces, informally, the concept of *software behavioral views* and a formal notation for specifying software behavior in terms of the behavioral views. (The formal definitions are presented in Chapter 4.) The notation is an extension of David Harel's Statecharts and, therefore, I call it *Viewcharts.* As in Statecharts, where the term *Statecharts* refers to the specification method, while a specification written in Statecharts is called a *statechart*, the term *Viewcharts* also refers to the notation, while a behavior specification using Viewcharts is called a *viewchart.*

3.1 Software Behavioral Views

A *(behavioral) view* of a software system is the behavior of the system observable from a specific point of view. (The formal definition of *view* is given in Chapter 4.) 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.

In the process of eliciting system requirements, software requirements engineers usually conduct interviews with potential users. The users describe their expectations of the system; in other words, each user describes his or her view of the system. A natural outcome of requirements elicitation, therefore, is a set of behavioral views. Traditionally, however, requirements engineers combine the behavioral views and provide a formal or informal requirements specification for the entire system. Consequently, the specification has no formal connection to the original views described by the users. This method of requirements engineering leads to the following issues:

- Considering that both the behavioral views (described by the user) and their integration (by the requirements engineer) are informal, there is no guarantee that the resulting requirements specification (either formal or informal) will reflect the original behavioral views.

- Integration of the behavioral views is a design and implementation issue. If the objective is software behavioral requirements specification independent of design and implementation (as it is the case for this dissertation), then integration of the behavioral views by requirements engineers is an unnecessary task that complicates the specification and limits the design choices.

I argue, therefore, that software behavioral requirements specifications should be expressed in terms of the behavioral views:

- Behavioral views can be viewed as stand-alone systems and should be described as such.

- Behavioral views should be defined (formally) and confirmed with the potential users.

- The overall system behavioral requirements should be specified (formally) in terms of the behavioral views. This is in contrast to the traditional approach, where the specification has no direct relation with the behavioral views, the informal requirements collected in the process of requirements elicitation.

- Integration of the behavioral views should be left to designers and implementers. Designers and implementers are then expected to deliver a system that provides the formally specified behavioral views.

This method of software behavioral requirements specification, compared to the traditional approach, simplifies the specification, directly reflects the behavioral views of the system, and thereby accurately records the original requirements collected in the process of requirements elicitation.

3.2 Informal Description of Viewcharts

The Viewcharts notation is based on Statecharts. Statecharts, however, has no concept of behavioral views. Viewcharts extends Statecharts to 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 3.2.3 discusses extended scopes.)

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, I believe that further experimental work will show that Viewcharts can eliminate the issue of scale. (Note that this is not a claim.)

3.2.1 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 3.2.2 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 in the view 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 must have at least one owner. 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 $\mathsf{V1}$ within the scope of x, then Viewcharts recognizes two different variables which can be referenced by their *qualified 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 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. A variable must have exactly one owner.

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 3.2, 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.

3.2.2 Ownership and Triggering

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, consequently, 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 actually triggered the event. As we will see in Chapter 4, for every viewchart there exists an equivalent statechart whose events are of the form $V.e$, where V is a view and e is an event of the viewchart. When the event $V.e$ occurs in the statechart, we say the view V triggers the event e in the corresponding viewchart. The scope of the event then is the view that triggers the event and possibly some other views as described in Section 3.2.3.

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. For example, we may say "the view V triggers the event e" or "$V.e$ occurs"; to state that "e occurs" is ambiguous in Viewcharts.

An event can be triggered only by a view that owns the event. The notions of ownership, scoping, and triggering are formally defined in Chapter 4, pages 54 and

58. Here we provide an intuitive and informal description on the notion of ownership.

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*. In 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.

The notion of *ownership* for an element (event, action, or variable) is a natural consequence of composing views, while the scope of the element is limited. 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 consequently extended scope, then it is not owned by the view. (Section 3.2.3 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.

3.2.3 Composing Behavioral Views

Views can be composed in three ways: SEPARATE, OR, and AND. The compositions keep the identity of the composed views intact and, thereby, support Item 4 of the necessary characteristics of the solution. Except for the effect of ownership and scoping restrictions, the OR and AND compositions of views, in Viewcharts, are similar to the OR and AND compositions of states, in Statecharts, respectively. The SEPARATE composition of views, however, is specific to Viewcharts. In a 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: All the subviews and states in a superview are visible to the superview. The scope of the elements owned by a superview covers all its subviews.

SEPARATE **Composition of Views**

In a SEPARATE 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. A SEPARATE composition of views forms a superview called SEPARATE-view.

Visually, the views involved in a SEPARATE composition are drawn on top of each other, as shown in Figure 3.1, giving the impression that they are located on different planes and, consequently, are hidden from each other.

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

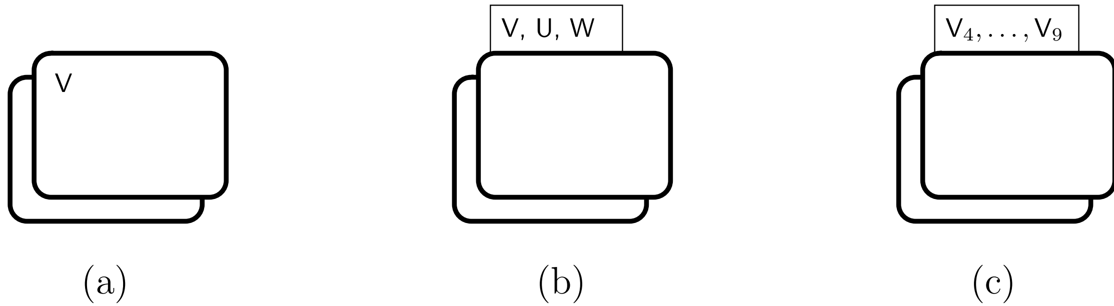

Figure 3.1: Visual representation of SEPARATE compositions.

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 six views $V_4, \ldots, V_9$. 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 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 3.2, which specifies a SEPARATE composition of views V1 and V2.

The event a, in the viewchart of this figure, belongs to V1, but does not belong to V2; the event c belongs to V2, but does not belong to V1; and the event b belongs to both V1 and V2. In any case, no event of V1 can trigger a transition in V2 or vice versa. However, the event a also belongs to V which, because of the encapsulation mechanism described above, can trigger simultaneous transitions in both V1 and V2. The following examples demonstrate the way in which the composition affects transitions with the same label.

Recall (Section 3.2.1) that a view can trigger the events it owns. Assuming that

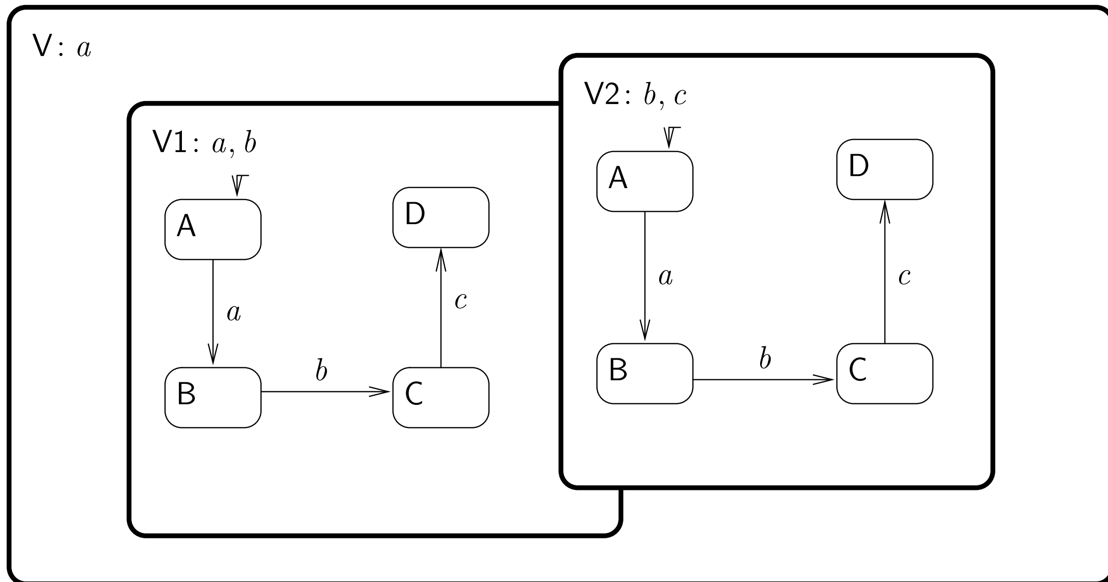

Figure 3.2: A SEPARATE composition of views in a viewchart.

the system is in configuration {V1.A, V2.A},

- if the view V triggers a, then the next configuration will be {V1.B, V2.B} (because of the encapsulation mechanism described above);
- if the view V1 triggers a, then the next configuration will be {V1.B, V2.A};
- V2 cannot trigger a, because it does not own a.

Assuming that the system is in configuration {V1.B, V2.B},

- if the view V1 triggers b, then the next configuration will be {V1.C, V2.B};
- if the view V2 triggers b, then the next configuration will be {V1.B, V2.C};
- V cannot trigger b, because it does not own b.

Assuming that the system is in configuration {V1.C, V2.C},

- if the view V2 triggers c, then the next configuration will be {V1.C, V2.D};
- neither V nor V1 can trigger c, because neither owns c.

OR Composition of Views

The OR and SEPARATE compositions are similar except that in an OR composition only one view can be active, but 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. An OR composition of views forms a superview called OR-view.

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. This is a special case of the way in which conflicting transitions are handled in Viewcharts, which is similar to that of Statecharts. Generally, in case of a conflict between two transitions, the priority is with the one for which the nearest common ancestor of the source and destination views or states is of the higher level.

Figure 3.3 shows a viewchart demonstrating an OR composition of views. Some examples of the transitions that are affected by the composition are as follows. Assuming that the system is in configuration {V1.A},

- if the view V1 triggers a, then the next configuration will be {V1.C};
- if the view V1 triggers b, then the next configuration will be {V1.B};
- if the view V triggers b, then the next configuration will be {V2.A} (notice that

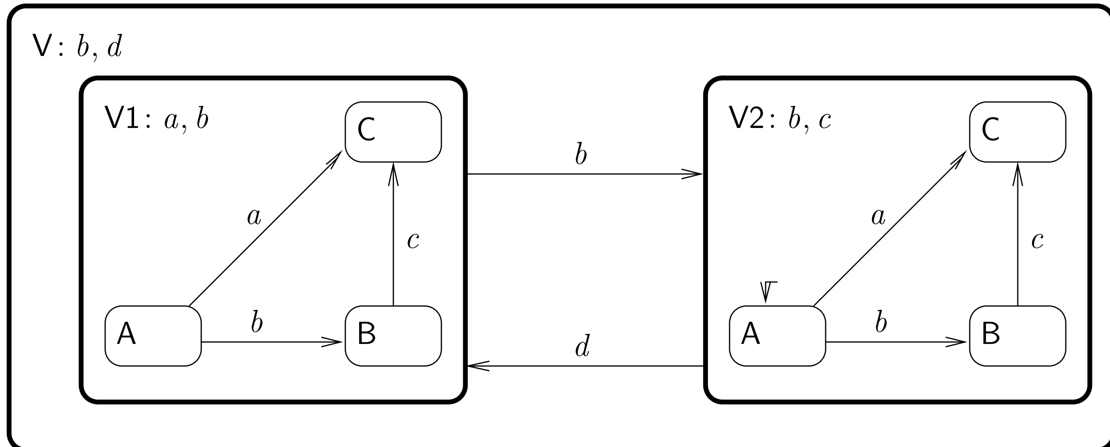

Figure 3.3: An OR composition of views.

there is no nondeterministic choice between {V1.B} and {V2.A});

- no other event can change the system configuration.

Assuming that the system is in configuration {V2.A},

- if the view V2 or V triggers b, then the next configuration will be {V2.B};
- if the view V triggers d, then the next configuration nondeterministically will be {V1.A}, {V1.B}, or {V1.C} (because of the encapsulation mechanism described above and the fact that there is no default state in V1);
- no other event can change the system configuration.

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. An AND composition

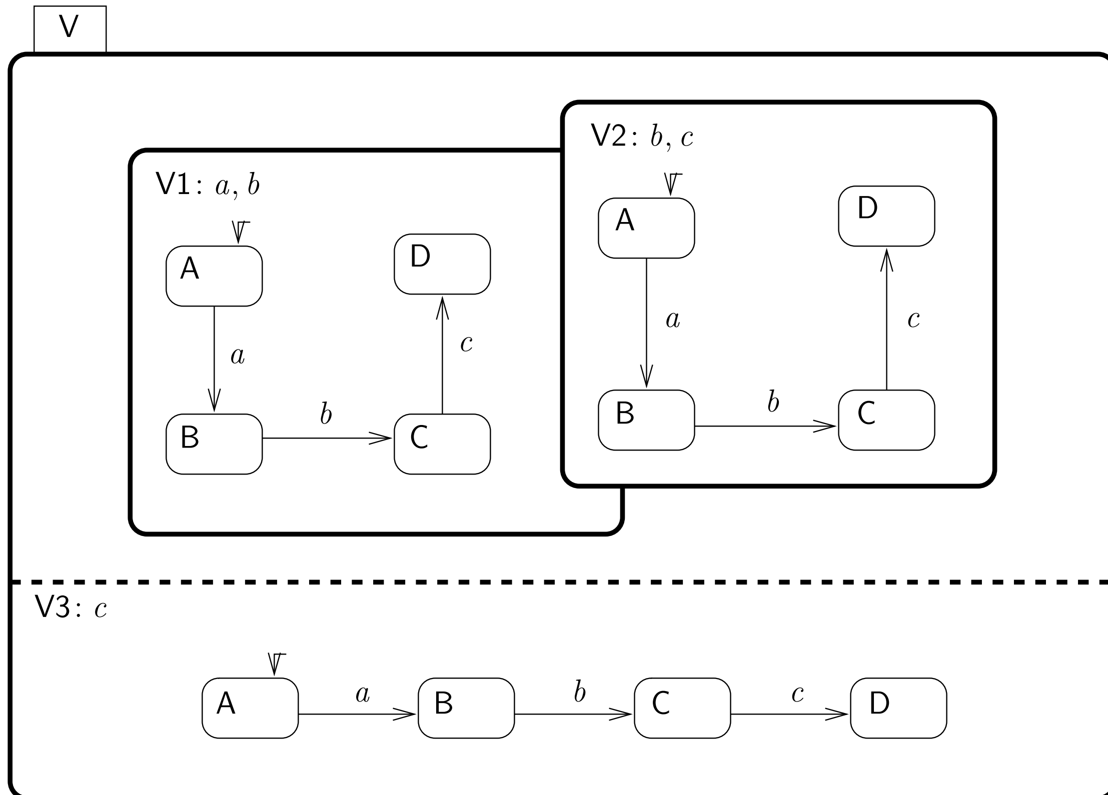

Figure 3.4: An AND composition of views.

of views forms a superview called AND-view. The viewchart of Figure 3.4 shows an AND composition of views.

