TIMETREES: A BRANCHING-TIME STRUCTURE FOR MODELING ACTIVITY AND STATE IN THE HUMAN-COMPUTER INTERFACE

by

Jeffrey L. Brandenburg

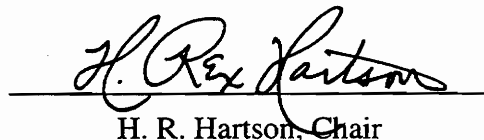
Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

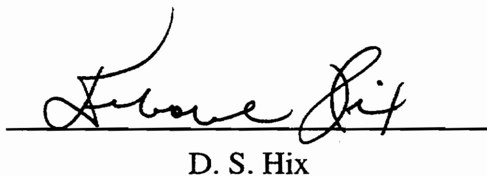
in

Computer Science

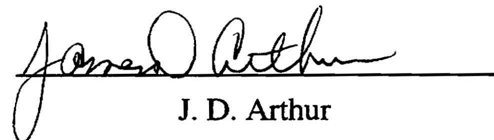
APPROVED:



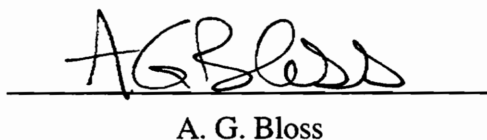
H. R. Hartson, Chair



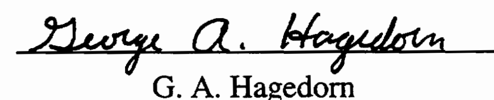
D. S. Hix



J. D. Arthur



A. G. Bloss



G. A. Hagedorn

April, 1995
Blacksburg, Virginia

Keywords: human-computer interaction, formal models, temporal logic, design representations, task analysis, User Action Notation

c.2

LD
5655
V856
1995
B736
c.2

TIMETREES: A BRANCHING-TIME STRUCTURE FOR MODELING ACTIVITY AND STATE IN THE HUMAN-COMPUTER INTERFACE

by

Jeffrey L. Brandenburg

Committee Chair: Dr. H. Rex Hartson
Computer Science

(ABSTRACT)

The design and construction of interactive systems with high usability requires a user-centered approach to system development. In order to support such an approach, it is necessary to provide tools and representations reflecting a *behavioral* view of the interface—a view centered on user activities and the system activities and states perceived by the user. While behavioral *representations* exist, there is no behavioral *model* of interaction between a user and a system. Such a model is necessary for formalization and extension of existing behavioral representations.

This dissertation presents a model of interactive behavior based on the *timetree*, a novel tree-based structure representing tasks, user actions, system activity, and system and interface state, all within a framework of branching sequential timelines. The model supports formal definitions, operations and abstraction techniques. Three application areas—a formal definition of an existing behavioral notation, connection between a behavioral representation and a formal model of input devices, and techniques for analysis of behavioral specifications—provide examples of the model's utility.

Dedication

To my family —

whose ties of blood and love support me in the present,
nourished me throughout my past, and will sustain me
for all my future.

Acknowledgements

All my training leads me to value brevity in writing. I will be as brief here as I can, but the process that led to this document was not brief, and the barest mention of those whose support has brought me here requires extended space and time.

This all begins with my parents. Their support, both financial and emotional, has sustained me throughout my academic career. But, more than that, the upbringing they provided made all this possible in the first place. They worried and wondered as all parents must, but now this stands as their achievement as well as mine.

Rex Hartson and Deborah Hix brought me into the DMS project as an undergraduate, and it remained my home for many years. They have been fine advisors, employers, friends, and role models, and it is a great privilege to work with them.

The other members of my committee, Sean Arthur, Adrienne Bloss, and George Hagedorn, have been supportive, helpful, and extremely patient. Their suggestions and guidance have contributed greatly to this document, and I am very grateful for their efforts on my behalf.

I've been blessed with many friends and companions during this process. The Multicians, especially Dave, Mark, and Stan, challenged and inspired me as I immersed myself in the mysteries of computation. Lisa and June helped set me on the path that kept me here; I thank them for that, and for all the time I had with them. Dave and Sharon taught me much of what I know about being a friend; I miss them, and I hope for the future. Eric was a great co-worker, and he and his wife Laura remain good friends and valued teachers. The music folks, John-John, Detlef, Geoff, Kevin, Sheila, Margaret, Melanie, and the rest, helped me out of several shells and shared many wonderful hours by the (ahem) fountain. Annie lurked as an acquaintance for a long time before revealing herself as a friend. Tara, Gordon, and Carey and Honey gave me support than only fellow doctoral students can share, in addition to all their other wonderful gifts. Tara also thoughtfully provided her sister Dana as a neighbor, close friend, and hiking pacesetter.

Shawn has been a friend almost from the beginning of my graduate career. Our lives have brought us together and kept us apart in the strangest of ways, but after changes upon changes, our connection endures. She's helped me to see the twists and turns to come as something to look forward to, instead of something to fear.

During my time in Blacksburg, Lauri became an acquaintance, then a friend, then a companion, then my wife. My pursuit of this degree has cost more than I ever expected, far beyond money or even time, and she has borne more of the expense than anyone. But she has remained steadfast, and kept faith in me even in the times when I lost faith in myself. What she has given me cannot be repaid; what I give her in return is all that I am to become, for the rest of our lives. *"This is what I give; this is what I ask you for..."*

It seems strange to shovel so many names into a few paragraphs and call it an "acknowledgement." All I can do here is name people and thank them; I cannot begin to show what they have brought to my life. We are all joined by threads of friendship and love, interests and enthusiasms, confidences and coincidences, music and words, journeys and destinations; and the tapestry that they form lends warmth and comfort and beauty to all of my life.

Thank you all.

Contents

Dedication iii

Acknowledgements iv

List of Figures x

1 Problem Statement 1

2 Background and Motivation 3

2.1 The need for behavioral support ... 3

2.1.1 Areas of system development ... 4

2.1.2 Processes, boundaries, and communication ... 5

2.1.3 Interaction design as a separate role ... 10

2.1.4 Design representations ... 12

2.2 The User Action Notation ... 13

2.3 The case for a behavioral model ... 16

2.3.1 Problems of definition ... 16

2.3.2 Connection to a formal model of input devices ... 17

2.3.3 Analysis of behavioral descriptions ... 18

2.4 Our response: Timetrees ... 18

3 An Introduction to Timetrees 19

3.1 A single task ... 19

3.2 A set of composed tasks ... 21

3.3 Low levels of abstraction ... 25

3.4 Ambiguities and design conflicts ... 26

3.5 Associated state information ... 28

3.6 Conditions of viability and state components ... 30

3.7 Timestate equivalence ... 32

4 **Related Work** 35

4.1 Behavioral representation techniques and user models 35
- 4.1.1 The keystroke-level model 35
- 4.1.2 Action grammars 36
- 4.1.3 CLG 37
- 4.1.4 GOMS 38
- 4.1.5 Kieras & Polson's cognitive complexity theory (CCT) 40
- 4.1.6 TAG 40

4.2 Related constructional models 42
- 4.2.1 Jacob's specification diagrams 42
- 4.2.2 UIDE 44

4.3 A Device Model 45

4.4 Branching Time Temporal Logic 46

4.5 Models of Concurrency 48
- 4.5.1 CCS 48
- 4.5.2 CSP 49

5 **The Timetree Model** 51

5.1 Formal representation 51

5.2 Timetree components 56
- 5.2.1 Activities 56
- 5.2.2 Timestates 57
- 5.2.3 State components 58

5.3 Compositions 59
- 5.3.1 Choice 59
- 5.3.2 Sequence 65
- 5.3.3 Interleaving 73
- 5.3.4 Concurrency 80
- 5.3.5 Disambiguation 86
- 5.3.6 Closure 89

5.4 Abstractions 92
- 5.4.1 Path pruning 92
- 5.4.2 Task-like abstractions 94
- 5.4.3 State component propagation 97
- 5.4.4 State component pruning 98
- 5.4.5 State component equivalence classes 98

6 **Formal Definition of UAN** **100**

6.1 Entities 100
6.1.1 Input devices 101
6.1.2 Display objects 102

6.2 User actions 104
6.2.1 Keys and buttons 104
6.2.2 Cursor movement 105

6.3 System feedback actions 107

6.4 Interface state and connection to computation 110

6.5 Conditions of viability 112
6.5.1 Conditional composition 115
6.5.2 Virtual activities 117
6.5.3 State propagation 123

6.6 Composition 126
6.6.1 Sequence 127
6.6.2 Choice 130
6.6.3 Grouping 132
6.6.4 Repetition 132
6.6.5 Optional performance 135
6.6.6 Order independence 135
6.6.7 Interruptibility and interleavability 135
6.6.8 Concurrency 150
6.6.9 Intervals and waiting 156

7 **Connecting a Device Model to the UAN** **159**

7.1 The CMR device model 160
7.1.1 Primitive movement vocabulary 160
7.1.2 Composition operators 161

7.2 The UAN view of devices 162

7.3 Translation from CMR to UAN device representations 164
7.3.1 Primitive devices and actions 164
7.3.2 Composite devices 167

8 Analytic Techniques **171**

8.1 Global interface analysis 173
8.1.1 Number of tasks 174
8.1.2 Number of user actions 174
8.1.3 Number of entities 175
8.1.4 Number of states and classes of state 175
8.2 Timing analysis 176
8.2.1 Performance prediction 176
8.2.2 System response specification 179
8.3 Local optimization within tasks 180
8.4 Applications in early stages of development 182
8.4.1 Undefined and unused tasks 182
8.4.2 Redundant tasks 183
8.4.3 Task design inconsistencies 184
8.4.4 Coverage of possible action sequences 186
8.4.5 Walkthrough and prototype support 188
8.5 Task information at run time 189

9 Future Work **191**

9.1 Translation to constructional representations 191
9.2 Philosophical questions 193
9.2.1 Infinite branching 193
9.2.2 User choice and "system choice" 195
9.3 Model extensions 195
9.3.1 Examination of new interface styles in relation to timetrees 195
9.3.2 Extension to multiple users and multiple systems 196
9.3.3 More general treatment of concurrency 196
9.3.4 Ranges in state values 197
9.3.5 Relations among activities and state values 198
9.3.6 Probable/unlikely choices 198
9.4 UAN extensions and problems 199
9.4.1 Context issues 199
9.4.2 Completeness of UAN specifications 200
9.4.3 Arenas and excursions 201
9.4.4 Interruptability and task abandonment 202
9.4.5 User-system concurrency 203
9.4.6 New forms of task composition 204

10 Conclusions **206**

Bibliography **210**

Vita **213**

List of Figures

2.1. Relationships among areas of interactive system. ... 5
2.2. Parts of software system development. ... 5
2.3. Parts of interactive system development. ... 7
2.4. UAN description of Delete File task. ... 15
3.1. Modified UAN description of Delete File task. ... 20
3.2. Timetree for Delete File task. ... 20
3.3. UAN description of Select File task. ... 21
3.4. UAN description of Move File Icon task. ... 21
3.5. UAN description of Manipulate File task. ... 22
3.6. Timetree for Manipulate File task. ... 22
3.7. Timetrees for Select, Move Icon, and Delete File. ... 23
3.8. Simple combination of timetrees. ... 23
3.9. Disambiguated timetree. ... 24
3.10. Timetree with system activities. ... 25
3.11. UAN descriptions of new Delete, Copy, and Manipulate tasks. ... 27
3.12. Timetree showing conflict in new Manipulate File task. ... 28
3.13. Disambiguated timetree with state information for Manipulate File. ... 29
3.14. UAN description of Manipulate File incorporating viability condition. ... 30
3.15. Timetree for (Manipulate File)*. ... 31
3.16. Extremely high-level timetree for Manipulate File task. ... 32
3.17. Manipulate File subtasks with significant state information. ... 33
3.18. Manipulate File subtasks after timestate condensation. ... 34
5.1. Representations of a sample timetree. ... 55
5.2. Choice composition of two timetrees. ... 60
5.3. Sequence composition of two timetrees. ... 66
5.4. Timetree interleaving before first step. ... 74
5.5. Timetree interleaving after one step. ... 74
5.6. Timetree interleaving after two steps. ... 75
5.7. Timetree interleaving after completion. ... 76
5.8. Partitioning and interleaving of concurrent activities. ... 82
5.9. Concurrent composition with propagation of continuations. ... 83

6.1. UAN entities and state representations. 104
6.2. Cursor movement activities and their associated state information. 106
6.3. System feedback activities and their associated state information. 107
6.4. Detailed timetree for "cursor-following" behavior. 109
6.5. Detailed state information for "cursor-following" behavior. 109
6.6. Conditional composition of timetrees with state conflicts. 116
6.7. (*ta* ; *tb*) after pruning non-terminal leaf timestates. 117
6.8. Timetree with virtual activities. 118
6.9. Timetree with virtual activities before and after disambiguation. 121
6.10. Timetree before and after collapsing virtual activities. 121
6.11. Sequential composition with disjoint state components. 123
6.12. Timetree for **X!** ; **Y!** with interposed virtual activity. 125
6.13. Timetree for **set-preconds** ; (**X!** ; **Y!**). 125
6.14. Timetree for **set-preconds** ; (**X!** ; **Y!**) after state propagation. 125
6.15. Timetree for **set-preconds** ; (**X!** ; **Y!**) after propagating state values and collapsing virtual activities. 125
6.16. Sequential composition with a virtual activity. 129
6.17. Choice composition with virtual activities. 131
6.18. Grouping and composition. 132
6.19. Star closure for timetrees representing UAN tasks. 133
6.20. Choice composition with the null timetree. 134
6.21. Timetrees for **X! Y!**, **X-!**, and **(X! Y!)** ⇔ **X-!**. 145
6.22. Timetree for **(X! Y!)** ⇔ **X-!** after state propagation. 146
6.23. Timetree for **(X! Y!)** ⇔ **X-!** after collapsing virtual activities. 146
6.24. Timetree for **(X! Y!)** ⇔ **X-!** after pruning nonterminal paths. 147
6.25. Timetrees for **X! Y!**, **X-!**, and **(X! Y!)** ⇔ **X-!** with state propagated before composition. 148
6.26. Timetree for **(X! Y!)** ⇔ **X-!** with state propagated before composition, state propagated after composition, and virtual activities collapsed. 149
6.27. Timetree from Figure 6.26 for **(X! Y!)** ⇔ **X-!** after pruning nonterminal paths. 149
6.28. Timetree representing **A B** with explicit interval. 157

7.1. Physical properties sensed by input devices. 161
7.2. UAN verbs for manipulation forms. 165
7.3. Timetree for primitive manipulation. 166
7.4. Timetree for merge composition. 167
7.5. Timetree for component device in layout composition. 168
7.6. Timetree for merge composition. 169
7.7. UAN definitions of devices during successive composition. 170
8.1. A possible graphical representation of an interface design with unconnected and undefined components flagged. 183
8.2. A task-labeled timetree. 190
9.1. Infinite timetree with infinite number of different state values. 193
9.2. Infinitely branching tree generated from timetree of Figure 9.1 194

1 Problem Statement

Improving usability of computer systems is a pervasive goal. The increasing prevalence of interactive computer systems has given rise to a variety of software tools and techniques supporting development of such systems. These tools are supposed to speed the interface development process, decrease its cost, and improve the usability of the resulting systems.