The scopes of events and actions owned by V3, in this figure, are extended to V1 and V2, while the scopes of events and actions owned by V1 or V2 are extended only to V3. The following examples show the effect of the composition on some transitions. Assuming that the system is in configuration $\{\mathsf{V1.A}, \mathsf{V2.A}, \mathsf{V3.A}\}$,

- if the view V1 triggers a, then the next configuration will be $\{\mathsf{V1.B}, \mathsf{V2.A}, \mathsf{V3.B}\}$;
- no other view can trigger a, because they do not own a.

Assuming that the system is in configuration $\{\mathsf{V1.B}, \mathsf{V2.B}, \mathsf{V3.B}\}$,

- if the view V1 triggers b, then the next configuration will be {V1.C, V2.B, V3.C};
- if the view V2 triggers b, then the next configuration will be {V1.B, V2.C, V3.C};
- V3 cannot trigger b, because it does not own b.

Assuming that the system is in configuration {V1.C, V2.C, V3.C},

- if the view V2 triggers c, then the next configuration will be {V1.C, V2.D, V3.D};
- if the view V3 triggers c, then the next configuration will be {V1.D, V2.D, V3.D};
- V1 cannot trigger c, because it does not own c.

3.2.4 A General Example

The viewchart of Figure 3.5 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.

The following examples show the effect of the composition on some transitions. A possible configuration of the system described in Figure 3.5 is {V3.V5.A, V3.V6.B, V3.V7.B, V4.V5.B, V4.V6.C, V4.V7.A}. 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, then the entire system configuration will nondeterministically change to either {V1.A} or {V1.B} (because no state of V1 is marked

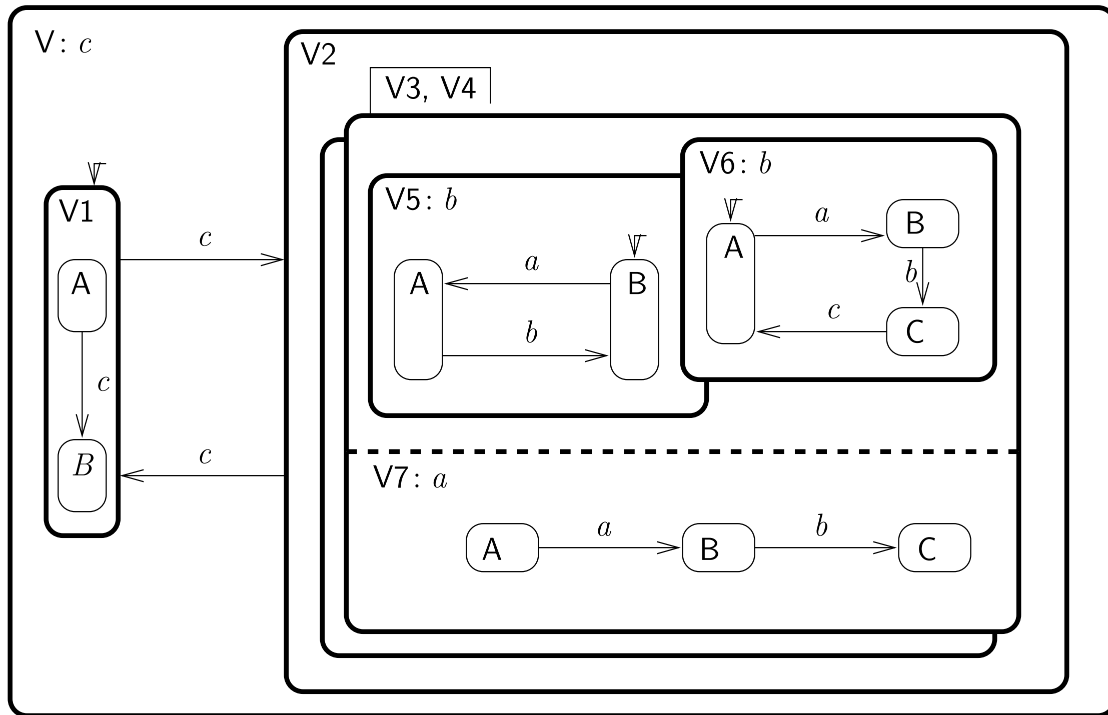

Figure 3.5: Composition of views in a viewchart.

as the default state; note that the configuration will be {V1.B} if the underlying Statecharts semantics allows *instantaneous states* described in Section 2.3.2, Page 19);

- 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, then the entire system configuration will nondeterministically 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, then the entire system configuration will nondeterministically change to either {V1.A} or {V1.B} (notice the priority of transition from V2 to V1 over V3.V6.C to V3.V6.A);
- no other event can change the sub-configuration.

3.2.5 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. The history transitions are not allowed in higher level views of a viewchart. A history transition in a higher level view can result in a transition that crosses view boundaries. A transition that crosses view boundaries violates the independence of views and, therefore, is not allowed in Viewcharts.

3.2.6 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 within 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 specified retrieve activity is completed. (See Section 5.1, page 63.)

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 [31]. 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.

3.3 Conflicts

Recall that the behavioral requirements specification of a system, in Viewcharts, is expressed as compositions of views, where a view specifies the behavior of the system observable from a specific point of view. In other words, Viewcharts composes specifications. An inevitable issue in composing specifications is the possibility of conflicts between the specifications. This issue, in a viewchart, is the possibility of having conflicting views. Viewcharts minimizes this possibility; consider the following points:

- 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;

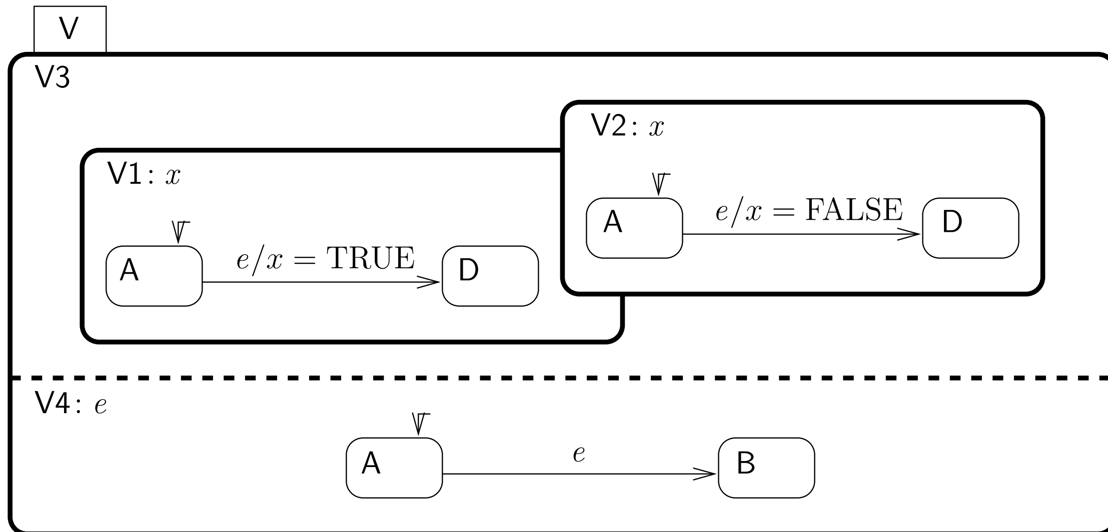

Figure 3.6: Consistent views in a viewchart.

doing so can result in conflicting views.

- In an AND composition, the scope of an element covers all the composed views. Therefore, ANDing views can result in conflict.

- In a SEPARATE or OR composition, the composed views are independent of each other; an element is local to the view in which it occurs. Therefore, except for the effect of the first item, there cannot be any conflict between the views.

Figure 3.6 shows an example of consistent views. If the event e is triggered by the view V4, while the system is in configuration $\{\mathsf{V1.A}, \mathsf{V2.A}, \mathsf{V4.A}\}$ then the new configuration is $\{\mathsf{V1.B}, \mathsf{V2.B}, \mathsf{V4.B}\}$ and the variable x is TRUE in the view V1 and FALSE in the view V2. Both V1 and V2 own x; therefore, in regard to the variable x, as described in Chapter 4, Viewcharts recognizes two distinct variables: V1.x and V2.x; the first one is TRUE and the second one is FALSE. Consequently, the views are consistent.

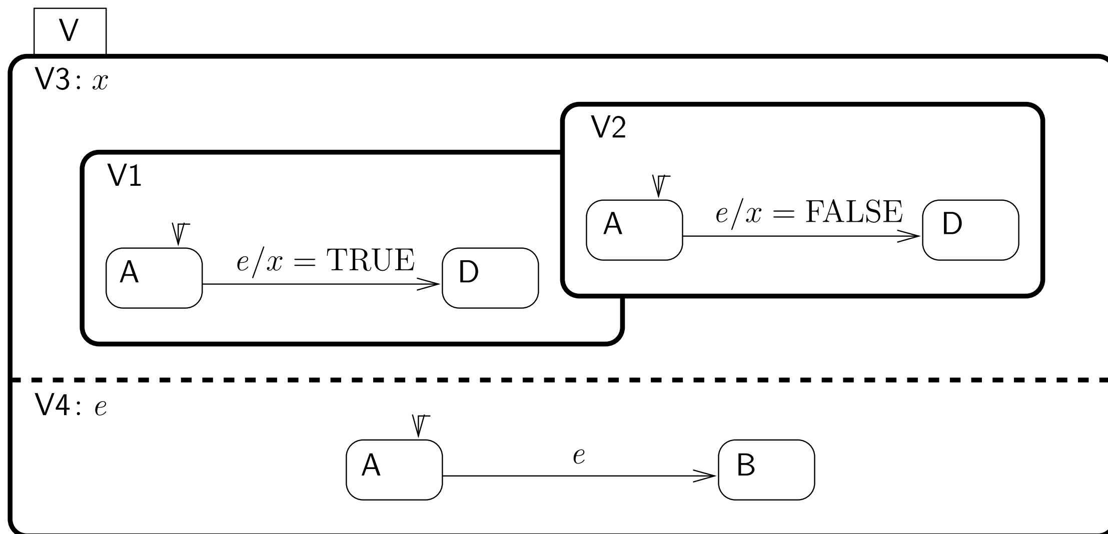

Figure 3.7: Conflicting views in a viewchart.

In the viewchart of Figure 3.7, however, neither V1 nor V2 owns the variable x. It is owned by V3; therefore, the views V1 and V2 attempt to set the value of the same variable V3.x, at the same time to conflicting values. Consequently, the views are inconsistent.

Note that where there is a possibility of conflicts between views (e.g., ANDing views or using global variables, as described above), the specifier must be aware of this possibility and be responsible for the consequences. In addition, there are arguments that structuring specifications into views generally makes it easier to deal with conflicts than if the view specifications were intertwined from the start [46]. Nonetheless, the possibility of conflicts between views does exist in Viewcharts and discussed further, as a topic of future research, in Section 6.4, Page 87.

Chapter 4

Formal Definitions and Semantics

Supporting Item 1 of the necessary characteristics of the solution, this chapter establishes a semantic basis for Viewcharts via translation to Statecharts. An algorithmic semantics of the Viewcharts notation is provided in [**?**]; here I will present a set theoretic-based approach for the definitions and semantics for the notation.

Informally, 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 [66, 31]. (See Section 2.3.2.) 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.

One way to provide a semantic basis for Viewcharts, is to show that:

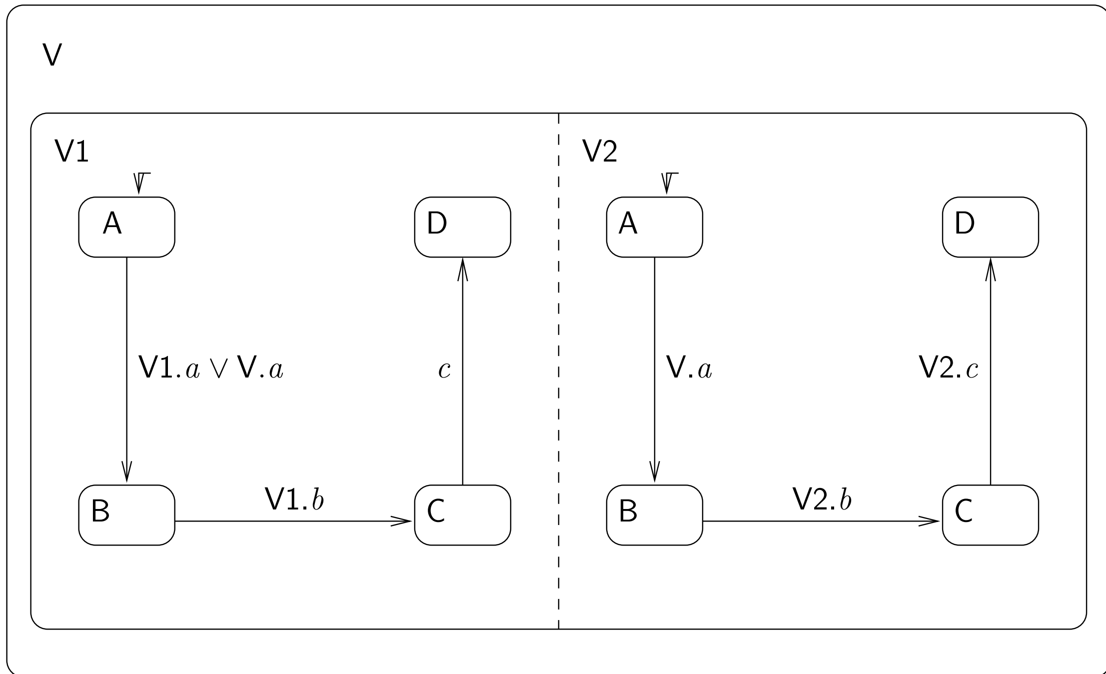

Figure 4.1: A Statecharts translation of the viewchart shown in Figure 3.2.

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

Examples of such translations are shown in Figure 4.1, 4.2, and 4.3, which are the Statecharts translations of the viewcharts shown in Figure 3.2, 3.4, and 3.5 respectively. Notice that the event c in V1 of Figure 4.1 and the event a in V2 of Figure 4.2 cannot occur and the corresponding transitions cannot take place.

Views, in Viewcharts, are similar to states; and states, in Statecharts, are uniquely identified. Simply 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

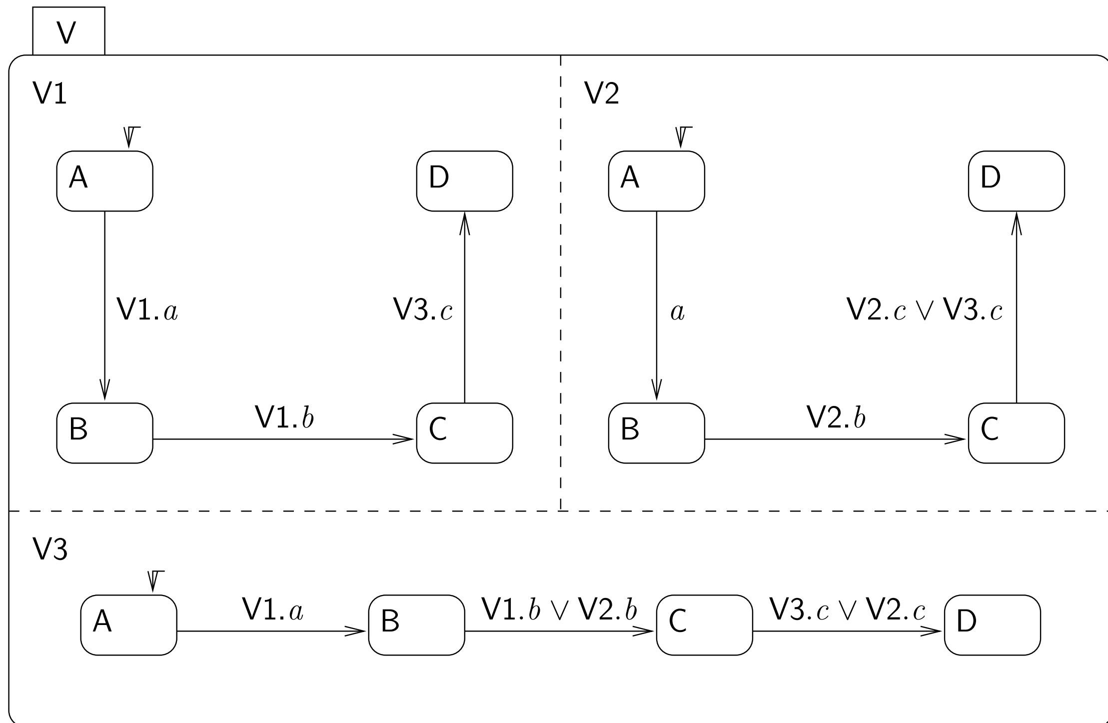

Figure 4.2: A Statecharts translation of the viewchart shown in Figure 3.4.

the behavior description, the elements must be renamed 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 compositions.

- Transform views to states.

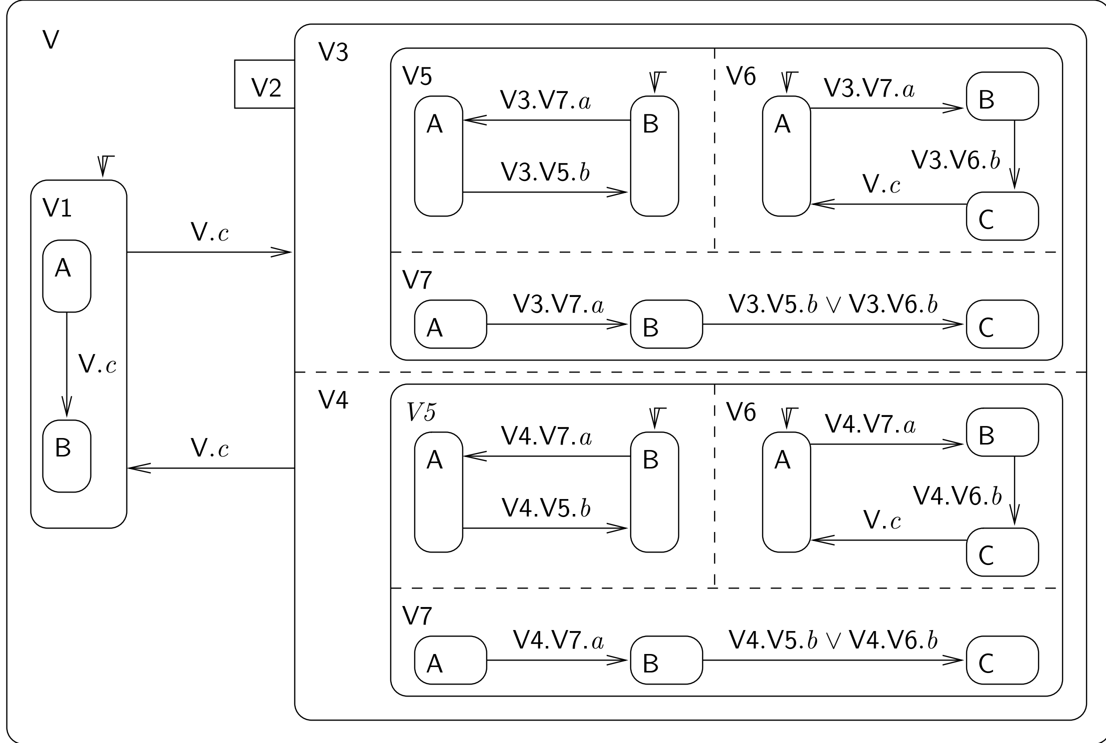

Figure 4.3: A Statecharts translation of the viewchart shown in Figure 3.5.

4.1 Primitives

We will use the following primitives from Statecharts. (Detailed definitions of these primitives are given by Harel [33].)

- S, a set of *states*.
- H, a set of *history symbols*.
- V, a set of *variables*.
- $\tilde{E}$, a set of *simple events and actions*.
- $\bar{E} \subseteq \{e | e \in \mathbf{seq}(\Omega_1)\} \cup \{< e > | e \in \tilde{E}\}$, a set of *basic events and actions*, where $\Omega_1 \supseteq S \cup V$. The notation $< e >$ represents a singleton sequence. Note

that all we are specifying here is that an element in $\bar{E}$, is either an element of $\tilde{E}$ or an event or action made of an expression in which elements from $S \cup V$ can occur; the actual definition of the expression is left to the Statecharts notation [33].

- $E \subseteq \mathbf{seq}(\Omega)$, a set of *events and actions*, where $\Omega \supseteq \bar{E}$. These are compound events; again, their actual definitions are left to Statecharts [33].
- $L \subseteq E \times E$, a set of labels, where for $\ell = (e, a) \in L$, e is called an event and a is called an action.
- *Statechart* and the terms used in its definition (Equation 4.3).

4.2 Definitions

A viewchart is defined as

$$w = (U, E, V, T, D, o, \tau, \rho) \qquad \text{where} \tag{4.1}$$

- U is a set of views,
- E and V are the primitives mentioned above,
- T is a set of transitions,
- $D \subseteq U$ is a set of default views,
- $o : U \rightarrow 2^{\bar{E} \cup V}$ is an *ownership function*,
- $\tau : U \rightarrow \{\text{AND}, \text{OR}, \text{SEP}\}$ is a *type function*, and
- $\rho : U \rightarrow 2^U$ is a *hierarchy function*.

4.2.1 Hierarchy of Views

Restriction: $\exists_1 r \in U, \forall u \in U \cdot r \notin \rho(u)$

r is called the *root* of the viewchart.

Restriction:

$$\forall x \in U, \forall y \in U \cdot \rho(x) \cap \rho(y) \neq \emptyset \Rightarrow x = y \tag{4.2}$$

These two restrictions give a tree structure for the hierarchy of views.

Notice that U is a set; i.e., $\forall u \in U$, u must be specified by a uniquely identifiable name, which may be a fully or partially qualified name (as described in Section 3.2.1).

Subviews: Extend ρ to ρ^+ and ρ^* by defining

$$\rho^0(u) = \{u\} \quad \text{and} \quad \rho^{i+1}(u) = \bigcup_{x \in \rho^i(u)} \rho(x)$$

$$\rho^+(u) = \bigcup_{i \geq 1} \rho^i(u)$$

$$\rho^*(u) = \bigcup_{i \geq 0} \rho^i(u)$$

$\rho(u)$ is the set of *immediate subviews* of u; $\rho^+(u)$ is the set of *all subviews* of u; and $\rho^*(u)$ is $\{u\} \cup \rho^+(u)$.

Superviews: For $u \in U$, define

$$\sigma(u) = \{x | u \in \rho(x)\}$$

$$\sigma^+(u) = \{x | u \in \rho^*(x) \wedge x \neq u\}$$

$$\sigma^*(u) = \{x | u \in \rho^*(x)\}$$

Restriction: $\forall u \in U \cdot \#\rho(u) \geq 2 \vee \rho(u) = \emptyset$

4.2.2 Basic Views

Define $\bar{U} = \{x | x \in U \wedge \rho(x) = \emptyset\}$

$\bar{U}$ is called the set of *basic views*; they are the leaves of the hierarchy. Note that based on our definitions, so far, a view is just an element of a set U. We now make a restriction on the basic views to specify what they are. Every other view u is then recursively said to be $\tau(u)$ composition of its immediate subviews $\rho(u)$.

Restriction: A basic view is a statechart.

Formally, $u \in \bar{U}$ is defined as:

$$u = (S_u, H_u, E_u, V_u, T_u, D_u, \tau_u, \rho_u) \qquad \text{where} \tag{4.3}$$

- $S_u, H_u, E_u \subseteq E, V_u \subseteq V$ are sets of states, history symbols, events and actions, and variables, respectively. $\forall u \in U \setminus \bar{U}$, define $S_u = \rho(u)$.

  Define/Generalize

  $$S = \bigcup_{u \in U} S_u \quad \text{and} \quad H = \bigcup_{u \in \bar{U}} H_u \tag{4.4}$$

  S_u is the set of immediate subviews of u, S is the set of all the states and views in the viewchart, and H is the set of all the history symbols in the viewchart.

- $T_u \subseteq S_u \times L \times S_u \cup H_u$, is a set of *transitions*.

- $D_u \subseteq S_u \cup H_u$ is a set of *default states and history symbols*. Related to D_u, we will also use another primitive from Statecharts: Π_u^0, the *initial configuration* of u.

- $\tau_u : S_u \to \{\text{AND}, \text{OR}\}$ is a *type function*; $\tau_u(s) = \text{AND}$ if s is an AND-state and $\tau_u(s) = \text{OR}$ if s is an OR-state.

- $\rho_u : S_u \rightarrow 2^{S_u}$ is a *hierarchy function*; $\rho_u(s)$ is the set of immediate substates of s; $\rho_u^+(s)$ is the set of all substates of s; and $\rho_u^*(s) = \rho_u^+(s) \cup \{s\}$.

4.2.3 Transitions

Let us generalize the definition of transitions in Statecharts for the entire Viewchart: $T \subseteq S \times L \times S \cup H$ is a set of *transitions*.

For $u \in U \setminus \bar{U}$, define

$$T_u = \{(x, \ell, y) | (x, \ell, y) \in T \wedge x \in \rho(u) \wedge y \in \rho(u)\}$$

$$E_u = \{z | \exists (x, (e, a), y) \in T_u \cdot z \in \mathbf{ran}(e \sim a \rhd \bar{E})\}$$

The operator $\sim$ concatenates two sequences, $\rhd$ is the range restriction operator, and the function **ran** gives the range of a sequence: $\mathbf{ran}(e \sim a \rhd \bar{E})$ is the set of elements from $\bar{E}$ that occur in e or a.

$$V_u = \{v | \exists (x, (e, a), y) \in T_u \cdot \exists z \in \mathbf{ran}(e \sim a \rhd \bar{E}) \cdot v \in \mathbf{ran}(z \rhd V)\}$$

T_u is the set of transition occurrences in u, E_u is the set of basic event and action occurrences in u, and V_u is the set of variable occurrences in u.

Restriction:

$$\bigcup_{u \in U} E_u = \bar{E} \quad \text{and} \quad \bigcup_{u \in U} V_u = V \quad \text{and} \quad \bigcup_{u \in U} T_u = T \tag{4.5}$$

Two implications of the above restriction on T and Equation 4.4 are: *no transition is allowed to cross view boundaries* and *history symbols can occur only in the base views*. Note also that a history symbol occuring in a higher level view results in a transition that crosses view boundaries, which is not allowed.

4.2.4 Composing Views

ANDed **Views:** For $u \in U$, define

$$\mathbf{and}(u) = \{x | \exists y \in U \cdot u \in \rho(y) \wedge x \in \rho(y) \wedge \tau(y) = \text{AND} \wedge x \neq u\}$$

Restriction: No transition is allowed between ANDed views:

$$\exists u \in U \setminus \bar{U} \cdot \tau(u) = \text{AND} \Rightarrow T_u = \emptyset$$

This stops transitions between immediate subviews of an AND-view. Recall also the restriction on T of Equation 4.5 that no transition is allowed to cross view boundary.

ORed **Views:** For $u \in U$, define

$$\mathbf{or}(u) = \{x | \exists y \in U \cdot u \in \rho(y) \wedge x \in \rho(y) \wedge \tau(y) = \text{OR} \wedge x \neq u\}$$

Restriction: In an OR composition of views, no subview or state in one view can be referenced by another one.

$$\begin{aligned} &\exists u \in U \setminus \bar{U} \cdot \tau(u) = \text{OR} \Rightarrow \\ &\quad \forall x \in \rho(u), \forall y \in \rho(u), \forall (i,(e,a),j) \in \bigcup_{z \in \rho^*(y)} T_z, \forall z \in \mathbf{ran}(e \sim a \rhd \bar{E}) \cdot \\ &\qquad \mathbf{ran}(z \rhd \bigcup_{k \in \rho^*(x)} S_k) = \emptyset \vee x = y \end{aligned}$$

SEPARATEd **Views:** For $u \in U$, define

$$\mathbf{sep}(u) = \{x | \exists y \in U \cdot u \in \rho(y) \wedge x \in \rho(y) \wedge \tau(y) = \text{SEP} \wedge x \neq u\}$$

Restriction: In a SEPARATE composition of views, no transition is allowed between the views and no subview or state in one view can be referenced by another one.

$$\exists u \in U \setminus \bar{U} \cdot \tau(u) = \text{SEP} \Rightarrow T_u = \emptyset$$

$$\exists u \in U \setminus \bar{U} \cdot \tau(u) = \text{SEP} \Rightarrow$$
$$\forall x \in \rho(u), \forall y \in \rho(u), \forall (i,(e,a),j) \in \bigcup_{z \in \rho^*(y)} T_z, \forall z \in \mathbf{ran}(e \sim a \rhd \bar{E}) \cdot$$
$$\mathbf{ran}(z \rhd \bigcup_{k \in \rho^*(x)} S_k) = \emptyset \vee x = y$$

4.2.5 Ownership and Scoping

For $u \in U$ and $l \in o(u)$, define

$$\psi(l,u) = \begin{cases} \{x | x \in \rho^*(u) \vee \exists y \in \mathbf{and}(u) \cdot x \in \rho^*(y)\} & \text{if} \quad l \in \bar{E} \\ \{x | x \in \rho^*(u) \vee \exists y \in \mathbf{and}(u) \cdot x \in \rho^*(y)\} \setminus & \\ \quad (\{x | \exists z \in \rho^+(u) \cdot l \in o(z) \wedge x \in \rho^*(z)\} \cup & \\ \quad \{x | \exists y \in \mathbf{and}(u) \cdot \exists z \in \rho^*(y) \cdot l \in o(z) \wedge x \in \rho^*(z)\}) & \text{if} \quad l \in V \end{cases}$$

$\psi(l,u)$ is called the *scope* of $u.l$ (the scope of element l with respect to u).