But development of usable systems requires more than software tools. It has long been acknowledged that systems analysis, design, and coding are different processes; each requires different skills, and indeed, a degree of isolation among the separate processes can improve the quality of the final product. It is now becoming clearer that distinct roles for *interface* development are also important.

Interface developers work in a *behavioral* realm. The concepts they manipulate include user tasks, user actions, and system activities which the user can perceive. These all reflect *behaviors* of the user — planning actions with the system, executing those actions, and perceiving the system's responses. These behaviors are of paramount importance in interface designs; the user's view of the system, fundamental to high usability, is founded on them, and so the interface developer must think, work, and speak in their terms.

In order to record and communicate these interface designs, the interface developer needs a *behavioral representation* — a user- and task-oriented design notation that allows designs to be specified in behavior-oriented terms. The User Action Notation (UAN) [Hartson, Siochi, & Hix, 1990] is one notation that has been developed to fulfill this need. An *interface* design, expressed in UAN, provides requirements for the *interface software* design, which then becomes the specification for the interface implementation.

So far, though, no *model* exists for the phenomena of the behavioral realm. The lack of such a formal model makes it difficult to produce anything more than ad hoc representation, design, and analysis techniques. Researchers and practitioners who develop and refine such techniques find it difficult to maintain consistency in design representations, to provide automated support for interface designers, and to develop

analysis and translation tools. Many models exist for computation, software structure, and even interaction in the *constructional*, system-centered realm. There are also *behavioral* models of users and their mental processes. But so far, no model focuses on *activity between user and system* in the behavioral realm.

The goal of this research has been to develop and evaluate such a model — specifically, a tree-based formal model of behavioral phenomena, including tasks, user actions, system activity, and system and interface state, all unfolding over time. We have developed the model based on previous models of nondeterministic systems and our own observations of behavioral phenomena. We demonstrate the model's utility in support of behavioral notation and analysis by applying it to three problems: a formal description of UAN structures and components, linkage to a formal theory of input devices, and techniques for analysis of behavioral specifications.

The central contribution of this research is the timetree model's formal representation of behavioral activities and state. By defining the UAN in terms of timetrees, we fulfill a long-standing need to resolve ambiguities in previous UAN definitions, and provide a rational foundation for extensions to the UAN that new application domains and interface styles may require. By formally representing behavioral interface specifications that emerge early in the system development process, we provide opportunities for analyzing interface characteristics even before prototypes exist, and throughout the entire process.

2 Background and Motivation

Our interest in modeling behavioral phenomena has grown out of our experience with behavioral notations, which in turn has come from our investigation of the interactive system development process. To establish a context for our model, this chapter presents our view of the development process and behavioral representations.

In Section 2.1, we discuss the system development process as a whole, different roles within the process, communication among these roles, and the importance of behavioral notations for communication. In Section 2.2, we describe the User Action Notation, a popular behavioral design technique. In Section 2.3, we summarize some of the problems that motivate us to develop a behavioral model.

2.1 The need for behavioral support

It is now well accepted that an iterative, evaluation-centered development process [Hartson & Hix, 1989] involving rapid prototyping and usability engineering [Whiteside, Bennett, & Holtzblatt, 1988] is a much more effective approach to development of a *usable* system than the conventional top-down software development paradigm based on functional decomposition and linear life cycle phases. In this context, the term “development” encompasses the entire system development process-in-the-large, including requirements analysis, task analysis, user definition, interface design, design representation, rapid prototyping, formative evaluation, design of interface software, and construction, coding, and maintenance of interface software. Each of these development tasks poses different requirements for support tools and languages.

Presentation and specification languages used in activities occurring early in the development process (for example, task analysis and interface design) must be oriented toward a view of the user’s behavior. *Behavioral* design and representation involves physical and cognitive user actions and interface feedback — the behavior both of the user and of the interface as they interact with each other. Each behavioral design must be translated into a *constructional* design that is the computer-system-oriented view of how

the behavior is to be supported. Any description that can be thought of as "executed by the machine" is constructional. This includes control flow and data flow mechanisms, state transition diagrams, event handlers, object-oriented representations, and many interface description languages.

In contrast, behavioral descriptions can be thought of as "performed by the user." Many existing interface representation techniques, especially those associated with User Interface Management Systems (UIMSs), are constructional. (Section 2.1.4 presents references to some of these techniques.) Such representations make it difficult to maintain a user-centered focus, because *it is in the behavioral domain that interaction designers and evaluators do their work.* Thus, there is a need for behavioral representation techniques — languages and supporting tools — to give a user-centered focus to the interface development process. Further, since these roles and activities must communicate with other roles and activities of the development process, there is a need for languages and tools to facilitate this communication.

2.1.1 Areas of system development

Figure 2.1 provides one overview of relationships among various areas of interface development within the context of overall system development. (This is not an architectural diagram of an interactive system or UIMS.) From the outside in, we have overall interactive system development, development of the part of the interactive system that is computer-based, user interface development, user interface software development, and interaction development. Each of these domains of development involves its own life cycle with its own set of processes. Except for projects to develop new computing systems or workstations, user interface developers usually select (or are presented with) existing interaction devices, rather than designing these items *de novo.*

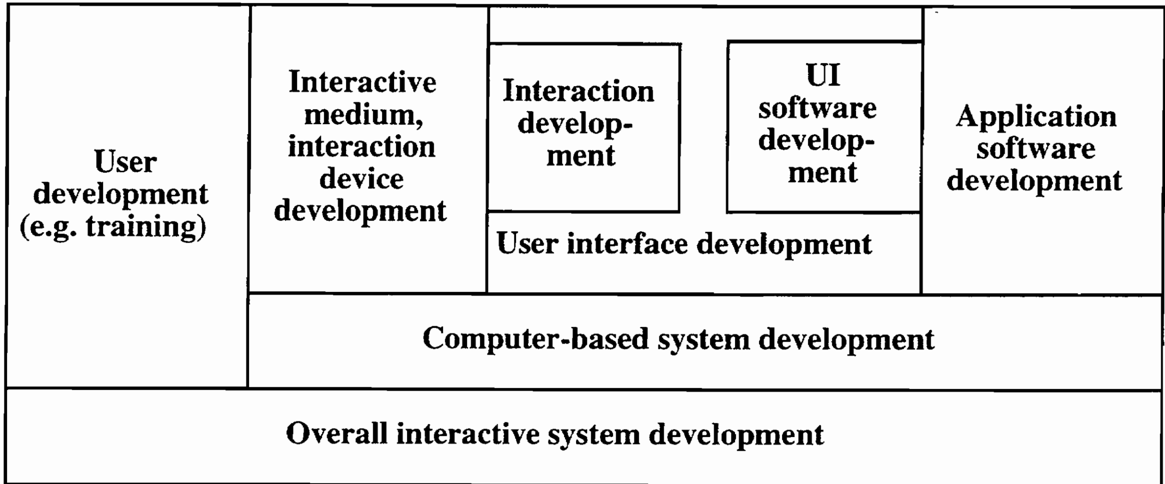


Figure 2.1. Relationships among areas of interactive system development [Hartson, Brandenburg, & Hix, 1992].

Our focus in this chapter is the highlighted area of Figure 2.1: abstract interaction development and its relation to user interface software development, in the context of user interface development in general.

2.1.2 Processes, boundaries, and communication

Our view of the development of interactive systems parallels the traditional software engineering view of software development (Figure 2.2). In this traditional view, systems analysis generates requirements for software design — the definition of modules, data structures, data flow, and operations. The software designer provides feedback to the systems analyst by reporting any inconsistencies, omissions, or ambiguities in the requirements, and through verification that the design meets the requirements.

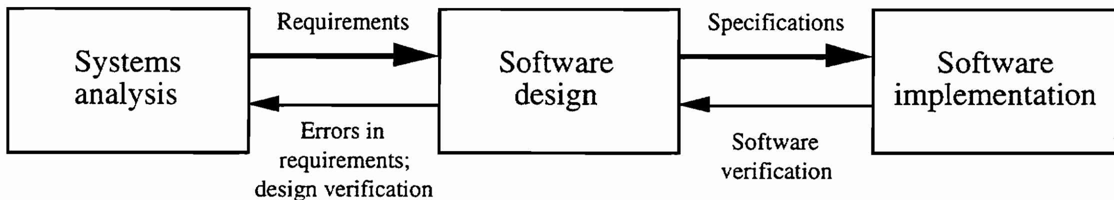


Figure 2.2. Parts of software system development [Hartson, Brandenburg, & Hix, 1992].

The software design in turn provides specifications for software implementation. The designer can receive the same sort of error feedback as described above; in addition, program verification techniques are available to confirm that the implementation meets the design specifications. The communication between these two parts of the development process has been the subject of much software engineering research, and is relatively well-understood.

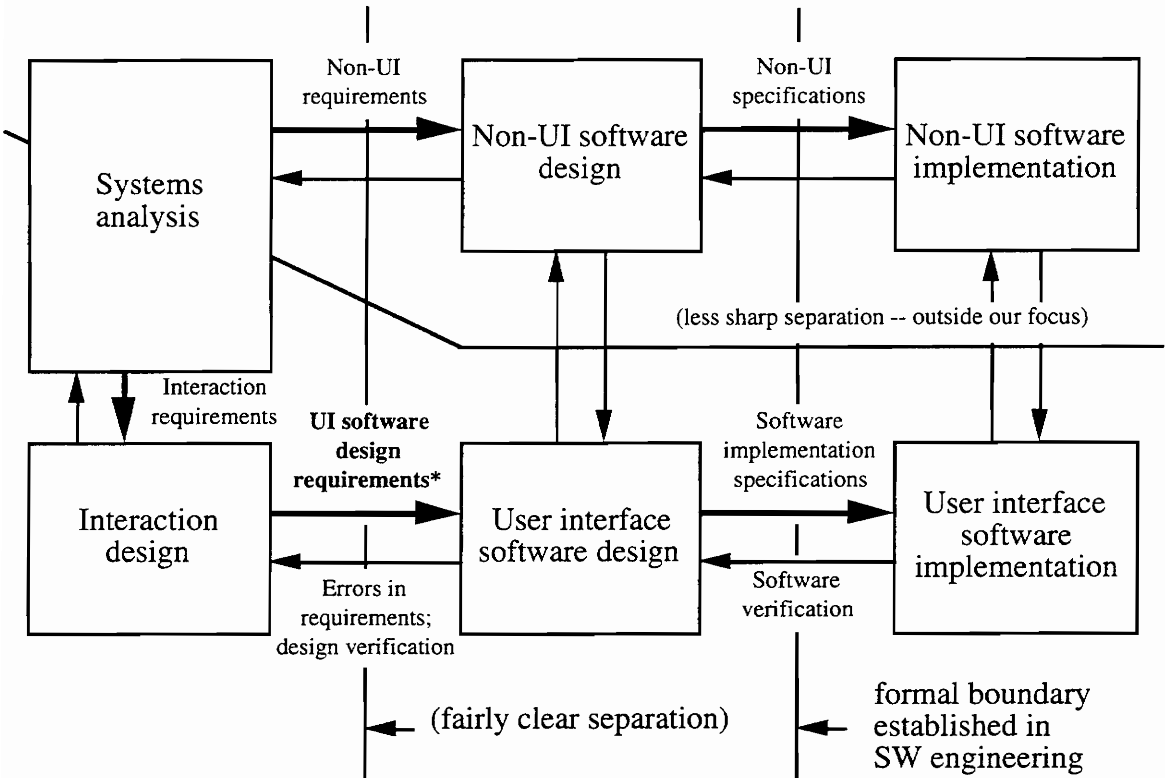
The notion of separation of roles between the analyst and the software designer, and between the designer and the implementer, is very important. Clearly, the roles of analyst, designer, and implementer require different skills and knowledge, pertaining both to the development process in general and to the specific problem domain.

The division of labor and corresponding specialization of roles afforded by this kind of separation is important to support iteration within the development process. For example, if a software design is incomplete, it often seems simplest to fill in the missing details during implementation. This casts the implementer into the role of designer, however, for which he or she is not necessarily qualified, and for which he or she does not have access to the applicable design conventions, rules, and standards.