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

$$\theta(l,u) = \{x | u \in \psi(l,x)\}$$

Restriction:

$$\#\theta(l,u) \begin{cases} = 0 & \text{iff} \quad \exists x \in U \cdot \exists y \in o(x) \cdot u \in \psi(y,x) \wedge l = x.y \\ \begin{cases} \geq 1 & \text{if} \quad l \in \bar{E} \\ = 1 & \text{if} \quad l \in V \end{cases} & \text{otherwise} \end{cases} \tag{4.6}$$

Informally we say, $\theta(l,u)$, for $l \in \bar{E}$, is the set of views that *can trigger* the occurrences of event l in u and, for $l \in V$, is a set whose only element is the owner of the occurrences of variable l in u. The restriction says that each occurrence of an element either is in the form of a qualified name or has an owner which, in the case of an event, can trigger it.

4.2.6 Initial Configuration

The *initial configuration* of u for $u \in \bar{U}$ is Π_u^0, the initial configuration of the statechart u.

The *initial configuration* of u for $u \in U \setminus \bar{U}$ is defined as

$$\Pi_u^0 = \begin{cases} \rho(u) \cup \bigcup_{x \in \rho(u)} \Pi_x^0 & \text{if} \quad \tau(u) = \text{AND}|\text{SEP} \\ \{x\} \cup \Pi_x^0 \quad \text{where} \quad \exists_1 x \in \rho(u) \cdot x \in D & \text{if} \quad \tau(u) = \text{OR} \end{cases}$$

The *initial configuration* of w is Π_w^0.

For $s \in S$, define

$$\rho'(s) = \begin{cases} \rho_u(s) & \text{if} \quad \exists u \in \bar{U} \cdot s \in S_u \\ \rho(s) & \text{otherwise} \end{cases}$$

Extend ρ' to ρ'^* and ρ'^+, similar to the corresponding extensions of ρ.

Restriction: Generalize the restriction 4.2 as follows:

$$\forall x \in S, \forall y \in S \cdot \rho'(x) \cap \rho'(y) \neq \emptyset \Rightarrow x = y \tag{4.7}$$

This extends the tree structure of the views to continue, at the basic views, to the corresponding states.

Basic States: $\bar{S} = \{s | s \in S \wedge \rho'(s) = \emptyset\}$ is called the set of *basic states*.

Initial Basic Configuration: $\bar{\Pi}_w^0 = \Pi_w^0 \cap \bar{S}$ is called the *initial basic configuration* of the viewchart.

4.3 Viewcharts Equivalence to Statecharts

For a given viewchart w, defined by Equation 4.1, we now construct a statechart w' as follows:

$$w' = (S, H, E', V', T', D', \tau', \rho') \qquad \text{where} \tag{4.8}$$

S, H, and ρ' are already defined.

For $u \in U$, $v \in V_u$, define

$$\mu_v(v,u) = \begin{cases} v & \text{if} \quad \theta(v,u) = \emptyset \\ x.v \quad \text{where} \quad x \in \theta(v,u) & \text{otherwise} \end{cases}$$

Note that $x.v$ in the definition above is a simple pairing, (x, v), and similarly $x.e$ in the definition below.

For $u \in U$, $e \in E_u$, define

$$\mu_e(e,u) = \begin{cases} e[\mu_v(v,u)/v] & \text{if} \quad \theta(e,u) = \emptyset \\ \bigvee_{x \in \theta(e,u)} x.e[\mu_v(v,u)/v] & \text{otherwise} \end{cases} \quad \text{(See also Equation 4.6.)}$$

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

$$\eta((x,(e,a),y)) = (x, e[\mu_e(\bar{e},u)/\bar{e}], a[\mu_e(\bar{a},u)/\bar{a}]), y) \quad \text{where}$$

$$\exists_1 u \in U \cdot t \in T_u \wedge \bar{e} \in E_u \wedge \bar{a} \in E_u$$

$$V' = \{\mu_v(v,u) | u \in U, v \in V_u\}$$

$$T' = \{\eta(t) | t \in T\}$$

$$E' = \{e' | \exists (x,(e,a),y) \in T' \cdot e' = e \vee e' = a\}$$

$$\bar{E}' = \{\mu_e(e,u) | u \in U, e \in E_u\}$$

$$D' = D \cup \bigcup_{u \in \bar{U}} D_u$$

$$\tau'(s) = \begin{cases} \tau_u(s) & \text{if} \quad \exists u \in \bar{U} \cdot s \in S_u \\ \text{AND} & \text{if} \quad s \in U \setminus \bar{U} \wedge \tau(s) = \text{SEP} \\ \tau(s) & \text{otherwise} \end{cases}$$

Consider the following facts which are the immediate conclusions of the above definitions:

- The set of states/views in the viewchart and the set of states in the statechart are defined by the same set S. The hierarchy functions for both viewchart and statechart are also defined by the same function ρ'. Therefore, both viewchart and statechart have the same structure.

- Our definition of Π^0_w is similar to that of Statecharts; therefore, the initial configurations of both w and w' are also the same: $\Pi^0_w = \Pi^0_{w'}$. (Or, we could have just defined $\Pi^0_w = \Pi^0_{w'}$.)

- For every transition in the viewchart, there is a corresponding transition in the statechart and vice versa; i.e., $\eta : T \longrightarrow T'$ is a bijective function:

$$\forall t \in T \cdot \eta(t) \in T' \wedge \forall t' \in T' \cdot \eta^{-1}(t') \in T$$

Both the viewchart w and the statechart w' start from the same initial configuration. We now restrict the viewchart such that there will be no change in the viewchart unless a transition takes place in the statechart, in which case only the corresponding transition in the viewchart takes place.

Let us define, in Statecharts, an operation $\uparrow e$, which evaluates to either TRUE, if and at the moment that the event e occurs, or FALSE, otherwise. When a transition $t = (x, (e, a), y)$ takes place, we say e enables t and denote it by $e \vdash t$. A basic event $\bar{e} \in \mathbf{ran}(e \rhd \bar{E})$ may also enable the transition, $\bar{e} \vdash t$.

Restriction:

$$e' \vdash t' \iff e \vdash t \quad \text{where} \quad t = (x, (e, a), y) \in T \wedge t' = (x, (e', a'), y) = \eta(t) \tag{4.9}$$

Since the initial configurations of w and w' are the same and the destinations of the transitions t and t' are also the same, the first conclusion of this restriction is that

the next configuration of w' defines the next configuration of w. When the transitions take place, the changes in the configurations of w and w' are the same; therefore, the configuration of w is always the same as the configuration of w'. Furthermore, the set of transitions accepted by the statechart defines the *behavior* of the statechart (to be exact, the behavior of the system represented by the statechart). Similarly, we say the corresponding set of transitions in the viewchart defines the *behavior* of the viewchart (to be exact, again, the behavior of the system represented by the viewchart). Finally, the two machines describe the same behavior.

Now let us see how this restriction defines/restricts the behavior of w: For a transition $t \in T$,

$$\begin{aligned} & e' \vdash t' \Rightarrow e' \vdash (x, (e', a'), y) \Rightarrow \uparrow e' \\ & \Rightarrow \exists \bar{e}' \in \mathbf{ran}(e' \rhd \bar{E}') \cdot \uparrow \bar{e}' \\ & \Rightarrow \exists x \in U, \exists u \in U, \exists \bar{e} \in E_x \cdot t \in T_x \wedge u \in \theta(x, \bar{e}) \wedge \uparrow u.\bar{e} \end{aligned} \tag{4.10}$$

Recall the definitions of θ, μ_e, and Equations 4.5 and 4.6 to verify Equation 4.10.

> **When the event $u.\bar{e}$ occurs in the statechart w', we say the view u *triggers* the event $\bar{e}$ in the corresponding viewchart w and denote it by $u \uparrow \bar{e}$.**

Informally, Equation 4.10 says, when a view triggers an event, $u \uparrow \bar{e}$, only the transitions within the views covered by the scope of event $\bar{e}$ with respect to u can be affected. (See also the definition of ψ.) We can also verify, based on the definition of μ_e, that the restriction 4.9 holds with this behavior:

$$\begin{aligned} & \exists u \in U, \exists \bar{e} \in E_u \cdot u \uparrow \bar{e} \\ & \Rightarrow \forall x \in \psi(\bar{e}, u), \forall t \in T_x \cdot (\bar{e} \vdash t \iff \mu_e(\bar{e}, x) \vdash \eta(t)) \end{aligned} \tag{4.11}$$

Considering that $\bar{e}' = \mu_e(\bar{e}, x)$ and $t' = \eta(t)$,

$$\bar{e} \vdash t \iff \bar{e}' \vdash t' \tag{4.12}$$

As a final point, note that a SEPARATE, AND, or OR composition of some views is also a view. Consider a viewchart w, where $\tau(w) = \mathrm{SEP}$ and $\rho(w) \neq \emptyset$. The root view w is, in fact, resulted from a SEPARATE composition of the set of views $\rho(w)$. Similarly, if $\tau(w) = \mathrm{AND}$ (or $\tau(w) = \mathrm{OR}$), then the set of views $\rho(w)$ are ANDed (or ORed) to form the view w.

Chapter 5

Examples

Supporting Item 5-6 of the necessary and desirable characteristics of the solution, this chapter presents some examples demonstrating the way in which the behavioral requirements of a software system can be specified as a composition of its behavioral views. The examples are also intended to show that Viewcharts can be used for specifying and composing software behavioral views.

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

5.1 Manufacturing Control System

This section presents a Viewcharts specification of a Manufacturing Control System (MCS). An informal, but detailed, description of a similar system is given by Dunietz

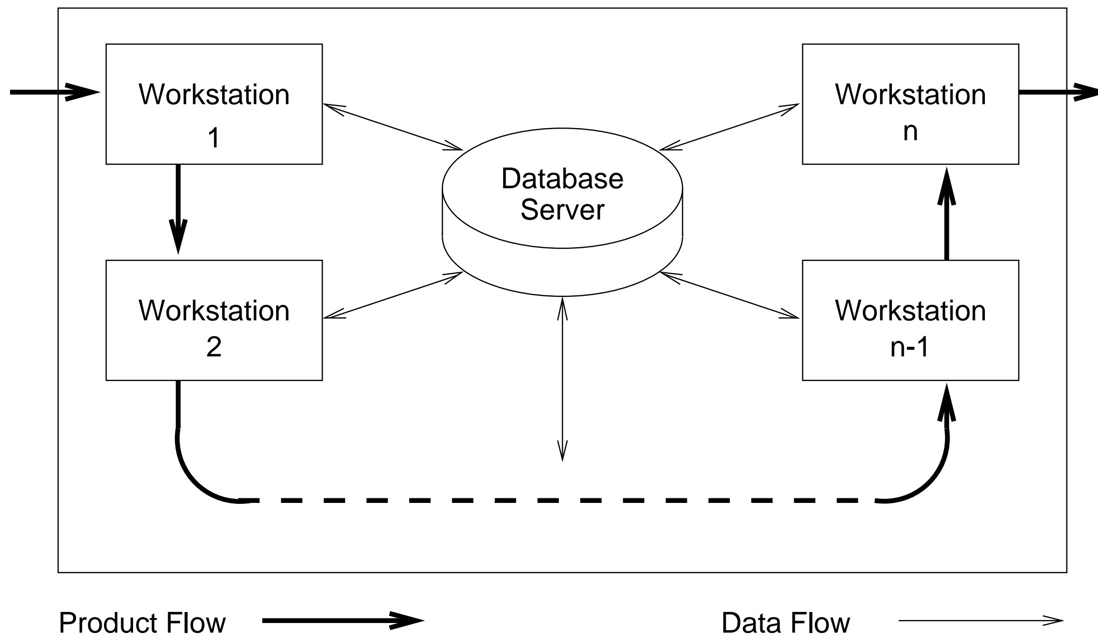

Figure 5.1: Product and information flow in a manufacturing line.

and others [17].

Consider a “flexible”[1] and “just-in-time”[2] manufacturing shop. It consists of a number of workstations, where each workstation performs a certain process on the product. Figure 5.1 shows an informal diagram representing the flow of product and information in the shop. Our objective is to specify the behavioral requirements of a MCS for this shop.

Central to the system is a database server (DBS) which maintains and supplies the information requirements of the workstations. At the beginning of the manufacturing line, the first workstation associates each product with a unique identification

[1]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.

[2]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.

number/string *pid*, which must be communicated to DBS to create a record for the corresponding product. From there on, each product is identified and tracked by the associated *pid*. When a product arrives at a workstation, the *pid* is scanned and communicated to DBS which, in turn, informs the workstation of the process that must be performed on the product. The workstation then proceeds with the process and when it is completed, informs DBS to update the product record.

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

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

5.1.1 Specifying Behavioral Views

The viewchart WS_1, shown in Figure 5.2, 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 "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

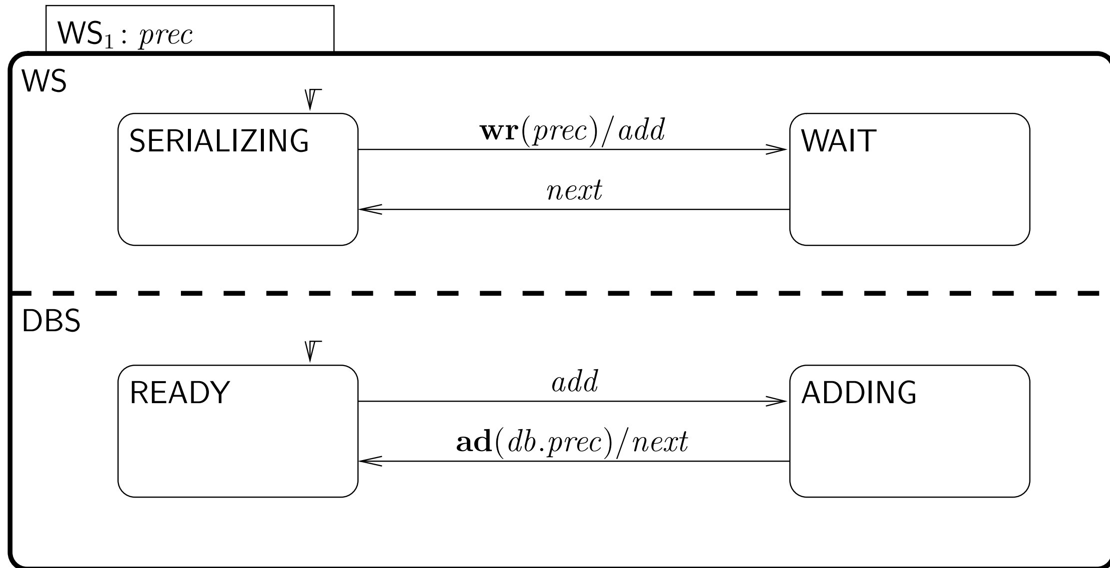

Figure 5.2: A viewchart for the Serialization Workstation.

and written to the compound variable *prec*. When the writing is completed, the event $\mathbf{wr}(prec)$, which is an abbreviation for $\mathbf{written}(prec)$, occurs which in turn generates *add*. This, in turn, takes DBS to the state of ADDING. DBS in this state adds *prec* to the database and when it is done the event $\mathbf{ad}(db.prec)$ occurs, generating the action *next* and taking both DBS and WS back to their starting states.

The other workstations, at the abstraction level of this specification, have identical behaviors. Therefore, a SEPARATE composition of $n-1$ behavioral views, as shown in Figure 5.3, can specify the behavior of the system observable at these workstations. Each view, 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, therefore, it is global to

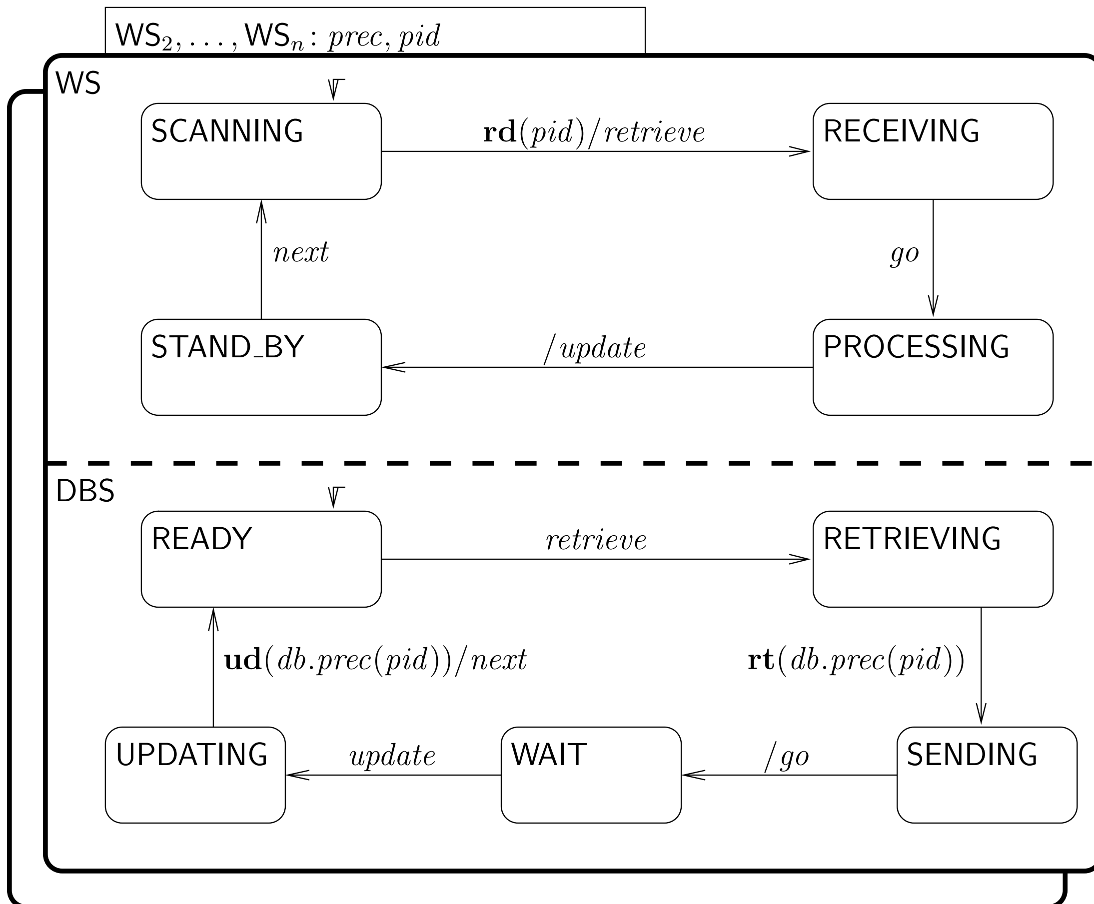

Figure 5.3: The workstations' views of the system.

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 **rd**(*pid*), which is an abbreviation for **read**(*pid*), occurs at a point in time when a *pid* is read (scanned). When WS is in the state of RECEIVING, it expects instructions from DBS. On the other hand, DBS retrieves the product record and based on the history and type of the product sends out the appropriate instructions[3]

[3]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."

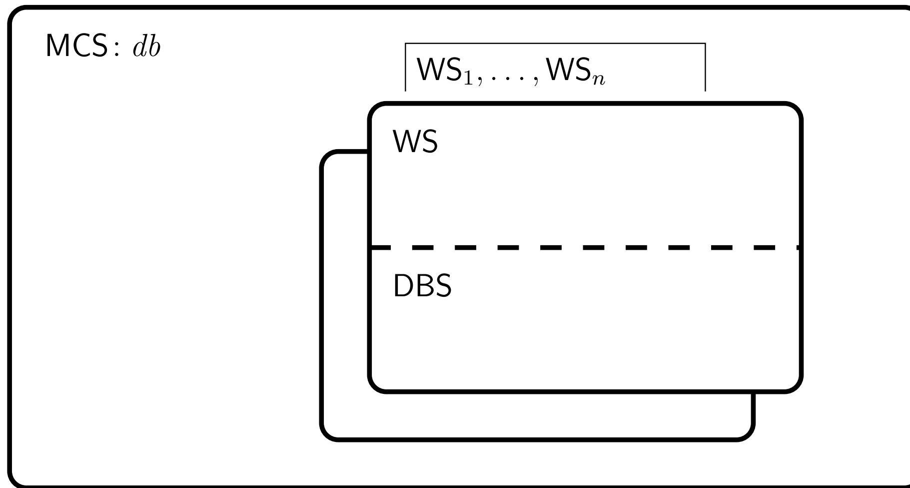

Figure 5.4: A viewchart for a manufacturing line.

regarding the process to be performed on the product.

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

Figure 5.3 should now be self-explanatory.

5.1.2 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 5.4 shows a SEPARATE composition of n views where each view describes the behavior of the

- "Perform a different process: transmitting the required program or instructions."
- "Ship" or "Do not ship;" at the shipping station.
- "Transmitting defect information;", at the repair stations.

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

system from a workstation's point of view.

Notice, once again, the declarations "$\mathsf{WS}_1 : prec$" of Figure 5.2, "$\mathsf{WS}_2, \ldots, \mathsf{WS}_n : prec, pid$" of Figure 5.3, and "$\mathsf{MCS}: db$" of Figure 5.4. 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.

5.1.3 Discussion

The Statecharts specification of this example would consist of $n + 1$ orthogonal components: one for each workstation and another one for DBS. If the manufacturing line consists of only a few workstations, then there is no problem; however, the specification becomes complex when the number of workstations increases. Figure 5.5 shows an attempt to specifying MCS in Statecharts. Each orthogonal component, in this figure, is basically the corresponding view of our Viewcharts specification. However, the figure is not a valid statechart; it has complications and issues that must be resolved.

First of all, the example requires a notation to represent the repetition of workstations. In Viewcharts we represented this repetition simply by a SEPARATE composition of views. In Statecharts we need a capability to represent the repetitions in orthogonal components. Statecharts does not have this capability; however Leveson's extension of Statecharts, RSML [53], provides a method for representing the repetitions. Assuming that Statecharts is extended to support this capability, we have to also parameterize the components to distinguish between the elements of one

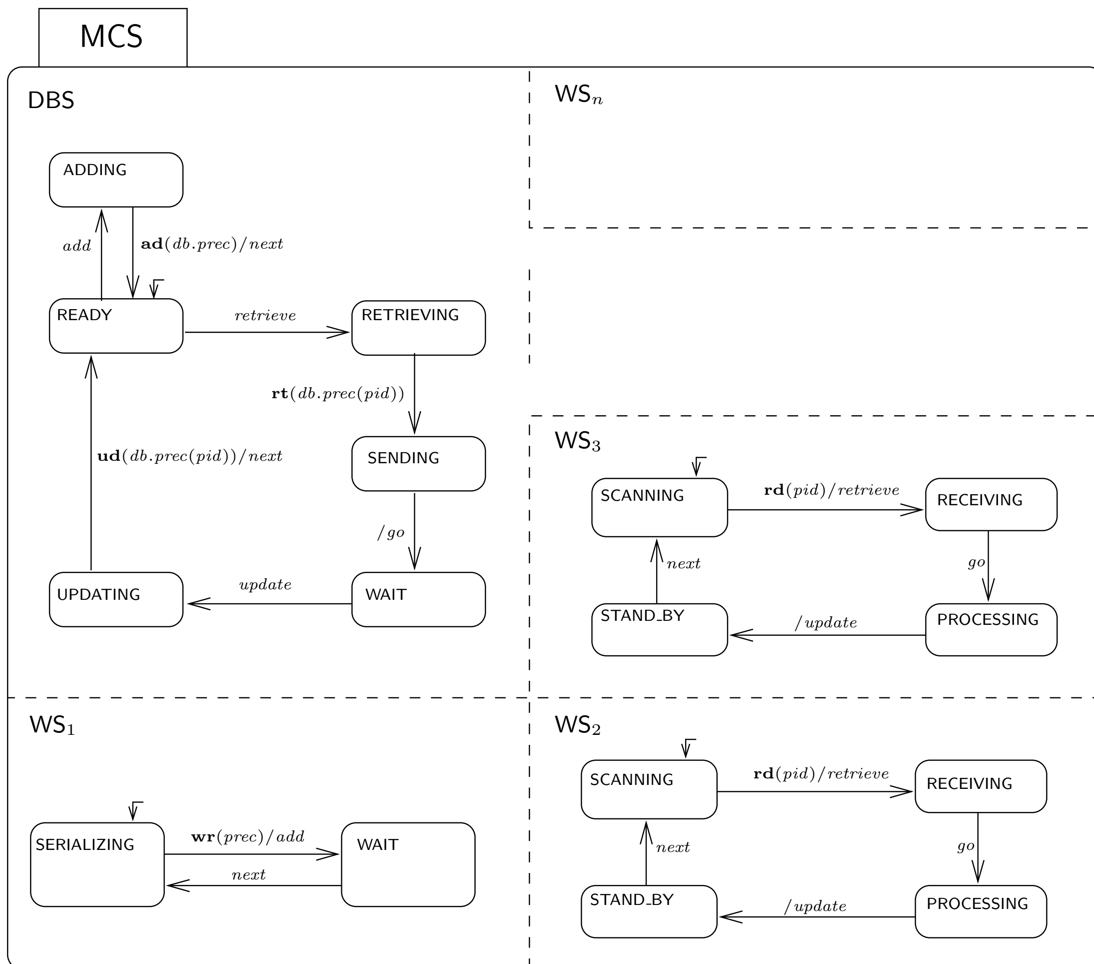

Figure 5.5: An attempt to specifying MCS in Statecharts.

component from those of the others. For example, the Viewcharts description of a workstation, for all workstations but the first one, specifies a local variable *pid*, which is used for passing a product ID from WS to DBS. To provide this specification in the global environment of Statecharts, we have to parameterize this variable to distinguish the product IDs being scanned (concurrently) by different workstations. The following are some of the other complications with the diagram of Figure 5.5:

- The events like **rd**(*pid*), which occurs in different orthogonal components, is

ambiguous. We have to specify the workstation that reads *pid*.

- Different occurrences of a variable like *pid* or *prec* in different orthogonal components are different. We have to replace each variable by an array of variables.

- The actions like *add*, *retrieve*, *update*, *go*, and *next*, which occur in different orthogonal components, are also ambiguous. We have to provide more detail to resolve the ambiguity; e.g., we have to specify which product record to be added, retrieved, or updated or to which workstation the action *next* or *go* is intended for.

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

Suppose that an action to update a product record is generated by a workstation while DBS is in a state of updating another one. DBS must be ready, at all times, to sense events that are indications of database transaction requests. The transactions are time consuming tasks; therefore, DBS must be ready for new events while processing transactions. In Viewcharts, this creates no complexity; because each workstation is interacting with its own view of the database on a one-to-one basis. In Statecharts, however, we have to introduce an additional orthogonal component, a queuing mechanism, which is always ready to sense the events, queue the corresponding transaction requests, and passe them to DBS as it becomes available for accepting requests. An additional orthogonal component does not necessarily complicate the specification. However, this component is a queuing mechanism, which has nothing to do with the

observable behavior of the system, it is there only for making the specification possible (or simple); and later in the design phase, a designer may or may not choose to use the mechanism. The fact that Viewcharts does not require such a mechanism shows the independence of Viewcharts from the design issues, supporting Item 6 of the desirable characteristics of the solution.

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

5.2 Telephone System

This section presents a Viewcharts specification of a simple telephone service provided by a Plain Old Telephone System (POTS). A LOTOS specification of this service is also given by Faci and others [21]. Their informal description of POTS includes the diagrams shown in Figure 5.6 and 5.7. The diagrams are self-explanatory. The timing aspect of POTS, however, is missing from the diagrams. It is also missing from the LOTOS specifications, because timing aspects cannot be specified in LOTOS. As the specifiers state, for example, LOTOS cannot deal with a specification element such as "the telephone can only be off hook for a maximum of 20 seconds, after which it would be disconnected". Our specification, on the other hand, includes the timing aspects of POTS.