It also seems clear that as the size of a system under development and the number of people working on it increase, isolation among separate roles reduces side-effects and unwanted dependencies. But even for small systems, and even when only one person is involved, separation among the analysis, design, and implementation roles supports a more disciplined and organized development process. Even if the design and implementation are done *by the same person*, an ad hoc approach to filling in gaps can lead to inconsistencies; at the very least, it is more likely to lead to duplicated effort and problems with maintenance.

Keeping the roles of designer and implementer separate makes it easier to verify that a design is complete and correct and that the implementation satisfies that design. Keeping the roles of analyst and designer separate makes it easier to verify that the systems analysis is complete and that the requirements accurately reflect it. Applied rigorously to the entire development process, these principles of separation can lead to better designs,

better code, and a better product. (These are not new ideas; see, for example, [Parnas, 1972b], [Parnas, 1972a], and [Parnas, Clements, & Weiss, 1985].)



* Behavioral representations to convey these user interface software requirements are the focus of our research.

Figure 2.3. Parts of interactive system development [Hartson, Brandenburg, & Hix, 1992].

Figure 2.3 expands the view of Figure 2.2 to describe the entire interactive software development process. It reflects the traditional divisions among requirements, design, and implementation, as well as the less well-defined division between interface and non-interface development.

In the analysis phase, the distinction between the user interface and the rest of the system is not necessarily as pronounced as it is during later stages of development. In an *interactive* system, the *user* is an integral part of the overall system's function, and so user activities must be considered on an equal footing with system activities. The

products of the systems analysis process include a description of the functions to be performed by the system, functions the user and system together can perform, and information items the system can present to or request from the user. These serve as requirements for the non-user interface functions to be performed by the system and for the design of the interactions that constitute the user interface.

Interaction design (Figure 2.3) determines the abstract form and content of the user interface. Since interaction design focuses on the way the user perceives and manipulates the system, the interaction design process must rely on behavioral representation — a depiction of the interface in terms of user actions and observable system presentations or responses. This depiction may include tasks represented in the interface, the kinds of user actions that form the vocabulary of the interface, visual representations, interaction styles, human factors considerations, temporal aspects of tasks, and so forth. In later chapters, we address techniques for analyzing and translating behavioral representations; historically, though, their main purpose has been to state the requirements for the user interface software.

On the other hand, constructional techniques are required for user interface *software* design (Figure 2.3), which determines the data structures, control paths, data flow, and other features of the user interface software. Since this design area deals with software rather than user and system actions, the traditional tools of software engineering can be applied. Of course, there are unique language issues which do not arise in non-interactive software design; these issues, and approaches to their solution, constitute another wide area of research (see, for example, [Myers, 1992]).

The relationship between the interaction design and interface software design processes (the lower path in Figure 2.3) parallels that of the systems analysis and non-interface software design processes (the upper path in Figure 2.3). Interaction design provides requirements for the interface software design to fulfill (see asterisk in Figure 2.3). These requirements, since they describe the behavior of the user and the interface, must be expressed in behavioral terms. The interface software design process can provide corrective feedback to the interaction design process if it uncovers omissions, ambiguities, or inconsistencies in these requirements. Further, the interface software design can guide construction of a prototype, which can then be evaluated for usability as

part of the iterative development paradigm. While it is possible to generate a prototype from the interaction design, this will only provide a shallow facade; prototypes incorporating information from the software design process can provide a more accurate and complete representation of the final system, which can lead to more efficient and effective evaluation.

The user interface software design process produces specifications for implementation of the user interface software. The relationship between these two development arenas is precisely the same as that between conventional software design and implementation, and the same techniques can be used to communicate between them.

The boundary between interface and non-interface development is not nearly as clear as the boundaries among analysis, interaction design, software design, and software implementation. First, systems analysis must straddle the border between the interface and non-interface parts of the system. Systems analysis, by definition, is concerned with the entire system. And since the user is an integral part of an interactive system, the systems analysis process must consider the user's actions and perceptions as well as the computer-based aspects of the system. Effectively, the systems analysis process must embrace all of Figure 2.1.

In addition, while some degree of separation between interface and non-interface *software* is desirable, the border drawn between them is often arbitrary. As the semantic processing associated with each user action or system presentation becomes more complex, non-interface logic can creep into interface parts of a system. If we guard against this, then functionality concerned only with the interface begins to creep into the non-interface software. This problem has been considered at length elsewhere (for example, [Hartson, 1989]).

During the *interaction* design process, these issues of software architecture should be of no concern to the interaction designer, because they are (or should be) invisible to the user. The interface is all that the user sees, and as long as that interface fits its design requirements, it makes no difference how the interface is implemented. The application might be written in Smalltalk to follow the Model-View-Controller paradigm, or it might be written in C with randomly sprinkled calls to X library routines, or it might be built

from tiny gears and springs; in fact, all three approaches have led to successful implementations of an analog clock display.

But even though they do not pertain directly to behavioral representations, it certainly *is* important to provide language features, software architectures, and development techniques to help maintain a disciplined relationship between user interface software and non-interface software. The true importance of support for interface software design and implementation becomes apparent during the process of iterative refinement. If evaluation reveals that some aspect of a user interface must be changed, the usual body of software engineering experience makes it obvious that a well-structured design and implementation of the interface software will be easier to modify than a poorly-structured implementation. Further, the better the separation between interface and non-interface software, the less deeply changes to the interface design will propagate through the design of the rest of the system. And the less time and effort it takes to modify the interface implementation, the more iterations will be possible within a given time, and (ideally) the higher the quality of the final application.

2.1.3 Interaction design as a separate role

In the previous section, we discussed the separation of roles in the development of non-user interface software (the top half of Figure 2.3). Many of the same kinds of issues arise in the development of the user interface (the bottom half of Figure 2.3).

The difference between the skills of an interaction designer and those of a software designer (even an interface software designer) is perhaps even more pronounced than the difference between systems analysis and software design skills. Experts on interaction design and evaluation should not need to know about software design, and software experts should not need to know about interaction issues. If a suitable method of communication between the two roles is available, it becomes unnecessary for designers to have expertise in both of these two essentially unrelated fields.

Further, the visibility of the user interface leads to an additional argument for the separation of development roles. As in the case of non-user interface software, both the interaction design process and the interface software design process can yield incomplete,

ambiguous, or inconsistent specifications. Without a clear separation of roles, the missing parts of the design might be completed in an arbitrary and ad hoc manner, each implementer supplying little pieces of design without a way to ensure consistency with the other pieces. The non-interactive part of the software system may be able to absorb a certain amount of this undisciplined design activity without outwardly visible negative effects on the end product, but the uniformity and internal consistency of the code may suffer, a detriment to software maintenance.

In the user interface, on the other hand, even small details can be visible to the user of the final product. When different interface software designers or implementers resolve the same interface ambiguity in different ways, the result can be an inconsistent and unsatisfactory interface.

If the interaction design process, the interface software design process, and the interface software implementation process are conducted separately, interface software implementers are less likely to resolve incomplete interaction specifications arbitrarily and inconsistently. Instead, the interface implementer feeds back a complaint to the interface software designer; if the interface software designer determines that the fault is in the interaction specification, then he or she returns the problem to the interaction designer, who can resolve it correctly.

This advantages of this approach seem obvious, but it is often ignored, or viewed as unnecessary and burdensome. This may be true partly because a large share of highly interactive interface designs are still for relatively small systems, and there is a perception that small systems allow a less formal and less structured development process. Many experimental languages, support environments, models, and tools are tested with these relatively small applications — paint programs, text editors, or games, mostly developed by an individual or a handful of people over a few months or years. While many interesting phenomena and useful achievements can be demonstrated in this realm, it does not necessarily scale to the real-world realm of large-scale, multi-user, multi-function integrated software systems involving hundreds of person-years and millions of lines of code. In this arena, separation of development roles is a matter of survival; software engineering researchers and practitioners have known this all along, and human-computer interaction researchers and practitioners must keep it in mind.

Developing large systems necessitates defining and delimiting the separate roles in the development process. Developing usable interactive systems requires an iterative development methodology that allows successive improvements to the user interface. Since this means the different roles must communicate repeatedly, we need some representation technique that can communicate designs from one stage to the next. As stated previously, communication between the interface software designer and the interface implementer is within the domain of software engineering. Communication between the interaction designer and the interface software designer (the path marked with an asterisk in Figure 2.3) must express behavioral concerns, though, and so it requires new and different techniques.

2.1.4 Design representations

The difference between behavioral design representations (for interaction design) and constructional design representations (for user interface software design) can be difficult to sort out, because there are so many different kinds of representation techniques, each for a different purpose. Each type of representation uses its own perspective to describe the same thing: what is happening in the user interface. For example, suppose the user clicks the mouse button when the cursor is on an icon. A behavioral view sees this as a user action within a task, but in a constructional view this is an input event received by the system, and in an implementation view this can be seen as something that fulfills a condition that triggers a function within a toolkit widget. When the icon is highlighted, it is seen as perceptual feedback in the behavioral view, and system response output in the constructional view, due to a default function or perhaps a callback in the implementation view.

A storyboard scenario is usually thought of as behavioral, because it depicts a procedure performed by the user. On the other hand, a state transition diagram is constructional because it is based on a view of interaction that casts the system in a role of waiting in some state for an input which, when received from the user, causes a state change. In the past the most common interface representation techniques have been constructional (for example, [Green, 1985]; [Green, 1986]; [Hill, 1987]; [Jacob, 1985]; [Jacob, 1986]; [Olsen

& Dempsey, 1983]; [Sibert, Hurley, & Bleser, 1988]; [Wasserman & Shewmake, 1985]; [Yunten & Hartson, 1985]).

Task analysis methods are behavioral and, therefore, have the potential to be used for design representation. However, they were not originally intended for this purpose, and would require some adaptation to be suitable. For example, hierarchical task decomposition used in task analysis does not usually carry procedural or temporal information. Operation sequence diagrams represent only sample instances of interaction. Scenarios are usually employed only informally to get an early impression of look, feel, and behavior, but they could be modified to play a more formal part in design representation [Hartson, Hix, & Kraly, 1990]. Other behavioral representation schemes include GOMS [Card, Moran, & Newell, 1983], the Command Language Grammar [Moran, 1981], TAG [Green, 1989], the keystroke model [Card & Moran, 1980], action grammars [Reisner, 1981], and the work of Kieras and Polson [Kieras & Polson, 1985]. These, however, are intended more for analysis (e.g., predicting user performance of existing designs), than for capturing designs as they are developed. Nonetheless, some of these techniques have seen some use for behavioral design representation (e.g., GOMS).

2.2 The User Action Notation

The User Action Notation (UAN) [Hartson, Siochi, & Hix, 1990] is one successful response to the need for a behavioral design representation. The UAN is a behavioral, user-oriented, task-oriented notation that describes the behavior of the user and the interface during their cooperative performance of a task. The primary abstraction of the UAN is a *user task*. A user interface is represented as a quasi-hierarchical structure of asynchronous tasks, that is, tasks with independent sequencing. User actions, corresponding interface feedback, and state information constitute the lowest level. As actions are composed into tasks at higher levels, abstraction hides these details, but additional notations for higher-level feedback and state changes can be added. At all levels, user actions and tasks are composed via temporal operators which represent sequencing, interleaving, concurrency, and other allowable arrangements of user behaviors in time.

A simple example illustrates the basic appearance of UAN behavioral descriptions. Consider the task of deleting a file using the Macintosh Finder, which represents files with icons and allows the user to perform file operations by manipulating the corresponding file icons. To delete a file, the user drags the file's icon onto another icon representing a trash can ("dragging it into the Trash"). The file's icon disappears, and (eventually, unless it is rescued) the file itself is deleted. We can describe this task in prose as:

(1) Move the cursor to the file icon. Press and hold down the mouse button. The file's icon will be highlighted, indicating that the file is selected.

(2) With the mouse button held down, move the cursor. An outline of the icon follows the cursor as you move it around.

(3) Move the cursor over the Trash icon. The Trash icon will highlight.

(4) Release the mouse button. The Trash icon will change (to a "stuffed can"), and the file icon will disappear.

The UAN description of these steps is:

(1) **~[icon]Mv**
(2) **~[x,y]***
(3) **~[Trash]**
(4) **M^**

In the first line, **~[icon]Mv** indicates moving (~) the cursor into the context of the file icon (**[icon]**) and pressing the mouse button (**Mv**). In the second line, **~[x,y]** indicates moving the cursor to some arbitrary position, and * (Kleene star indicating iterative closure) indicates that this can be repeated any number of times. In the third line, **~[Trash]** indicates moving the cursor to the context of the Trash icon, and the fourth line (**M^**) indicates releasing the mouse button.

Of course, this describes only user actions; it provides no indication of feedback, state changes or connections to non-interface application logic. To represent these aspects of system behavior, the UAN adopts a tabular representation with four columns, one for each aspect:

Task: Delete File			
User Actions	**Interface Feedback**	**Interface State**	**Connection to Computation**
~[icon] Mv	icon!	selected = file	
~[x,y]*	outline(icon) > ~		
~[Trash]	outline(icon) > ~; Trash!		
M^	erase(icon); erase(outline(icon)) ; Trash-!; Trash!!	selected = none	mark file for deletion

Figure 2.4. UAN description of Delete File task.

In the first line, **icon!** indicates that the file icon is highlighted, and **selected = file** indicates that the interface's notion of "the currently selected object" should be updated to indicate that the file is now selected. In the second line, **outline(icon) > ~** indicates that an outline of the icon follows the cursor's movement. In the third line, the icon outline continues to follow the cursor, and **Trash!** indicates that the Trash icon is highlighted. In the fourth line, **erase(icon)** and **erase(outline(icon))** have the expected meanings, **Trash-!** indicates that the original highlighting of the Trash icon is removed, and **Trash!!** indicates an alternate highlighting mode for the Trash icon — in this case, an "inflated" appearance. **selected = none** indicates that there is no currently selected object in the interface. **mark file for deletion** in the fourth column indicates an action to be performed outside the interface, and thus serves as a connection to the non-interface part of the application.

In the Interface Feedback column of the last two rows, we separate activities by semicolons to indicate that they take place in sequence; we could have listed each activity in a separate row instead, but the notation presented is more concise and legible. We adopt the semicolon as a sequence separator for consistency with the literature at large; others have used the semicolon for a special purpose, such as indicating interruptible points [Siochi, Hartson, & Hix, 1990], and have used the comma to indicate sequence.

As this example indicates, the UAN does not address all aspects of interface design; it says little about the physical appearance of interface items, and does not necessarily provide a compact representation of interface modes. Other behavioral description techniques, particularly scenarios, storyboards, and task transition diagrams [Siochi, Hartson, & Hix, 1990], exist to represent these aspects of the interface, and can be used to complement the UAN in a complete behavioral design.

In the realm of description of user behavior, though, the UAN is a powerful tool. The UAN is more concise and precise than natural language, and users report that it is easy to learn to read and write [Hartson, Siochi, & Hix, 1990]. In fact, the UAN is currently being used as a part of the user interface development process at a number of external sites.

2.3 The case for a behavioral model

While the UAN has proven useful in the development process, it is *only* a notation; it has no formal underlying model of interaction. The UAN developed in an ad hoc fashion as a tool for internal communication of interface designs. It has served well in this capacity, and as previously noted, it has proven useful in many different settings and with many different interaction styles. But as it has been further extended and exercised, certain recurring problems and weaknesses have appeared.

2.3.1 Problems of definition

The UAN provides separate columns for user actions, system feedback, interface state, and connection to computation. It is generally understood that user actions are ordered from left to right and top to bottom, that system feedback appears after the associated action is performed, that interface state changes occur at some point after the user action, and that connections to computation occur at an "appropriate" time. But depending on general understanding is hardly a recipe for reliable communication, and the absence of a model has made formal definition of these orderings difficult.

A UAN task description can be preceded by a *viability condition* that must be satisfied for the task to be eligible for performance. These viability conditions are usually

specified in terms of some aspect of interface state, but the connection is not well-defined, and so formal consideration of viability conditions is not possible.

In fact, the concept of "interface state" is itself not well-defined. Concepts generally represented as part of interface state include selected items in the interface, the highlighting status of interface objects, and representation of "current" items. But all such references are at the whim of the UAN writer, and so reasoning about the state of the interface can be very difficult.

Further, there are certain aspects of interface state that have so far remained implicit in most UAN descriptions, but are very important to system behavior. These include the state of individual keys (pressed or released) and the location of the mouse cursor (within or outside the context of an interface object). In many cases, these aspects of state are involved in *implicit viability conditions* — for example, the user cannot press a key if it is already pressed, or leave the context of an icon if the cursor is not already in that context.

All these problems are related to formal definition of the UAN. Efforts have been made to define certain aspects of the UAN, most notably temporal issues [Hartson & Gray, 1992], but so far no definition has been complete enough to encompass the issues above.

2.3.2 Connection to a formal model of input devices

One of the reasons the UAN is popular is because it is extensible, and one of the ways it is being extended is through addition of new interaction devices. Some of these devices (for example, trackballs or foot pedals) are so similar to familiar devices (mice or keys) that creation of new notation for them is trivial. For less familiar devices, it is usually possible to invent new notation in the same ad hoc spirit that engendered the original UAN. But for the sake of portability, understandability, and consistency, it would be helpful to have a systematic way to add new devices.

[Card, Mackinlay, & Robertson, 1990] presents a formal model for input devices and claims that it is sufficiently general to represent nearly all existing devices. Devices are mapped into a space with dimensions corresponding to degrees of freedom, types of motion, and various other characteristics. A behavioral model that is compatible both

with this device model and with the UAN is an important step toward providing a systematic method for creating UAN descriptions of new devices and their actions.

2.3.3 Analysis of behavioral descriptions

Behavioral descriptions of an interactive system, including UAN descriptions, begin to emerge at a very early point in the development process. Catching problems at this early stage makes them easier and quicker to solve, since it reduces the length of the channels along which changes must percolate (Figure 2.3) and the amount of work that must be undone. Thus, analytical techniques that can catch problems likely to exist in early stages have great potential value.

It should be possible to reduce these analytical techniques to algorithms and automate them. But algorithms need data structures. To define "data structures" representing behavioral phenomena, we need a behavioral model.

The UAN's hierarchical task structure gives rise to certain other problems when tasks share common prefixes — when a sequence of user actions matches several eligible tasks. Highly interactive systems such as direct manipulation interfaces require quick system responses as feedback to user actions, and system designs with tasks that share common prefixes can exhibit feedback conflicts. These conflicts are not immediately apparent in UAN descriptions, but a behavioral model can make them easier to find.

2.4 Our response: Timetrees

We have developed the timetree model in response to our experience with the UAN, problems that arise in UAN descriptions, and capabilities that we would like to see in future behavioral representations. In the next chapter, we informally introduce the timetree model, presenting examples to illustrate its relationship to UAN task descriptions and demonstrate some of its capabilities.

3 An Introduction to Timetrees

In the previous chapter we discussed the nature of behavioral notations, concentrating on the UAN, and the need for a model to support these notations. In this chapter we provide an introductory overview of our model for behavioral phenomena. This model is based on the *timetree*, a tree structure representing all possible behaviors of a system and user interacting over time. The model incorporates tasks, user actions, system activity, and system and interface state into a framework that reflects their ordering over time. User or system activities are represented by arcs; these arcs connect nodes representing abstract states of the human-computer system. Components of interface, system, or user state can be associated with these abstract states.

In Section 2.2, we introduced the UAN with the aid of a UAN description of a simple task. In this chapter we introduce the properties and applications of timetrees by modeling and expanding upon this simple example. Timetrees representing UAN task descriptions introduce the general techniques that we will use for a formal definition of the UAN in Chapter 6. An example involving a branching timetree hints at some "task interaction phenomena" to be revisited in Chapters 6, 8, and 9. Finally, examples of composition, abstraction, state representation, and equivalence illustrate many of the components and operations formalized in Chapter 5 and used in the chapters that follow.

3.1 A single task; a simple timetree

Recall the task of deleting a file using the Macintosh Finder, as described in Section 2.2. Figure 3.1 reiterates the UAN description of this task, with one small change: the actions of moving to the context of an icon (**~[icon]**) and pressing the mouse button (**Mv**) are separated for clarity. (Again, we use the semicolon to separate sequential actions.)

Task: Delete File			
User Action	**Interface Feedback**	**Interface State**	**Connection to Computation**
~[icon]			
Mv	icon!	selected = file	
~[x,y]*	outline(icon) > ~		
~[trash]	outline(icon) > ~; trash!		

M^	erase(icon);
	erase(outline(icon));
	trash-!;
	trash!!
		selected = none	mark file for deletion

Figure 3.1. Modified UAN description of Delete File task.

Figure 3.2 represents a timetree describing user actions that constitute the Delete File task. Each node is a *timestate*, an abstract state of the interacting system and user. Each arc is labeled with an *activity* (in this case, a user activity). The first timestate, describing the state of the system and user before the first activity, is called the *initial timestate*; the last timestate, reached after the last activity, is called a *terminal timestate*. We represent terminal timestates by double circles.

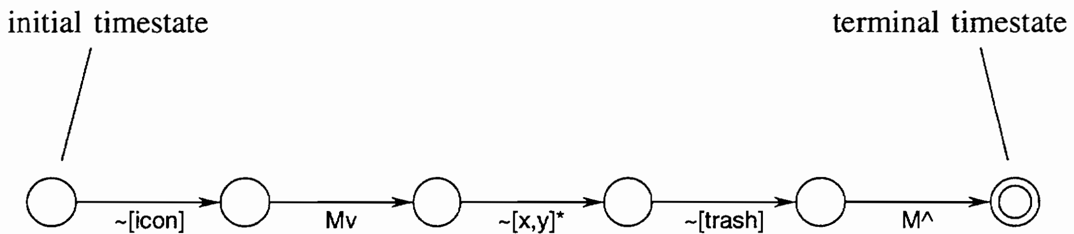


Figure 3.2. Timetree for Delete File task.

To traverse the timetree from initial timestate to terminal timestate, one follows each arc in sequence. The sequence of arc labels in this traversal corresponds to the sequence of activities specified by the UAN description of Figure 3.1. Note that there is a one-to-one correspondence between user actions in the UAN description and arcs in this timetree. Note also that the nodes in the timetree do not correspond directly to anything in the UAN description; they simply "mark places" to help define the topology of the tree.

Of course, a tree this simple is not very interesting. In fact, the task description in Figure 3.1 is also not very interesting, because it does not allow the user to make any choices — it prescribes only one possible course of behavior. In a broader view of the Macintosh Finder, the user can choose among a number of alternative tasks. We next examine how the addition of some of these tasks affects the structure of our timetree.

3.2 A set of composed tasks; a branching timetree

Instead of deleting a file, the user may choose to select the file or move the file's icon within its window. To select the file, the user moves the cursor to the file's icon and presses and releases the mouse button; to move the icon, the user moves to the icon, presses the button, moves the mouse, then releases the icon. Figures 3.3 and 3.4 present UAN descriptions for these tasks.

Task: Select File			
User Action	**Interface Feedback**	**Interface State**	**Connection to Computation**
~[icon]			
Mv	icon!	selected = file	
M^			

Figure 3.3. UAN description of Select File task.

Task: Move File Icon			
User Action	**Interface Feedback**	**Interface State**	**Connection to Computation**
~[icon]			
Mv	icon!	selected = file	
~[x,y]*	outline(icon) > ~		

M^	erase(icon);
	erase(outline(icon));
	display(icon)
	@[x,y];
	icon!		

Figure 3.4. UAN description of Move File Icon task.

We indicate that the user can choose among these three tasks by composing them with the UAN choice operator (|). For future reference, we name this composite task "Manipulate File". Figure 3.5 presents the UAN for the task.

Task: Manipulate File			
User Action	**Interface Feedback**	**Interface State**	**Connection to Computation**
Select File | Move File Icon | Delete File			

Figure 3.5. UAN description of Manipulate File task.

Timetrees represent choice with branching structures; the timestate at which a choice of activities is available has several arcs leaving it, one for each activity. Thus we can model the structure of the Manipulate File task as follows:

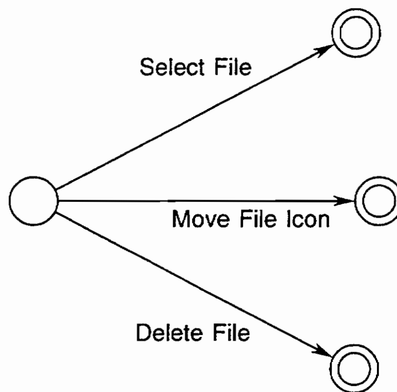
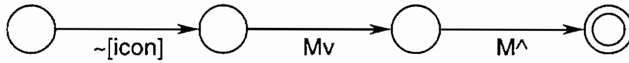


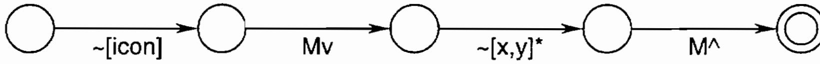
Figure 3.6. Timetree for Manipulate File task.

In this timetree, each component task is represented by a single arc; the internal structure of each task is abstracted away, just as it is in the UAN description of Manipulate File (Figure 3.5.). It is possible to expand this timetree back to the level of abstraction shown in Figure 3.2, but this expansion is more involved than simply replacing each arc with its task's expansion.

Select File:



Move Icon:



Delete File:

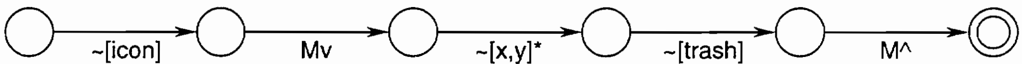


Figure 3.7. Timetrees for Select, Move Icon, and Delete File.

Consider the timetrees shown in Figure 3.7. Note that each begins with the same activities (**~[icon] Mv**), and that Move Icon and Delete File also share the next activity (**~[x,y]***). We call such sequences *common prefixes*, since they are shared by multiple tasks that are available at a particular point in a dialogue. Thus, **~[icon] Mv ~[x,y]*** is a common prefix of Move Icon and Delete File, and **~[icon] Mv** is a common prefix of all three tasks.

Joining these timetrees by combining their initial timestates yields the tree of Figure 3.8:

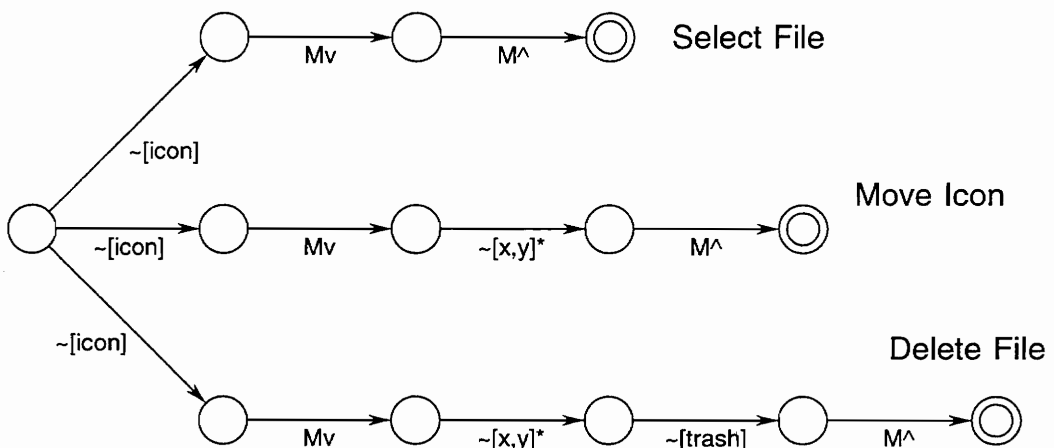


Figure 3.8. Simple combination of timetrees.