The Viewcharts notation is designed to specify the behavioral requirements of

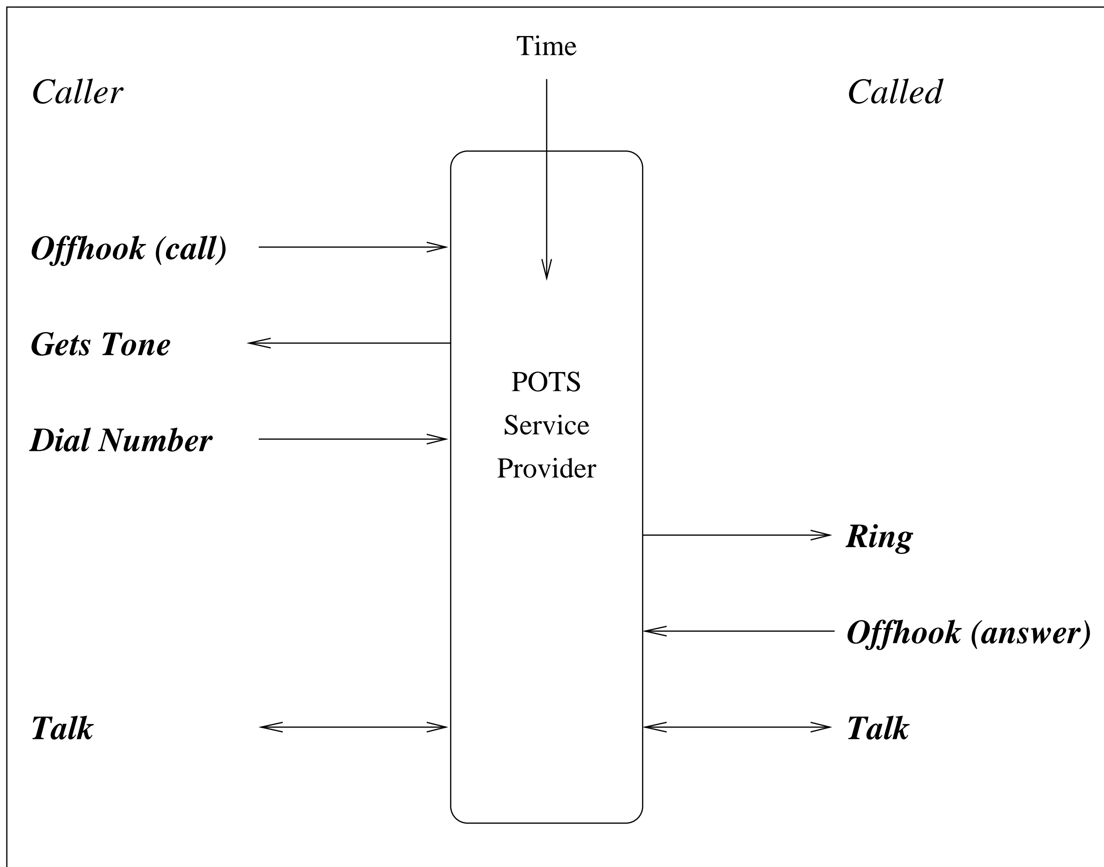

Figure 5.6: A high level scenario for establishing a telephone connection (From [21]).

large-scale complex systems; and we can do it on a *need-to-specify* basis. In Viewcharts, we do not have to specify the full behavior of the system; therefore, we are not concerned with the complexity or scale of the system. A complex system may have many different features; we specify only the features of our interest, i.e., our view of the system.

POTS is a good example to illustrate this approach. We want to specify a simple telephone service provided by POTS. That is only one of the many behavioral views of POTS (and we will still specify it as a composition of even simpler behavioral

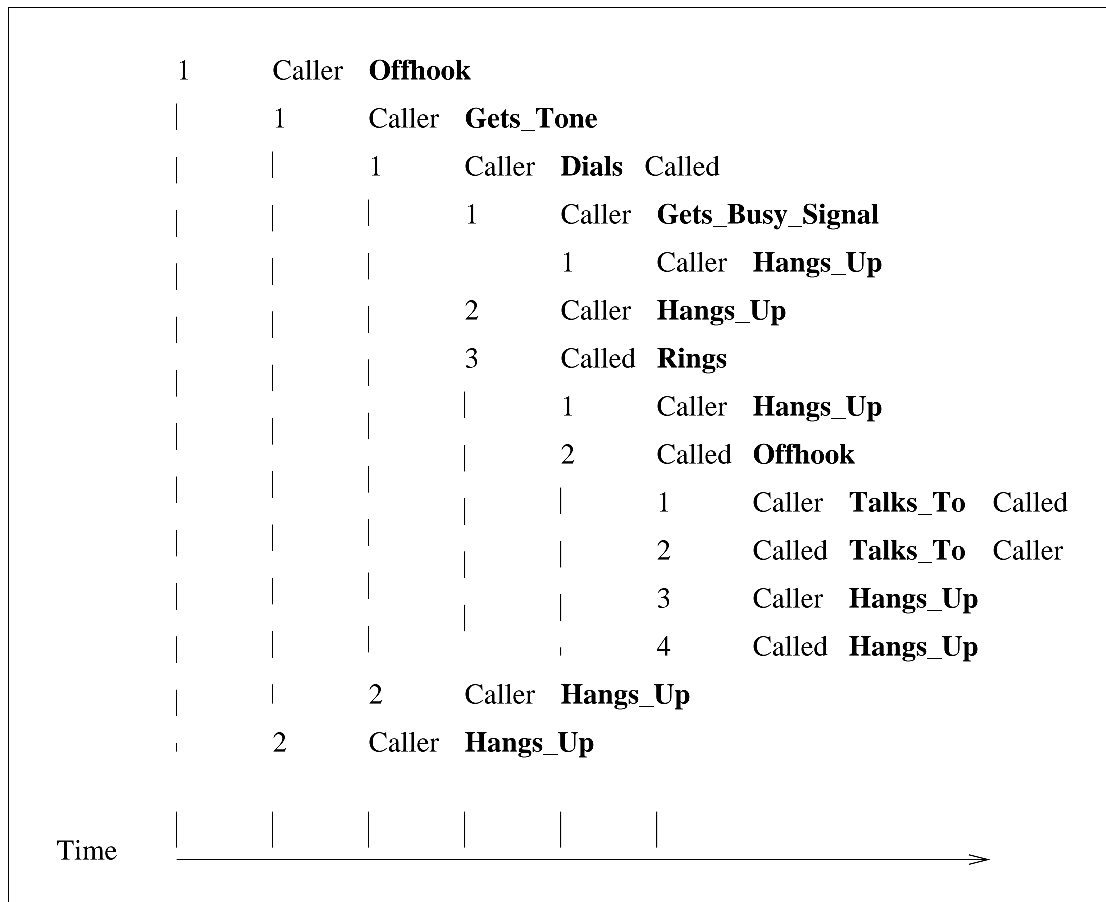

Figure 5.7: A detailed scenario for establishing a telephone connection (From [21]).

views). Accounting, routing, diagnostics, maintenance, and other aspects of POTS have their own views of the system and can be specified as separate behavioral views.

The Viewcharts specification of POTS consists of a SEPARATE composition of many, but a finite number of, identical views called CALLs. A CALL is the behavioral view of POTS with respect to a single telephone connection. Each CALL, in turn, is composed of three behavioral views: CALLER, the caller view of a telephone set, CALLED, the called view of the set, and CONTROLLER, the telephone set's view of POTS.

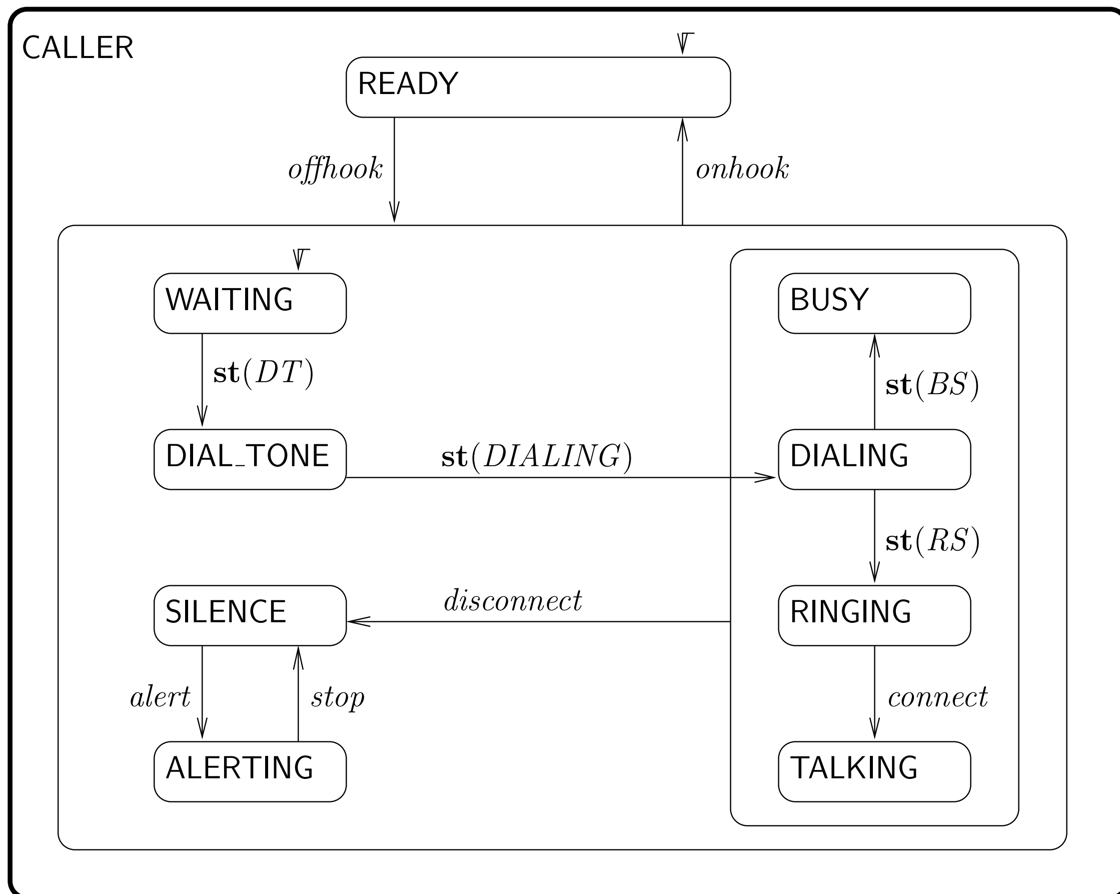

Figure 5.8: The caller view of a telephone set.

5.2.1 Specifying Behavioral Views

As a component of POTS, a telephone set has two different behavioral views: CALLER and CALLED. In specifying the behavior of a telephone set, we really want to specify these two views. The fact that a telephone set is physically a single device with two behavioral views and the issue of how it provides these behaviors are not of our concern. In fact, as far as our specification is concerned, an implementor may choose to deliver two devices: one for CALLER and the other for CALLED.

Figure 5.8 shows the caller view of a telephone set. The view specifies that the

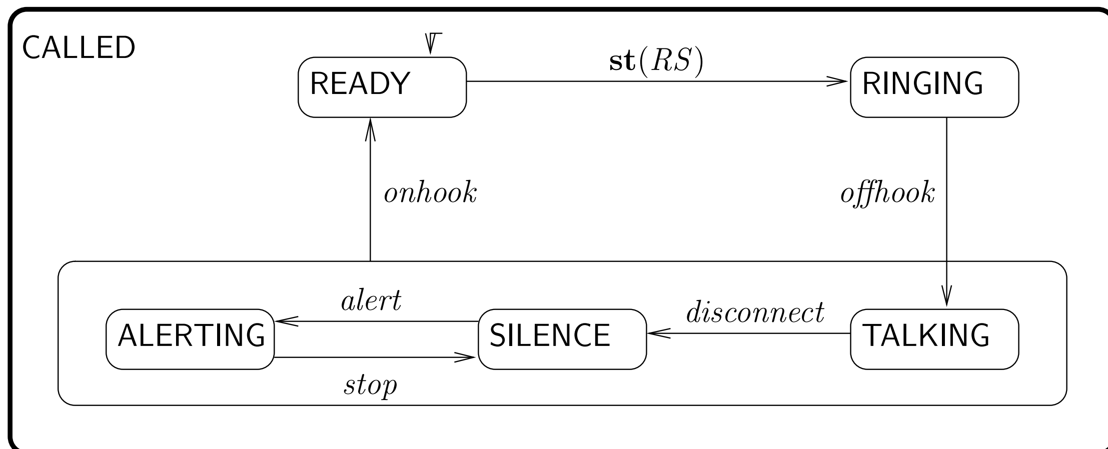

Figure 5.9: The called view of a telephone set.

telephone by default is in the READY state. When a caller picks up the handset, an event *offhook* is triggered by CALLER and the telephone set enters to the state of WAITING. In addition to the event *offhook*, the view CALLER also owns the events **st**(*DIALING*), which occurs when the user starts dialing, and *onhook*, which occurs when the user hangs up. (These events must be declared by CALLER; for clarity, however, in this and the following views, wherever the ownership of an element is obvious, we have omitted the corresponding declaration from the views.) All other events belong to CONTROLLER and are described below. As far as CALLER is concerned, however, they are events that CALLER expects to occur and upon their occurrences it behaves as specified.

Figure 5.9 shows the called view of a telephone set. This view owns only the events *offhook* and *onhook*. All other events belong to CONTROLLER. The figure is self-explanatory.