This timetree is *ambiguous*: it contains multiple paths starting from a single timestate that match the same sequence of actions. Specifically, in this case, the sequence of actions **~[icon] Mv** matches three different paths from the initial timestate. In this sense the timetree resembles a nondeterministic finite-state automaton (NFA). But, just as any NFA can be transformed into a deterministic finite-state automaton, any ambiguous timetree can be *disambiguated*, or turned into a timetree in which no action sequence matches multiple paths from the initial timestate. For the Manipulate File composite task, we can construct a timetree like that of Figure 3.9:

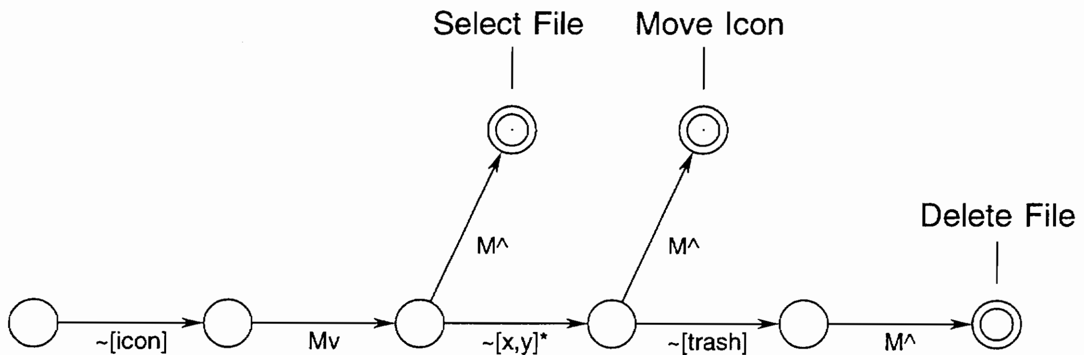


Figure 3.9. Disambiguated timetree.

This tree has the same number of terminal timestates as the trees of Figures 3.6 and 3.8, and as the labels indicate, each terminal timestate corresponds to the completion of one of the three possible tasks. The path from the initial timestate to a terminal timestate yields the same sequence of activities as the original timetree for the corresponding task.

The resemblance of timetrees to finite-state machines is not surprising. A behavioral interface design specifies the sequences of activities that constitute tasks, like a grammar specifies the sequences of tokens that constitute strings. But interfaces differ from parsers or lexical analyzers in that they often do not have the luxury of looking ahead to resolve ambiguous input. For example, in a direct manipulation interface, it is often not acceptable to collect an entire task's worth of user actions before issuing a response; for *manipulation* to be *direct*, an interface object must immediately respond to an action directed toward it. In such situations it is necessary to specify a predictable and consistent response for each user activity; disambiguated timetrees like that of Figure 3.9 represent such a specification.

This ability to specify immediate and unambiguous system responses to user actions is one of the main motivations for the development of the timetree model. To address the issue more directly, we must introduce system activities to our timetrees.

3.3 Low levels of abstraction; system activities

As we stated earlier, timetrees can represent system activities as well as user activities. The timetrees we have seen so far represent only user activities, but the UAN descriptions of their associated tasks include system responses, state changes, and connections to computation. We can add these additional activities to the timetree of Figure 3.9 to yield Figure 3.10.

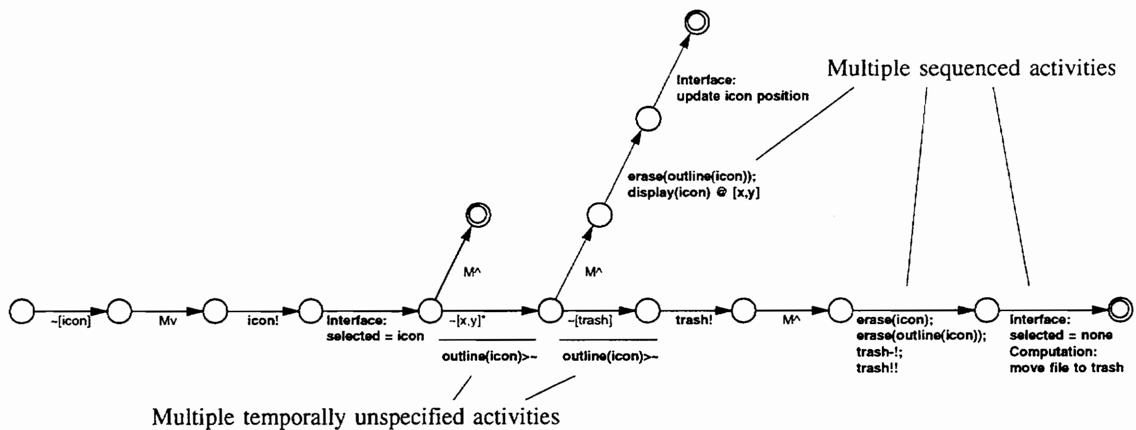


Figure 3.10. Timetree with system activities.

Note that the overall topology of this tree is identical to that of Figure 3.9. While it contains many added arcs and timestates, it still has one initial and three terminal timestates, corresponding to the three original tasks. This makes sense, since it is user decisions that determine which user activity takes place at the branching points; in this design, the system must respond predictably and consistently to each of the specified user actions.

Note that a number of arcs are labeled with several activities. On most arcs, these activities are separated by semicolons, indicating (as in the UAN) that they take place in sequence. This is a notational convenience; it would be possible to draw out a sequence

of separate arcs, one for each activity, separated by new timestates, but this would contribute little to clarity or legibility. Likewise, it is only for clarity and convenience that state changes and computational connections (**selected = file**, **move file to trash**) are segregated from visible system responses.

Two arcs, though, are labeled with a pair of activities separated by a dashed line. These arcs correspond to the portion of the UAN in which an icon outline moves to "follow" the motion of the cursor. Although the UAN does not explicitly indicate it, the definition of **icon > ~** specifies that the icon's movement takes place *during* the movement of the cursor, rather than *after*. In an actual implementation, the outline would presumably move in small increments in response to small incremental mouse movements; this design, though, abstracts away this level of detail, and so we use a corresponding abstract representation in the timetree. The dashed-line notation indicates that this timetree *does not specify* the temporal relationship between the two activities; to specify it, we require that the arc be expanded to a lower level of detail. The notion of true concurrent activities is another matter, and we address it formally in Chapter 5.

3.4 Ambiguities and design conflicts

In some system designs, a single sequence of user actions might match several different tasks. This is the case in the first few arcs of Figure 3.9; the user, having performed **~[icon] Mv**, might be performing any one of the three tasks that compose Manipulate File. This ambiguity constrains the system design; since the system cannot determine which task the user is performing, it must provide the same responses to these activities for all three tasks.

It is easy to create designs that violate this constraint. For example, consider a new version of the Manipulate File task, one that allows the user to delete a file by dragging it to the trash or copy a file by dragging it to the icon of a floppy disk. To provide more direct and meaningful feedback for cursor movement, the new Delete File task specifies that the file icon itself (rather than an outline of the icon) follows the cursor; for the new Copy File task, a copy of the file icon appears and follows the cursor. Figure 3.11 presents UAN descriptions for these tasks.

Task: New Delete File			
User Action	**Interface Feedback**	**Interface State**	**Connection to Computation**
~[icon]			
Mv	icon!		
~[x,y]*	icon > ~		
~[trash]	icon > ~; trash!		
M^	erase(icon); trash-!; trash!!		mark file for deletion

Task: New Copy File			
User Action	**Interface Feedback**	**Interface State**	**Connection to Computation**
~[icon]			
Mv	display(copy(icon)); copy(icon)!		
~[x,y]*	copy(icon) > ~		
~[floppy_icon]	copy(icon) > ~; floppy_icon!		
M^	icon-!		copy file to floppy

Task: New Manipulate File			
User Action	**Interface Feedback**	**Interface State**	**Connection to Computation**
New Delete File | New Copy File			

Figure 3.11. UAN descriptions of new Delete, Copy, and Manipulate tasks.

Figure 3.12 shows a timetree describing the behavior of the new Manipulate File task. The timestate marked "?" indicates the precise location of the design conflict: at this point, the system must choose a response based on which task the user is performing, but there is no way to determine this from the user's actions so far. In effect, the system must decide which action to perform on the basis of a *future* user action, rather than the sequence of user actions so far. Metaphysical considerations make this behavior difficult to implement.

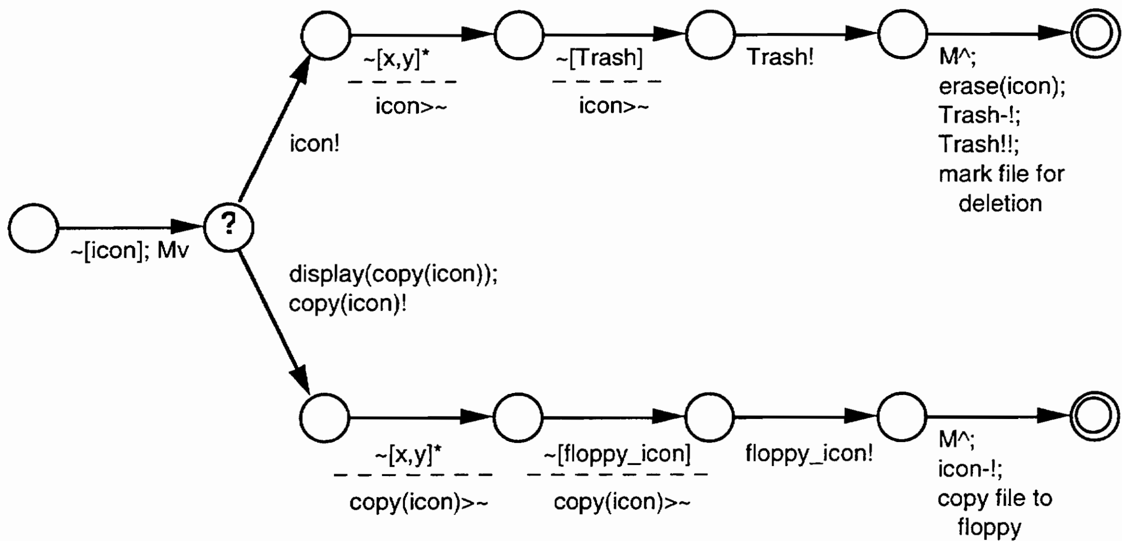


Figure 3.12. Timetree showing conflict in new Manipulate File task.

3.5 Associated state information

So far, our examples have emphasized user and system activities. Sometimes, though, we want to discuss aspects of system or user state as well. Timetrees represent state by associating state components and their values with timestates. In this way, the value of a particular component of state can be specified at particular points during the performance of a task. Since every individual activity has one timestate as its source and one as its destination, timestates also provide a natural place for state information associated with conditions of viability and postconditions — conditions that must hold before or after a task is performed.

The UAN descriptions of Select File, Move File Icon, and Delete File (Figures 3.3, 3.4, and 3.1, respectively) refer to several components of system state. For instance, the interface state column mentions a component called "selected" which can refer to a file or can be empty (**selected = none**). References to the context of an icon (**[icon]**, **[trash]**) involve the visible location and extent of the icon; these are visible aspects of state. Highlighting actions (**icon!**, **icon-!**, **trash!!**) change the visible state of icons. Even pressing or releasing the mouse button (**Mv**, **M^**) changes a component of state — in this

case, the state of the mouse button. For simplicity, we assume that there is precisely one file icon and one trash icon, and that both are visible and unhighlighted at the outset of the task.

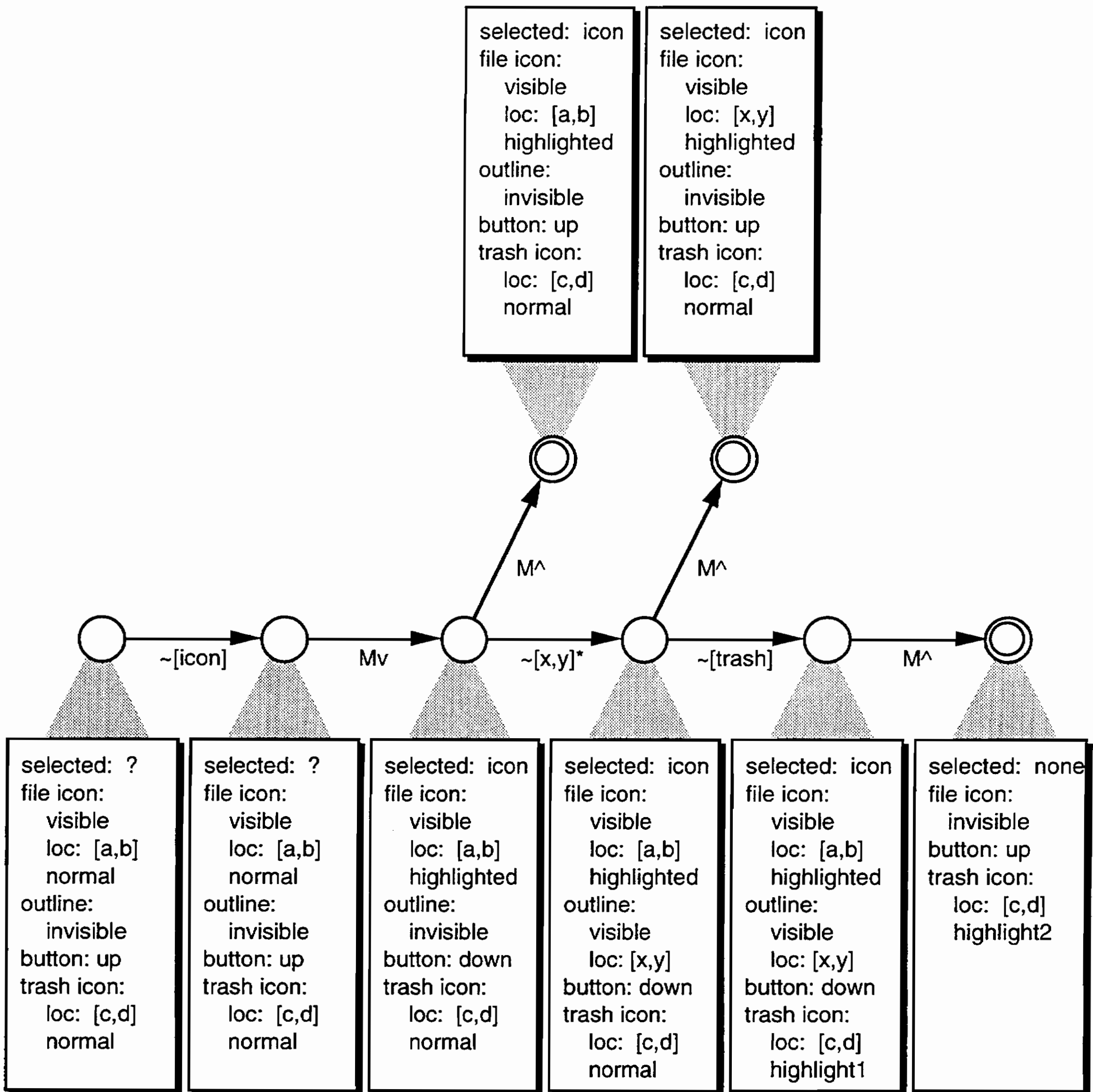


Figure 3.13. Disambiguated timetree with state information for Manipulate File.

Figure 3.13 shows a mapping from timestates to state component values for the Manipulate File timetree of Figure 3.9. Note that the value of "selected" is unknown at the first two timestates; it is not specified or restricted until later in the task, so its value at

these states is unknown and unimportant. Note also that, while each arc is labeled only with a user action, we include the effect of system actions on state; again, we hide the system activity arcs for legibility.

The set of state components illustrated in this example is not intended to be complete. There could be other state information that is not mentioned explicitly in the UAN. In addition, other state components may appear at later points in the development process. Such additional components can be treated similarly to the ones we show here.

3.6 Conditions of viability and state components

The Manipulate File task allows three subtasks: selection, icon movement, and deletion. To perform any of these subtasks, though, the user must move the cursor to the icon's context and press the mouse button; this is not possible unless the file icon is visible.

Select File and Move File Icon leave the file icon visible, so the user can perform them repeatedly. Delete File, on the other hand, leaves the file icon invisible, thus making it impossible to perform any of the three subtasks.

The UAN provides the * operator to specify that a task can be performed zero or more times. Thus we can write (**Manipulate File**)* to indicate that the user can choose among the three file manipulation tasks any number of times. But we still must specify that once the user performs Delete File, further repetitions of Manipulate File are impossible. The UAN addresses this need with *viability conditions*, predicates on some aspect of state that must be true for a task to be available.

Task: Manipulate File			
User Action	**Interface Feedback**	**Interface State**	**Connection to Computation**
file icon visible: (Select File | Move File Icon | Delete File)			

Figure 3.14. UAN description of Manipulate File incorporating viability condition.

Figure 3.14 is a UAN description of the Manipulate File task incorporating the viability condition that the file icon must be visible. To incorporate this viability condition into a timetree, we simply evaluate the condition against the state component values at the initial timestate; if the condition is met, we append the task to the timestate, and if it is not met, we do not.

Figure 3.15 illustrates this process for a timetree describing (**Manipulate File**)*. Since the * operator specifies zero or more repetitions, the resulting timetree must contain infinite paths; we explicitly represent two levels of repetition, with dashed lines indicating the continuing paths. The initial timestate is a terminal timestate because the task description allows zero repetitions. (If we expanded the timetree to include internal timestates as in Figure 3.9, those internal timestates would *not* be terminal timestates.) Finally, every occurrence of Delete File leads to a timestate in which the file icon is not visible, and no such timestates serve as the source for any further arcs.

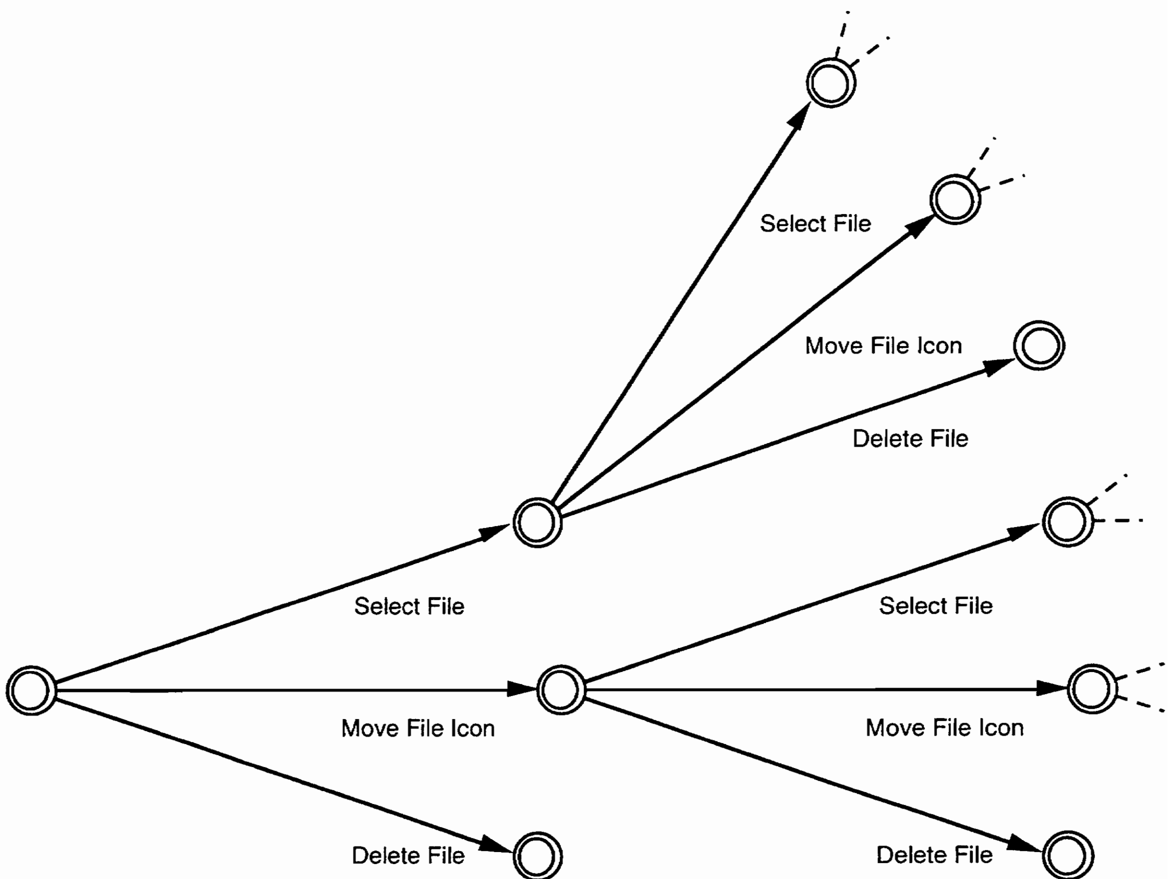


Figure 3.15. Timetree for (**Manipulate File**)*.

In fact, we make several simplifying assumptions in this example; for instance, we assume only one file icon is visible at the beginning of the task, so we do not have to represent sets of file icons, items already in the trash, and so on. While timetrees can handle all these other factors, the result would be more complex, and less useful as an illustration. We consider more details in Chapter 6, when we present our formal representation of viability conditions.

3.7 Timestate equivalence; transition diagrams

In principle, and at the most explicit level of representation, every different sequence of behavior specified by a timetree leads to a different terminal node. In practice, many of these terminal states do not present significant differences. This is reflected in the UAN; consider, for example, the Manipulate File task. In many cases, it may not matter that a file is selected or that its icon's position changes. Tasks that invoke Manipulate File as a subtask may well follow the same path afterward, regardless of which subtask was actually performed.

In situations where the exact choices of tasks or their effects on state components are not significant to the future behavior of the interface, the UAN allows the designer to abstract away these differences. A higher-level task can invoke Manipulate File without concern for its internal structure, just as Manipulate File can combine its three subtasks without concern for their internal structure.

If we consider Manipulate File as a single, abstract task, we can model it with a timetree like that of Figure 3.16. This timetree represents all the possible variations of internal behavior by one arc, labeled with one high-level task, and leading to one terminal timestate. As long as we are not concerned with state components and their values, this is acceptable.

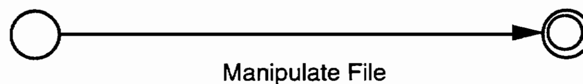


Figure 3.16. Extremely high-level timetree for Manipulate File task.

The presence of viability conditions as discussed in the previous section indicates a situation in which this representation is not acceptable. If we abstract away *all* information about state, it is impossible to evaluate viability conditions, and thus it can be difficult to construct timetrees to model these abstract descriptions. One solution is to abstract away only *some* state information — the state information mentioned in viability conditions, or especially important to the user's perception of the system — and to differentiate timestates based on the remaining, significant state component values.

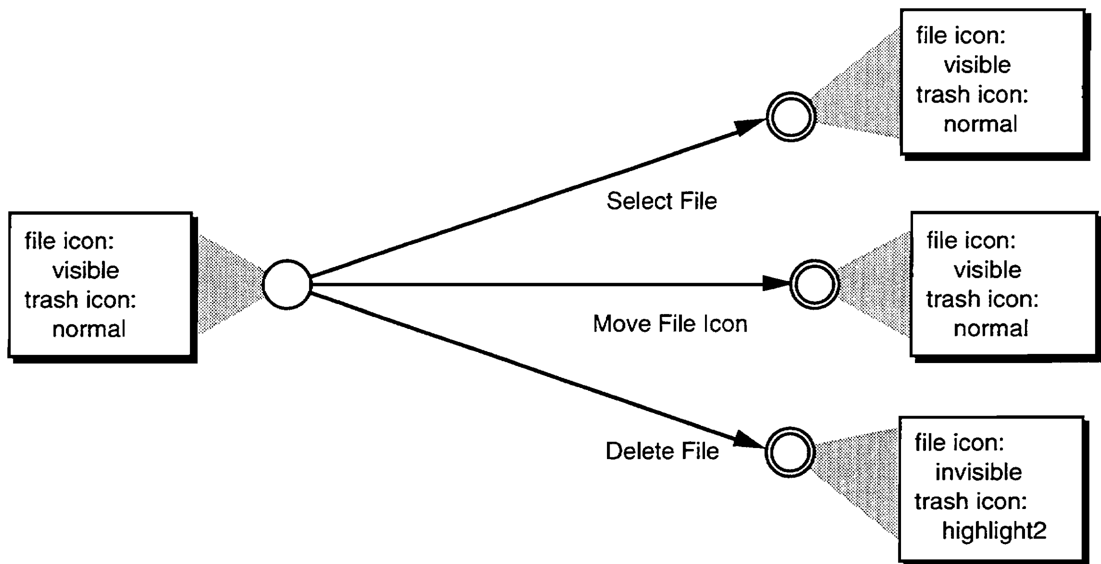


Figure 3.17. Manipulate File subtasks with significant state information.

Figure 3.17 labels timestates with some state components that significantly differentiate among terminal timestates. From this perspective, Select File and Move File Icon do *not* produce significantly different results, and so it makes sense to condense them into a single activity, as in Figure 3.18.

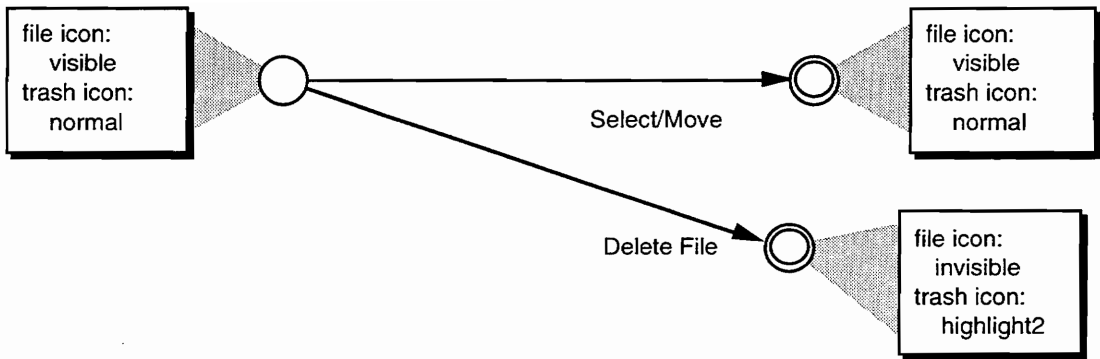


Figure 3.18. Manipulate File subtasks after timestate condensation.