Figure 5.10 shows the view CONTROLLER. It provides the interactions between CALLER and CALLED. The view CONTROLLER owns all the events that occur in this

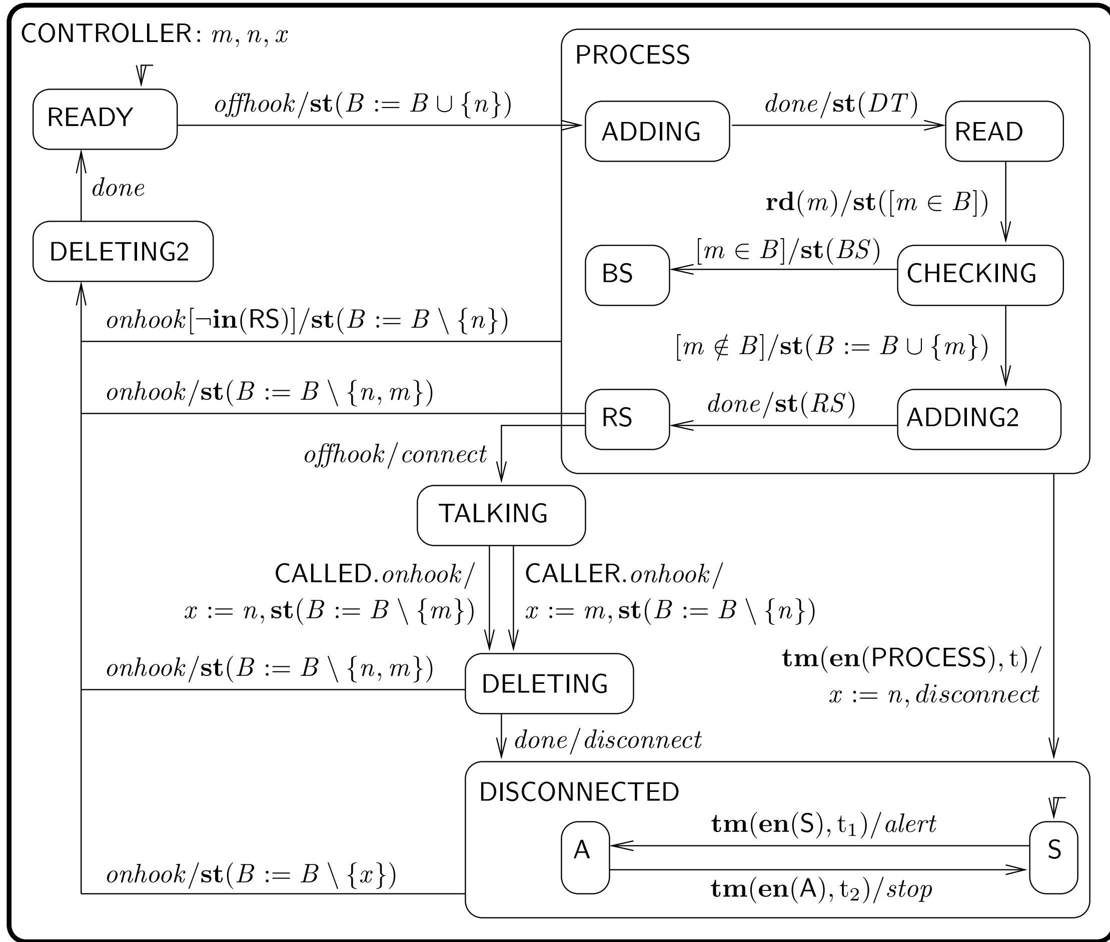

Figure 5.10: A telephone set's view of POTS.

view, except those that belong to CALLER or CALLED. CONTROLLER also declares the ownership of n (a variable used for the caller's telephone number), m (a variable used for the called's telephone number), and x (a temporary variable). In addition, CONTROLLER uses another variable B (the set of busy numbers) which is global to CONTROLLER.

As the figure shows, CONTROLLER is ready for the *offhook* event of CALLER. This event triggers an action $\mathbf{st}(B := B \cup \{n\})$, which means "start adding n to the set B". The time-consuming activity of adding n to B takes place in the state

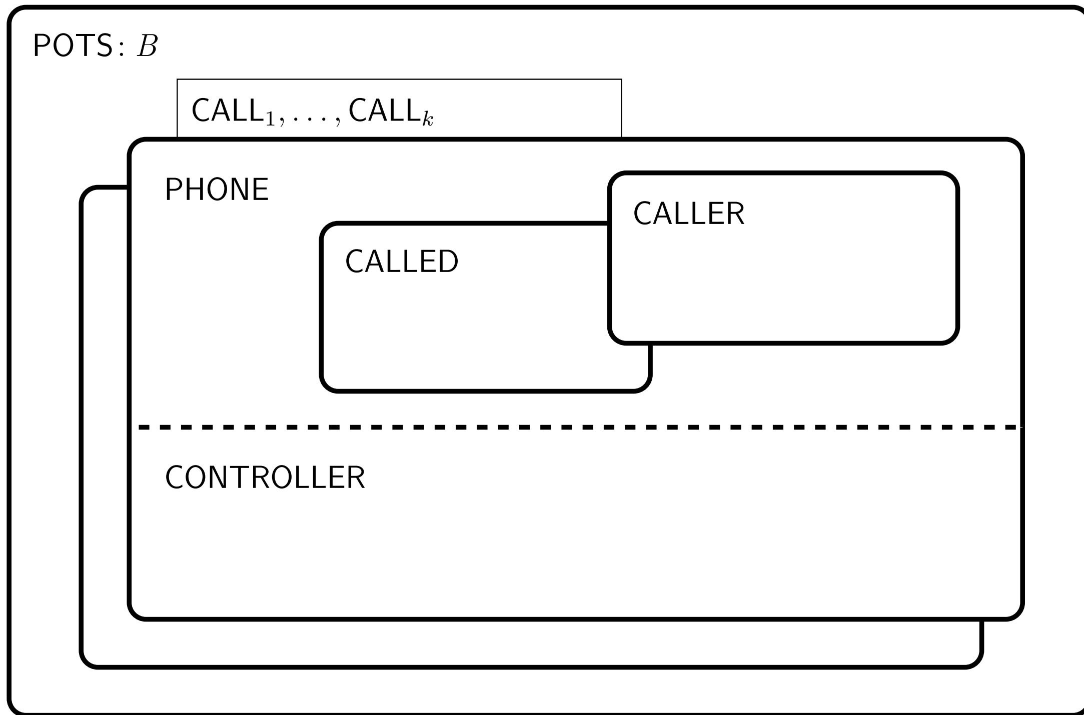

Figure 5.11: A viewchart for the telephone service provided by POTS.

PROCESS.ADDING. The completion of this activity triggers the event *done* which, in turn, triggers the action $\mathbf{st}(DT)$. The actions $\mathbf{st}(DT)$, $\mathbf{st}(BS)$, and $\mathbf{st}(RS)$, start dial-tone, busy signal, and ring signal, respectively. $\mathbf{rd}(m)$ is an event that occurs when m, the called's telephone number is read. The action $\mathbf{st}([m \in B])$ starts checking whether or not m is in B, i.e., whether or not the called party is busy. Again, the checking activity takes place in the state CHECKING. Finally, the event $\mathbf{tm}(\mathbf{en}(\mathsf{PROCESS}), \mathrm{t})$ occurs at exactly t time units after the time that the system enters to the PROCESS state. The rest of the specification in Figure 5.10 should now be self-explanatory.

5.2.2 Composing Behavioral Views

Similar to the previous example, having specified the behavioral views of the system, we can now compose them to form the overall system behavioral requirements specification. The viewchart POTS, shown in Figure 5.11, specifies the composition. A SEPARATE composition of CALLER and CALLED forms the view PHONE, which describes the behavior of the system observable at the two ends of a telephone line. PHONE is, in turn, ANDed with CONTROLLER, forming the view CALL which specifies the behavior of the system with respect to one telephone connection. Finally, POTS consists of a SEPARATE composition of k CALLs, where k is the maximum number of connections that POTS allows at any given time.

Notice that all CALLs are completely independent of each other except for sharing the variable B.

5.2.3 Discussion

As in the case of MCS, the Statecharts specification of POTS would require extending Statecharts to support parameterized repetition of AND-states. Without parameterized states, considering the number of CALLs, the Statecharts specification of POTS will not be practical. Recognizing this fact, Harel describes that such an extension "represent significant potential strengthening of the Statecharts formalism as a tool for specifying real systems" [27]. Assuming that Statecharts is extended to support this capability, Harel provides an informal diagram of a portion of a statechart that would specify a telephone system (Figure 5.12).

This diagram does not show the complete picture of the statechart; but we can still see how much more easily our Viewcharts specification of POTS expresses the

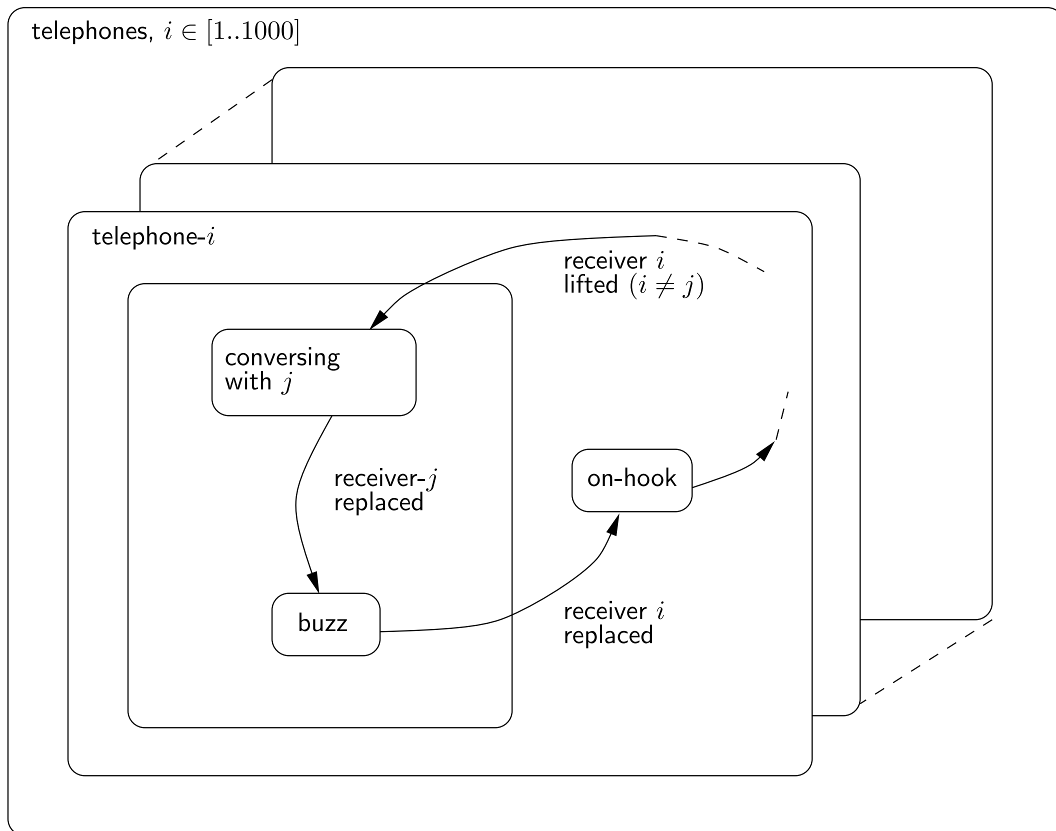

Figure 5.12: A portion of a statechart specifying POTS (From [27]).

specification compared to Statecharts. Notice that the telephone states of the statechart is similar to the CALL views of our viewchart. However, the telephone states are ANDed, while the CALL views are SEPARATEd. Therefore, in the statechart we have to make sure that all the elements are uniquely specified within the 10,000 orthogonal states; and to do that we have to use parameters like i and j (e.g., receiver i lifted, receiver j replaced, etc.). In the viewchart, on the other hand, there is no need for these parameters. As mentioned in the previous example, the parameters are within the structure of the viewchart.

The diagram of Figure 5.12 does not show a portion of the statechart, the controller, that establishes the connections between telephones. The controller must be ANDed with all telephones. To get an idea of the complexities associated with expressing the behavior of the controller, note that if the network had only two telephones, then we could express the behavior of the controller as the CONTROLLER view of our viewchart (Figure 5.10). The network, however, has many telephones and we have to express the behavior of a controller that interacts with many telephones. Many telephones may request call setup independent of each other. As in the case of DBS of MCS, we have to specify the way in which the controller must uniquely identify all these telephones and respond to their concurrent requests. Consequently, compared to our Viewcharts specification, we have to provide

- more details to uniquely identify the telephones and the events they generate;
- an additional component, a queuing mechanism, to handle the concurrent requests.

Once again, the fact that Viewcharts does not require the queuing mechanism shows the independence of Viewcharts from the design issues, supporting Item 6 of the desirable characteristics of the solution.

We do not see any of these details in our Viewcharts specification of POTS. The details are implicitly provided by our notion of views and their compositions. In summary, when we specify the behavioral requirements of a system by a viewchart, part of the specification is expressed implicitly by the structure of the viewchart. The specifications therefore, are expressed more easily in Viewcharts compared to Statecharts.

Chapter 6

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 lose their identities and be replaced by different mechanisms; the requirements cannot necessarily be traced back to their originating views. Integration may require 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 its component. In other words, the complexity of such a statechart is affected only by the complexity of the corresponding behavioral view. Consequently, since large-scale system behavior can be described in terms of

simple behavioral views, Viewcharts simplifies the specification by reducing it to the specifications of behavioral views.

6.1 Contributions

The contributions of this dissertation can be summarized in two items, an *approach* to systems behavioral specifications and a *notation* for the approach:

- *The Approach:* The dissertation has provided a novel approach to specifying behavioral requirements of complex systems, independent of implementations. In this approach the behavioral views of the system under specification are building blocks of the specification. Behavioral views are the natural outcome of requirements elicitation process, during which potential users describe their expectation (i.e., their view) of the system. Traditionally, however, requirements engineers combine behavioral views and provide a formal or informal requirements specification for the entire system. Consequently, the specification may have no formal connection to the original views described by the users. The approach provided by the dissertation has the following characteristics:
  - Directly reflects the behavioral views of the system, and thereby accurately records the original requirements collected in the process of requirements elicitation.
  - Allows the possibility of tracing each requirement to its originating view.
  - Simplifies the specifications by describing behavioral requirements of complex systems in terms of simple views.

- *The Notation:* The dissertation has introduced the Viewcharts notation for specifying and composing behavioral views. 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 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 notation allows these views to be specified independent of each other. It is, therefore, expected that the Viewcharts notation will be practical in large-scale systems behavioral specifications.

6.2 Demonstration of Claims

The following is a discussion of the Viewcharts notation with respect to the list of necessary and desirable characteristics of the proposed solution (presented in Section 1.2).

1. We have given (in Chapter 4) a precise meaning to the term “behavioral view”.

2. Viewcharts allows specification of behavioral views, fulfilling the second necessary characteristic of the solution.

3. Software behavioral requirements, in the Viewcharts notation, are specified as compositions of behavioral views. The notation, therefore, fulfills the third necessary characteristic. (The possibility of conflict between composed views is discussed in Section 3.3, Page 42 and also, as an item of future work, in Section 6.4, Page 87.)

4. Traceability: Viewcharts preserves the identity of the views; therefore, a given requirement can be traced back to its originating view. This fulfills the fourth necessary characteristic.

5. Practicality: We have shown by our examples (in Chapter 4) that not only Viewcharts is, at least, as expressive as Statecharts, but in many ways it is more expressive than Statecharts. This fulfills the fifth necessary characteristic.

With respect to the desirable characteristics:

6. Independence from design: Viewcharts generally leaves the design issues, including integration of views, to designers and implementers. This item is further discussed in Section 6.4, Page 88.)

7. Reuse of components: Software behavioral requirements specifications, in Viewcharts, are generally modular. A view is a module. Viewcharts allows independent specifications of the modules and maintains their independence to the extent possible. The SEPARATE and OR compositions preserve the independence of the modules. An element, in these compositions, is local to the view in which

it occurs. Any subview or state in one view is hidden from (i.e., cannot be referenced by) the other views. There are two cases, however, where Viewcharts does not guarantee the modularity of the specifications: (1) in an AND composition the ANDed views are visible to each other; (2) a superview has access to all its subviews. An alternative approach for the second case, which may better encapsulate the views, is discussed in Section 6.4. However, introducing the notions of "export" and "import" as an alternative for this case, as in block structured programming languages, also has drawbacks. In short, Viewcharts does not offer total modular specifications (if total modularity can be achieved), but a Viewcharts specification of a system behavioral specification can have a good degree of modularity.

6.3 Discussion

Our notion of "view" differs from that used by others, as discussed in Section 2.1. Embley and others [20], 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 [25], Ayers [4], and Rumbaugh [60], also describe the notion of view within the Model-View-Controller (MVC) paradigm of the Smalltalk community. A view, in MVC, refers to the method of presenting the information contained in the underlying model of an application. About reducing the complexity of describing the

model, they neither make any claim nor offer any mechanism.

STATEMATE's solution to the problem of reducing the complexity of scale (name space, basically) is its language of *activity-charts* [34]. This language is used for functional description of the system under design. To reduce the complexity of scale, STATEMATE suggests a functional decomposition of the system using activity-charts. An activity-chart is like a data flow diagram; it describes the system's functions (or activities, in STATEMATE terminology) and the flow of data between them. The behavior of some activities, called *control* activities, are then described by encapsulated statecharts. These statecharts can only interact with each other via a data-flow mechanism provided by the activity-chart. Similarly, RSML [53] and ROOMcharts [62] reduce the complexity of managing name space by providing direct communications between *components*. (A component is a statechart in RSML and an actor in ROOMcharts.)

There are similarities between our notion of view and the control activity of STATEMATE, the component of RSML, and the actor of ROOMcharts. However, there are also major differences: The direct communications mechanisms provide limited interactions between control activities, components, or actors, while Viewcharts provides whatever interaction required by the composed views.

Our notion of a view is the behavior of the system under specification observable from a specific point of view; and as far as an observer at the point of view is concerned, that is the whole system. Our notion of view, therefore, reduces the problem of specifying the behavior of a system to specifying its behavioral views.

6.4 Future Directions

This dissertation has introduced the Viewcharts notation to the extent needed to establish the claims presented in Section 1.2. 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 Chapter 4. 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. To establish a conference call, for example, a caller dials several numbers, switches between them, and sets up several connections. A simple way of specifying such behavior would require transitions between several CALLER views (See Section 5.2).

- *Exporting and importing elements*: In a viewchart hierarchy of views, the scope of an element owned by a view covers the view and all its subviews. There are, however, other alternatives that should be explored. It may, for example, better encapsulate the views to limit the scope of an element to the view that owns the element, and then to introduce the notions 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.

- *Modeling*: The requirements activities of the software engineering process are generally performed in three stages: The first stage, *problem analysis and requirements elicitation*, includes understanding the problem to be solved and collecting the requirements for a possible solution i.e., system to be developed. The activities in this stage are generally informal and include interviews with potential users. This dissertation has been concerned basically with the second stage, *requirements specification*, precisely developing requirements. Further research is required on the third stage, *analysis of the specification*, which includes proving system properties and checking the specification for completeness, consistency, ambiguity, feasibility, and so on.

  On the issue of consistency, as discussed in Section 3.3, Page 42, there are arguments that structuring specifications into views generally makes it easier to deal with conflicts than if the view specifications were intertwined from the start [46]. Nonetheless, the possibility of conflicts between views does exist in Viewcharts and a good topic of future research is to investigate the way in which these conflicts can be recognized and dealt with.

  Joanne Atlee [2, 3] describes how the requirements of a software system, specified in *Software Cost Reduction* (SCR) notation, can be analyzed. She translates the SCR requirements into a logical model, expresses the system properties as logical formulas, and proves whether or not the formulas hold in the model. A similar approach for Viewcharts specifications is a possible future direction of this research.

  We have shown in Chapter 4 that for a given viewchart there exists an equivalent statechart. It is possible to produce an executable model of a viewchart by

constructing the equivalent statechart and using 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.

- *Viewcharts in design stage*: Viewcharts belongs to a class of languages which are designed to express the behavioral requirements of a system independent of design and implementation issues. A designer can expect from a Viewcharts specification only an accurate description of what to do. The specification is not intended to provide any method, approach, or guideline of how the design work should proceed. Therefore, Viewcharts neither complicates nor facilitates the design stage.

  On the other hand, there are languages, like ROOM [62] described in Section 2.5, that link specification to design and implementation. These languages believe in a smooth transition from the requirements specification stage of software development process to the design and implementation stage.

  A somewhat general problem is to investigate the effect of specifications written in the two classes of languages (e.g., Viewcharts and ROOM) on the design stage and determine to what extent each one facilitates or complicates the design stage.

- *Feature Interaction Problem*: Finally, an interesting problem will be to extend our POTS example of Section 5.2 and specify various telephone features using Viewcharts. The specification then has to deal with the problem of *feature interactions* [71].

Bibliography

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[2] J. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 280–292, January 1996.

[3] J. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[4] K. E. Ayers. The MVC paradigm in Smalltalk4. *Dr. Dobb's Journal: Software Tools for the Professional Programmer*, 15(7):168–175, November 1990.

[5] 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, March 1993.

[6] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design, an Entity-Relationship Approach.* Benjamin/Cummings Publishing Company, Redwood City, California, 1992.

[7] 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.

[8] 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.

[9] 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.

[10] C. J. Date. *An Introduction to Database Systems.* Addison Wesley, Reading, Massachusetts, 1986.

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

[12] N. Day. A model checker for Statecharts. Technical Report 93-35, Department of Computer Science, University of British Columbia, Vancouver, BC, Canada, October 1993.

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

[14] 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.

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

[16] D. Drusinsky and D. Harel. On the power of bounded concurrency I: Finite automata. *Journal of the ACM*, 41(3):517–539, May 1994.

[17] 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.

[18] S. Easterbrook and B. Nuseibeh. Using ViewPoints for inconsistency management. *To appear in BCS/IEE Software Engineering journal*, November 1995.

[19] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings Publishing Company, Redwood City, California, 1994.

[20] 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.

[21] M. Faci, L. Logrippo, and B. Stepien. Formal specification of telephone systems in LOTOS: The constraint-oriented approach. *Computer Networks and ISDN Systems*, 21:53–67, 1991.

[22] A. Finkelstein, J. Kramer, and M. Goedicke. ViewPoint oriented software development. In *Proceedings of 3rd International Workshop on Software Engineering and its Applications*, Toulouse, December 1990.

[23] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in systems development. *International Journal on Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992. Special issue on Trends and Research Direction in Software Engineering Environments.

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

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

[26] J. Guttag, J. Horning, and J. Wing. Some notes on putting formal specifications to productive use. *Science of Computer Programming*, 2(1):53–68, October 1982.

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

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

[29] D. Harel. Private communication, November 1995. Weizmann Institute of Science, Rehovot, Israel.

[30] 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.

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

[32] 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.

[33] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of Statecharts. In *Proceedings of Symposium on Logic in Computer Science*, pages 231–274, Ithaca, New York, June 1987.

[34] D. Harel and M. Politi. The languages of STATEMATE. Technical report, i-Logix, Inc., 22 Third Avenue, Burlington, Mass. 01803, USA, November 1991.

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

[36] K. L. Heninger, J. W. Kallander, J. E. Shore, and D. L. Parnas. Requirements for the A-7E aircraft. Technical Report NRL 3876, Naval Research Laboratory, Washington, DC, November 1978.

[37] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 8(21):666–677, August 1978.

[38] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[39] D. Hoffman and P. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Press, 1995.

[40] 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.

[41] J.J.M Hooman, S. Ramesh, and W.P. de Roever. A compositional axiomatization of Statecharts. *Theoretical Computer Science*, 101(2):289–335, July 1992.

[42] C. Huizing and W. P. deRoever. Introduction to design choices in the semantics of Statecharts. *Information Processing Letters*, 37(4):205–213, February 1992.

[43] C. Huizing and R. Gerth. Semantics of reactive systems in abstract time. In *Proceedings of International Conference on Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 291–314, New York, June 1991. Springer-Verlag.

[44] C. Huizing, R. Gerth, and W. P. de Roever. Modeling Statecharts behavior in a fully abstract way. In *Proceedings of 13th Colloquium on Trees in Algebra and programming*, volume 299 of *Lecture Notes in Computer Science*, pages 271–294, Nancy, France, March 1988. Springer-Verlag.

[45] A. Isazadeh. Configuration languages for distributed software systems. Ph.D. Depth Paper, Department of Computing and Information Science, Queen's University, Kingston, Canada, August 1994.

[46] D. Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389, October 1995.

[47] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering, A Use Case Driven Approach.* Addison Wesley, Reading, Massachusetts, 1992.

[48] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12:890–904, September 1986.

[49] 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.

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

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

[52] Y. Kesten and A. Pnueli. Timed and hybrid Statecharts and their textual representation. In *Proceedings of Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 591–620, New York, January 1992. Springer-Verlag.

[53] 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.

[54] 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.

[55] F. Maraninchi. Operational and compositional semantics of synchronous automation composition. In *Proceedings of Third International Conference on Concurrency Theory (CONCUR'92)*, volume 630 of *Lecture Notes in Computer Science*, pages 550–564, New York, August 1992. Springer-Verlag.

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

[57] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, October 1994.

[58] J. L. Peterson. Petri Net. *Computing Surveys*, 9(3):223–252, September 1977.

[59] A. Pnueli and M. Shalev. What is a step: On the semantics of Statecharts. In *Proceedings of International Conference on Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264, New York, September 1991. Springer-Verlag.

[60] J. Rumbaugh. Modeling models and viewing views: A look at the Model-View-Controller framework. *Journal of Object-Oriented Programming*, pages 14–, May 1994.

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

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

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

[64] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, Maryland, 1980.

[65] A. J. van Schouwen. The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Technical Report 90-276, Telecommunications Research Institute of Ontario (TRIO), Department of Computing and Information Science, Queen's University, Kingston, Canada, May 1990.

[66] M. von der Beek. A comparison of Statecharts variants. In *Proceedings of Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, New York, September 1994. Springer-Verlag.

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

[68] M. S. White. Requirements: A quick and inexpensive way to improve testing. Testing Techniques Newsletter (TTN), On-Line Edition, December 1994. ttn@soft.com.

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

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

[71] P. Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, 26(8):20–30, August 1993.

[72] P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, October 1993.

Vita

NAME Ayaz Isazadeh

EXPERIENCE Over **ten years of industrial experience,** mainly with AT&T Bell Labs, in telecommunications, manufacturing automation, and control systems, covering all phases of software systems engineering; some teaching and research assistantship (Queen's University); and **two years of full-time academic experience** (Tabriz University), teaching undergraduate computer science courses.

EDUCATION

Ph.D. Computing and Information Science,
Queen's University, 1996 (expected).

M.S.E. Electrical Engineering and Computer Science,
Princeton University, 1978.

B.Sc. Mathematics, Tabriz University, 1971.

AWARDS Several scholarship awards.

AFFILIATIONS Senior Member of IEEE and IEEE Computer Society.

PUBLICATIONS

Published or Accepted Refereed Journal or Conference Papers:

[1] A. Isazadeh, T. Shepard, and D. A. Lamb. A review of configuration languages for distributed software systems. *IEEE Transactions on Software Engineering Journal*. To appear.

[2] A. Isazadeh and D. A. Lamb. An algorithmic semantics for Viewcharts. In *Proceedings of IEEE Complex Systems Engineering Synthesis and Assessment Technology Workshop (CSESAW)*, Montreal, Canada, October 1996. IEEE Computer Society Press. To appear.

[3] A. Isazadeh, D. A. Lamb, and G. H. MacEwen. Viewcharts: A formalism for complex systems. In *Proceedings of 4th Iranian Conference on Electrical Engineering (ICEE-96)*, pages 498–505, Tehran, Iran, May 1996.

[4] A. Isazadeh, D. A. Lamb, and G. H. MacEwen. Viewcharts: A behavioral specification language for complex systems. In *Proceedings of International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, pages 208–215, Honolulu, Hawaii, April 1996. IEEE Computer Society Press.

[5] A. Isazadeh, D. A. Lamb, and G. H. MacEwen. Behavioral views for software requirements engineering. In *Proceedings of IEEE Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'96)*, pages 300–307, Friedrichshafen, Germany, March 1996. IEEE Computer Society Press.

[6] A. Isazadeh, G. H. MacEwen, and A. Malton. A review of post-factum software integration methods. In *Proceedings of Computer Society of Iran Computer Conference (CSICC'95)*, pages 91–101, Tehran, Iran, December 1995.

[7] A. Isazadeh, G. H. MacEwen, and A. Malton. Behavioral patterns for software requirements engineering. In *Proceedings of IBM CASCON'95*, Toronto, Canada, November 1995.

Technical Reports and Other Publications:

[8] A. Isazadeh, D. A. Lamb, and G. H. MacEwen. The semantics of Viewcharts. External Technical Report ISSN-0836-0227-95-395, Department of Computing and Information Science, Queen's University, Kingston, Canada, December 1995.

[9] A. Isazadeh, G. H. MacEwen, and A. Malton. A review of post-factum software integration methods. External Technical Report ISSN-0836-0227-95-389, Department of Computing and Information Science, Queen's University, Kingston, Canada, October 1995.

[10] 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, Department of Computing and Information Science, Queen's University, Kingston, Canada, October 1995.

[11] A. Isazadeh. Configuration languages for distributed software systems. Ph.D. Depth Paper, Department of Computing and Information Science, Queen's University, Kingston, Canada, August 1994.

[12] A. Isazadeh. Behavioral views for software requirements engineering: Extended abstract. In *Proceedings of TRIO/ITRC Researcher Retreat*, Queen's University, Kingston, Canada, May 1996.

[13] A. Isazadeh. Configuration languages for distributed software systems: Extended abstract. In *Proceedings of TRIO/ITRC Researcher Retreat*, Queen's University, Kingston, Canada, May 1995.

[14] A. Isazadeh. Post-factum integration of software systems: Problems and approaches — extended abstract. In *Proceedings of TRIO/ITRC Researcher Retreat*, Queen's University, Kingston, Canada, May 1994.