This resulting timetree represents an intermediate level of abstraction, representing more detail than the timetree of Figure 3.16, but less than that of Figure 3.13. In some applications, such as those requiring reasoning about the number of icons on the screen or the status of the trash icon, this level of abstraction may be the most economical.

In this chapter we have presented an informal overview of the timetree model, its relationship to UAN task-based descriptions, and some of its applications. The next chapter reviews some related models from the literature; we return to the timetree model with a formal exposition of the model's components and operations in Chapter 5.

4 Related Work

A number of existing models, representation techniques, and systems of logic have influenced the timetree model. In this chapter, we summarize some of this previous work. Section 4.1 addresses previous behavioral representation techniques and user models. Section 4.2 describes two constructional models related to our approach. Section 4.3 reviews the device model we have chosen for input device representation. Section 4.4 addresses some varieties of branching time temporal logic, which suggested the basic structure of the timetree model. Finally, Section 4.5 discusses two models of concurrency that also influenced the development of the model, and our view of concurrency.

4.1 Behavioral representation techniques and user models

While the examples of behavioral descriptions in previous chapters have used the UAN, other task-oriented behavioral representation techniques exist. These techniques, though, are mostly oriented toward user modeling and analysis of existing interface designs; unlike the UAN, they have not seen widespread use in development of new interactive systems.

4.1.1 The keystroke-level model

The keystroke-level model [Card & Moran, 1980] represents an early attempt to predict user performance from an interface description. Given a task, a command language for a system, parameters for user motor skill and system response time, and the method used for the task, the model predicts the time an expert user making no mistakes will take to perform the task.

The model computes the expected time to perform a task by summing times for user and system actions. Physical user actions are divided into keystroking (**K**), pointing (**P**), homing (**H**), and drawing (**D**). User mental actions are represented by one mental operator (**M**). A response operator (**R**) represents system responses. Thus task execution time is represented by

$$T_{execute} = T_K + T_P + T_H + T_D + T_M + T_R$$

Each of the time terms is computed as the product of the number of individual actions (for example, keystrokes) and the time for one such action. Time per keystroke is determined empirically from typing tests. Time for pointing (with a mouse in the original study) is computed as a function of target size and distance according to Fitt's Law [Card, Moran, & Newell, 1983]. Time to home on a device is assigned a constant value from empirical studies. Drawing time is a linear function of number of line segments and total length of line segments, and again is evaluated empirically. Mental preparation time is assigned a constant value from experimental data, and system response time is specified for each system action (the model does not incorporate any theory of system response time).

Some components of the model, such as the specialized draw operator, seem to indicate that it may be difficult to generalize to other interaction styles — for example, a direct-manipulation drawing program that provides a variety of "drawing" functions. In addition, a set of heuristics is needed to determine placement of the mental action operators, although all the heuristics are based on the single psychological principle of chunking. But within these limits, the model allows accurate prediction of task execution time.

While the keystroke model is intended to predict performance from a system design, it does not incorporate any specific design representation. Instead, it generates predictions for strings of activities constituting performance of a particular task.

4.1.2 Action grammars

Reisner [1981] provides a formal, grammar-based representation of interface structure that captures "action languages" for interactive systems. Analysis of these grammar-based descriptions yields predictions about usability and learnability. Unlike the keystroke model, though, action grammars do not incorporate temporal performance information.

Action grammar representations follow Backus-Naur form (BNF), with terminal symbols representing user actions and nonterminal symbols representing groups of these actions. In addition to metasymbols for concatenation and choice of actions, she defines an operator to indicate that two actions must take place simultaneously; the definition of "simultaneous," though, is not formalized. In her examples, Reisner apparently intends that the hierarchy of tasks reflect the user's view of the system's structure, but she provides no formal method for verifying the match.

Predictive measures from action grammars are based on terminal symbol counts, lengths of terminal strings for particular tasks, and the number of rules necessary to describe a related set of terminal strings. The first measure indicates the number of different actions in the language, analogous to vocabulary size in conventional languages. The second indicates the length of the sequence of activities needed to perform a given task. The third indicates one aspect of consistency — similarity among the structures of related tasks. Reisner tests these measures on descriptions of two electronic sketching systems and finds that they are correlated with ease of learning and remembering command structures.

While action grammars appear to be useful for predicting some aspects of usability, they have some deficiencies as a design notation. They can handle only a limited amount of temporal structure; there are operators for sequence, choice, and an extremely limited degree of concurrency (the "simultaneous" operator), but none for temporal relations at a higher level. Perhaps more importantly, they provide no facility for denoting system responses, system state, or connection to computation.

4.1.3 CLG

Moran's Command Language Grammar (CLG) [1981] is more extensive and more ambitious than the keystroke or action grammar models. It seeks to represent interactive systems from the highest level of tasks to the lowest levels of display layout and device behavior. It uses a linguistic approach to reveal the structure of a user interface, a psychological approach to describe the user's mental model of the system, and a design approach for system design specification.

CLG proposes a top-down design approach. First, at the *task level*, the designer analyzes the user's needs and develops a structure of specific tasks to be carried out with the aid of the system. Next, at the *semantic level*, the designer lays out the conceptual entities and operations that the user will manipulate and perform to accomplish the tasks. These two levels, containing the abstract concepts that organize a system, constitute the *conceptual component* of a CLG design.

The next level of design, the *syntactic level*, embeds the conceptual model of the system in a language structure of commands, arguments, contexts, and state variables. The lowest level of this language must be expressed in terms of physical actions, specified in the *interaction level*. This level also specifies system actions in response to user actions. These two levels constitute the *communication component* of the design.

The lowest levels of design, the *spatial layout level* and the *device level*, constitute the *physical component* of a CLG design. Moran does not discuss these levels in detail.

CLG provides extensive and thorough support for design specification, but this very thoroughness makes it unwieldy for even small examples. In addition, it has some shortcomings; it allows specification of system responses only at the relatively low interaction level, and it does not consider temporal relationships among tasks. (Moran himself raises this latter point with respect to "interruption behaviors," involving task abandonment, errors, or help.)

4.1.4 GOMS

GOMS [Card, Moran, & Newell, 1983] is a technique for modeling the knowledge a user needs to perform tasks with a device or system. Its name is an acronym for Goals, Operators, Methods, and Selection rules, the components it uses to represent user mental operations.

A *goal* in the GOMS model represents something the user tries to accomplish. To accomplish a goal, a user may need to accomplish several subgoals; this generally leads to a hierarchical goal structure.

An *operator* is an action that a user executes. Like goals, operators may decompose into sequences of lower-level operators, until every operator is *primitive*; primitive operators are not further analyzed. Operators are divided into *external operators*, involving interaction with the system or the environment, and *mental operators*, representing unobservable mental activity (hypothesized by the designer or analyst, since it is unobservable). Some examples of operators provided by GOMS are:

Accomplish goal <goal description>
Report goal accomplished
Decide: If <operator> Then <operator> Else <operator>
Goto Step <number>

Recall that <memory-object>
Retain that <memory-object>
Forget that <memory-object>

Home-hand to mouse
Press-key <key name>

A *method* is a sequence of steps to accomplish a goal. Each step consists of an operator. *Selection rules* route control to the appropriate method when there is more than one method to accomplish a particular goal.

While the GOMS model appears to provide a detailed and precise representation of user cognitive processes, it relies very heavily on the judgement of the person performing the system analysis. If the analyst produces a GOMS model that accurately reflects the user's mental processes, it can be used to predict performance, learning time, mental workload, and other quantitative measures. But there is no way to determine how accurate such a model is; at best, its predictions can be compared against real measurements, with a mismatch indicating that there must be a problem.

The task-based high-level structure of GOMS resembles that of the UAN, but GOMS provides almost no facility for specifying system actions or temporal relations among tasks. It also, like the other techniques presented so far, assumes error-free performance.

4.1.5 Kieras & Polson's cognitive complexity theory (CCT)

Kieras and Polson [1985] seek to represent the complexity of an interactive system from the user's point of view. They develop a formal representation of user knowledge about a particular device, and a separate formal representation of the behavior of the device itself. They use these two representations to simulate user-device interactions and generate measures of cognitive complexity.

User task knowledge is represented using the GOMS model. Device behavior is represented by *generalized transition networks* (GTNs). GTNs augment ordinary transition networks by adding nesting, recursion, backtracking, and conditions on arcs. Arc conditions can test and manipulate an arbitrary number of registers with arbitrary capacity, thus yielding Turing-equivalent power.

Kieras and Polson evaluate the mapping of tasks to a device by drawing a correspondence between the hierarchical goal structure of the user's task knowledge and the hierarchical network structure of the device. If there exists an isomorphism between the goal structure graph and a subgraph of the device structure, the two structures are said to correspond. If the structures do not correspond, the problem is resolved by modification of either the device structure or the user's goal structure.

4.1.6 TAG

Task-Action Grammars (TAG) ([Green, 1989], [Payne & Green, 1986]) provide a formal model of a user's mental representation of a task structure ("task language"). A TAG description of an interface language consists of a *simple-task dictionary*, defining each low-level user task in terms of a set of semantic components, and a *feature grammar* to translate these semantic components into actions. [Payne & Green, 1986] provides examples of task-action grammars for a number of small experimental languages and a very small subset of the MacDraw drawing program.

Payne and Green refer to Reisner's BNF-based action grammars, and claim that task-action grammars make it possible to represent and measure some forms of consistency that BNF cannot capture. Consider as an example an editor with commands to delete a

line (**DL**), delete a word (**DW**), insert a line (**IL**), or insert a word (**IW**). A BNF represenation of this language fragment might resemble the following:

```
edit-command      ::= delete-command | insert-command
delete-command    ::= delete-word | delete-line
insert-command    ::= insert-word | insert-line
delete-word       ::= D + W
delete-line       ::= D + L
insert-word       ::= I + W
insert-line       ::= I + L
```

A TAG description would maintain delete-word, delete-line, insert-word, and insert-line as entries in the simple-task dictionary, parameterized by semantic concepts for operation and scope:

delete-word [operation = delete, scope = word]
delete-line [operation = delete, scope = line]
insert-word [operation = insert, scope = word]
insert-line [operation = insert, scope = line]

The feature grammar to translate these simple-tasks into actions would then consist of the following rules:

Task [operation, scope] → letter [operation] + letter [scope]
letter [operation = delete] → **D**
letter [operation = insert] → **I**
letter [scope = word] → **W**
letter [scope = line] → **L**

The single task schema that generates all four editing commands reveals their underlying structural consistency. In addition, this structured representation can reveal an aspect of command-structure completeness: if one of the commands were missing — if, for example, the delete command could not be applied to a scope of a single word — the process of developing a task schema would reveal that omission.

Payne and Green report useful results in determining consistency and complexity using TAG. Like GOMS, though, the utility of a TAG representation depends on the skill and

judgement of the analyst producing the description. In addition, task-action grammars, like most of the preceding approaches, do not address system activity or temporal relations among tasks.

4.2 Related constructional models

In Chapter 2, we discussed the importance of the behavioral approach in the development process, and we indicated that our work is focused on behavioral issues. However, crossing the gap between interface design and implementation necessitates some contact with the constructional realm. We have chosen two constructional models for consideration as possible targets for translation.

4.2.1 Jacob's specification diagrams

Jacob ([1985], [1986]) presents a dialogue specification technique that incorporates transition diagrams for definition of interaction object syntax. This technique is aimed at specification of direct-manipulation interfaces, which present the user at nearly every point with a wide variety of possible activities (for example, selecting any command from a palette, or selecting one of many objects). Jacob argues that such interfaces, though they appear modeless, are actually highly moded — since (for example) moving the cursor to a new object changes the effect that the mouse button will have, the cursor's location reflects a mode. Since transition among these modes is quick and reversible, and since the current mode is always obvious, many of the usual objections to interface modes do not apply.

To reflect this modal structure, an interface is represented as a collection of concurrent *interaction objects*, each supporting its own small dialogue. An interaction object is the smallest unit exhibiting syntax, that is, the smallest unit with which the user conducts a step-by-step dialogue. Each object's syntax is specified by a single-thread state diagram, with nodes representing states of the object's local dialogue, and arcs labelled with events representing user or system activities. When the user action causes an event not recognized by the current object's dialogue, that dialogue is *suspended*; a higher-level *executive* determines which object should handle the action, and resumes the appropriate object's dialogue.

This specification technique appears quite similar to the timetree formalism. Both have nodes representing states of a dialogue and arcs corresponding to user or system activity. It is easy to interpret Jacob's diagrams as specifications in the behavioral realm; in fact, Jacob cites the need to consider dialogue "from the user's point of view" as one motivation for his work. But while he does consider the user's view of direct-manipulation interfaces, several characteristics of his representation technique make it insufficient for behavioral specification.

One descriptive shortfall in Jacob's specification technique is associated with the lexical behavior of his dialogue objects. These objects receive events reflecting user actions, and generate events that correspond to actions visible to the user. The events are actually machine-oriented abstractions of actual user activities or visible system responses. To get from user actions to input events, something must sense the user actions, perform lexical analysis to generate tokens, and dispatch the events representing these tokens to the state machine representing the appropriate interaction object. To get from output events to actual perceptible output, something must accept the event and generate the appropriate perceptible activity.

Jacob abstracts all these duties out of his dialogue representation to reduce the complexity of the specifications. While this is certainly useful at some stages of design, it is important to *allow* discussion of these issues in behavioral terms. While Jacob discusses subdiagrams for lexical specification, he acknowledges that many lexical tasks "...could best be described in some more intuitive fashion than a large state diagram with a very regular structure" [Jacob, 1985].

In a similar vein, representing an interface as a collection of parallel dialogues under the control of some executive does not allow any consideration of task structure. Many kinds of higher-level interaction among low-level dialogues are easy to represent in terms of task composition; under Jacob's representation, such interactions can only be represented in terms of "synthetic events" and semantic variables, a very constructional approach.

Jacob's specification technique is oriented toward interfaces with a uniform, flat high-level structure, composed of objects with differing syntactic structures. In such systems, a task-based behavioral description might closely fit Jacob's constructional description.

The behavioral specification would consist of many small task descriptions, composed at a higher level by an interleaving operator; the constructional specification would consist of many small state diagrams, under the control of the executive.

Systems with a more complex behavioral structure require a specification technique that can capture that structure. Timetrees can represent such structure, and so are powerful enough to define behavioral representations that can capture it.

4.2.2 UIDE

The User Interface Development Environment (UIDE) [Foley, Kim, Kovacevic, & Murray, 1989] aims to capture knowledge about an interface design at a higher level of abstraction. It is centered around "a knowledge-based representation of the interface's conceptual design based on an object-oriented data model and an operation-oriented control model." These models, though, are constructional.

The data model is based on machine-oriented objects, their class hierarchy, properties, and actions. The control model is based on actions (which resemble methods), information required for the actions (which resembles parameters), and pre- and postconditions to control sequencing of actions. Preconditions must be true for a command to be invoked (like viability conditions for tasks in the UAN), and can enable and disable commands. Postconditions change values of the variables specified in the preconditions, and thus can control the progress of a dialogue.

This scheme provides some features not found in other environments. Since knowledge about information required by commands is included in the interface specification, it is possible to prompt for information not supplied with a command. When the user tries to specify a disabled command, the system can determine what preconditions for that command are not met and automatically provide help. Information needed for commands can be factored out via defaults or global values, and interface designs can be checked for some forms of completeness and consistency.

But the knowledge base does not incorporate any information about higher-level task structure. If the higher-level structure of the interface is complex, this structure must be

coded in terms of pre- and postconditions or pushed into the non-interface part of the system, written in a conventional programming language. In addition, presentation style, syntactic styles, and basic interaction techniques are left to a user interface management system, SUIMS (Simple User Interface Management System), which interprets UIDE interface representations at execution time.

UIDE captures some important information about an interface design, and SUIMS provides some behavioral support. But the specification of this interface information takes place in a constructional environment, and so UIDE cannot provide the behavioral support we claim is necessary for the development process.

On the other hand, timetrees can maintain the information that UIDE uses for preconditions and postconditions as state components. In fact, we refer to such components in task viability conditions. This suggests a bridge between a timetree representation of an interface and a UIDE implementation of the behavior it specifies. We will discuss this relationship further in Chapter 9.

4.3 A Device Model

A user wishing to provide input to a system must do so through manipulation of some input device. Sometimes the "manipulation" is subtle and indirect, as vibrations induced in a microphone. Sometimes the "input device" is usually thought of in other terms, as a power plug. But the need for some transducer between the user and the machine is a technological reality that is unlikely to change in the near future.

Many current representation techniques and models assume some ad hoc set of input devices — keyboards, mice, dials, switches, and so forth. But technological progress leads to an increasing variety of input devices suitable for use in interactive systems. Any general model of interactive behavior should be able to deal with these new devices as they appear.

[Card, Mackinlay, & Robertson, 1990] presents a formal model of input devices that appears to be extensible to most classes of new devices. Their model consists of primitive input device descriptions and composition operators over those primitives. An input

device is represented as a six-tuple **<M, In, S, R, Out, W>**. **M** is a *manipulation operator* describing the physical property sensed by the input device; this property is some combination of linear or rotary, absolute or relative, pressure or force. **In** is the *input domain*, the range of values that the manipulation can yield. **S** is the *current state* of the device, some value from the input domain. **R** is a *resolution function* that maps from the input domain to the *output domain*, **Out**. Finally, **W** is a catchall set of device properties which do not fit into the existing categories. Since Card et al. do not provide any examples of values for **W**, it is difficult to say more about its significance.

Each primitive input device exhibits a single degree of freedom, mapping from one sensed input value to one "output" value sent to the system. These primitive elements can be composed by *merge*, *layout*, and *connect* composition operators. Merge composition of two devices creates a new device whose input domain is the cross product of the input domains of the two component devices. Layout composition describes a physical association of the component devices. Connect composition maps the output domain of one device onto the input domain of another device; this composition operator allows description of virtual devices.

These primitive descriptions and composition operators describe a design space for input devices. Card et al. provide several graphical representations of this space to help visualize the taxonomy of input devices. They claim that they have been able to fit every device but voice input into their taxonomy. They also go on to analyze usability properties over various points of the design space, and illustrate measures of expressiveness (how precisely and uniquely an input activity conveys a meaning) and effectiveness (how easily and conveniently an input activity conveys its meaning).

It is possible to describe a straightforward mapping between the components of this input device model and the components of the timetree model. We present this mapping in Chapter 7.

4.4 Branching Time Temporal Logic

One of the tools used to formalize system behavior over time is *temporal logic*. There are many kinds of temporal logic, just as there are many forms of conventional logic [Hughes

& Cresswell, 1968]. Temporal logics provide modal operators to describe situations that vary over time. One way to visualize this is to imagine a sequence of states unfolding over time, with various propositions either true or false in each state. Conventional logics allow statements about one state. Temporal logics allow this, but they also can express statements such as "X is *always* true," "X will *eventually* be true," or "X will be true *before* Y is true."

One way to distinguish among temporal logics is by their model of time. Some temporal logics assume that each moment has only one possible future, with moments forming a linear chain from past to future; these are *linear time temporal logics*. In others, each moment may have many possible futures, with moments forming branching trees of possible futures; these are *branching time temporal logics*. The parallel with timetrees, in which choices among actions lead to branching structures of possible action sequences, is obvious.

There has been a long-running discussion over the relative expressive power of these two classes of temporal logic ([Lamport, 1980], [Emerson & Halpern, 1986], [Emerson & Srinivasan, 1988]). The discussion centers around the suitability of these logics for verification of concurrent programs and control systems, which requires expression of qualities such as fairness. One claim is that branching time is better suited for dealing with nondeterministic behavior. While some argue over whether nondeterministic behavior has any place in systems that are to be proven correct, we acknowledge that nondeterminism is certainly a necessary consideration in systems that involve a human being, and so we believe that a branching model of time is well-suited for reasoning about interactive systems.

Emerson has developed a temporal logic, CTL* ([Emerson & Halpern, 1986], [Emerson & Srinivasan, 1988]), based on a branching model of time. It provides formulae true at particular moments or true over paths, that is, over a series of moments. It incorporates path modalities to specify "all futures" or "at least one future"; these modalities are analogous to the universal and existential quantifiers of first-order logic, or the "necessity" and "possibility" modalities of modal logic. It also incorporates time modalities to express the concepts of "always" (at each moment along a path),

"sometime" (at some moment along a path), "nexttime" (at the next moment along a path), and "until" (at all future moments along a path until a condition becomes true).

Clarke, Emerson, and Sistla [1986] discuss application of CTL, the predecessor of CTL*, to concurrent system verification. In this application, tree-based time models are generated from state diagrams, with various system characteristics mapped onto states. Again, this strongly resembles the mapping of state information onto abstract states in timetrees.

In most of these applications of branching time temporal logic, though, only states are examined in detail. There is no facility for labelling arcs, since they only indicate the advance of time to the next moment. In addition, many of the abstractions we need for behavioral modelling do not seem applicable in these domains. However, the general branching structure these logics represent has had a strong influence on the structure of the timetree model.

4.5 Models of Concurrency

While we do not intend timetrees as a powerful and general model of concurrent systems, models of concurrency have some features in common with timetrees. We will illustrate this similarity for two models of concurrency: Milner's Calculus of Communicating Systems (CCS) [Milner, 1980] and Hoare's Communicating Sequential Processes (CSP) [Hoare, 1985].

Both CCS and CSP are based on algebraic specifications of systems. Systems are built up from *processes* that engage in *events.*

4.5.1 CCS

CCS describes an algebraic representation whose terms stand for behaviors of systems and whose operators compose such behaviors. Milner's exposition of CCS begins with finite state acceptors. Instead of referring to "events" or "tokens," he uses the term *experiment*. An α-experiment on an acceptor fails if there is no α-transition out of the acceptor's current state; if there is, the experiment succeeds, and the acceptor takes the transition to its next state. By considering only the possible unfoldings of experiments

(the *behavior* of the acceptor) and ignoring state information, it is possible to generate a tree of behaviors (a *synchronization tree*) resembling a timetree.

Milner then goes on to describe *mutual experimentation*, in which two machines can interact by engaging in complementary experiments (written as, for example, α and $\bar{\alpha}$); such internal experiments are not observable from outside the composition, and so provide a form of information hiding. He presents operations for construction and composition of systems, and while he defines these operations in algebraic terms, he illustrates them with synchronization tree diagrams.

4.5.2 CSP

CSP is also based on algebraic descriptions of systems. In this formalism, an object engages in atomic activities (*events*), and the behavior of such an object is represented by a *process*. Processes are composed recursively from such operators as prefixing, choice, concurrency, and interleaving. Concurrent processes interact by engaging together in events that they both perceive.

Like Milner, Hoare defines his operators in algebraic terms. His notation is based on LISP-style lists, generally represented by a **car** (the first element of the list) and a **cdr** (the rest of the list); in fact, he presents implementations of his language features in a small, purely functional dialect of LISP.

But also like Milner, Hoare offers tree-based visualizations of the behavior of his processes. In fact, like synchronization trees, his diagrams label arcs with behaviors in which a system engages, and label nodes with nothing at all.

Both these formalisms are oriented toward describing and reasoning about interacting collections of concurrent systems. Such systems present unique problems involving synchronization, deadlock, and fairness. Other approaches to specifying concurrent behavior exist, notably Petri nets (see, for example, [Reisig, 1985]).

Both Hoare and Milner view the interaction between a user and a machine as yet another interaction of concurrent systems. This is certainly a valid view; user mental processes and internal machine processes are generally independent of each other, with interaction taking place only at input and output devices from the machine's perspective, or with perception and action from the user's perspective. But user modeling and system modeling are very different problems, and it seems unlikely that a model suitable for one would be suitable for the other.

The UAN avoids this problem by attempting to model neither internal user structure and activity nor internal system structure and activity. Instead, it focuses on the *observable* behaviors of both entities, as well as observable system state (observable user state has not traditionally been a significant consideration). Since it is sometimes possible and convenient to refer to or alter internal system state, the UAN provides facilities for representing it, but only in an ad hoc fashion.

As a result of this approach, the UAN generally has not had to deal with the thornier issues of concurrency. It deals with relatively well-behaved sequences of user actions and system responses; where internal system activities are specified, their exact temporal relationship to other activities usually is not significant, beyond the usual restriction that they must follow their causes and precede their effects.

The versions of "concurrency" and "interleaving" that are important to the UAN concern simultaneously active *tasks*, but these tasks still represent interaction between a single user and system, and thus typically can be reduced to a single stream of actions. Situations in which multiple user activities or system activities can take place simultaneously generally are of limited scope, and so the full power and complexity of these models of concurrency is unneeded. The timetree model might be extended to deal with multiple users and multiple systems, but we do not attempt to do so in this research.

5 The Timetree Model

We have previously described the nature of timetrees in general terms. In this chapter, we present a formal definition of timetree components and operations.

We begin with a formal representation for timetrees. We next describe activities and their classification; timestates and the state information that can be mapped onto them; and finally, composition and abstraction operations on timetrees.

We adopt several typographical conventions to distinguish among elements of our notation. We represent a tuple by a list of its components, separated by commas and enclosed by angle brackets. We capitalize names of sets, and use lower case for names of single elements. Finally, we use dot (.) to specify a component of a particular tuple. So, for example, a simple tree can be represented as a tuple

$<N, A, r>$

where N is the set of nodes in the tree, A is the set of arcs, and r is the (single) root node. If T and S are both trees, we can refer to the root of T as $T.r$, or the arcs of S as $S.A$.

5.1 Formal representation

A timetree consists of a tuple

$<TS, SCV, SCM, AV, A, ts_{init}, TS_{term}>$

where

TS	is a set of *timestates*;
SCV	is a *state component vocabulary*, a set of *state component range assignments*;
SCM	is a *state component map*, a set of *state component value assignments*;
AV	is an *activity vocabulary*, a set of *activities*;
A	is a set of *activity arcs*;
ts_{init}	$\in TS$ is an *initial timestate*; and
TS_{term}	$\subseteq TS$ is a set of *terminal timestates*.

TS, ts_{init}, TS_{term}, and A represent the nodes and arcs of the timetree, and thus define its topology. SCV and SCM represent the state components associated with timestates, and AV contains the activities that compose the task described by the timetree.

Each state component range assignment in SCV consists of a pair

$<sc_{name}, SC_{range}>$

where

sc_{name} is a *state component identifier* unique within a timetree and

SC_{range} is a *state component range* from which that state component takes its values.

Intuitively, the state component vocabulary describes a set of state components and the range of values each state component can take on.

Each element of SCM consists of a triple

$<ts, sc_{name}, sc_{value}>$

where

ts is a timestate from TS,

sc_{name} is a state component identifier, and

sc_{value} is a value within that state component's range.

Thus, each element of the state component map associates a particular value with a particular state component at a particular timestate.

Each element of A consists of a triple

$<ts_{start}, ts_{end}, A_{label}>$

where

$ts_{start} \in TS$ is a *start timestate*,

$ts_{end} \in TS$ is an *end timestate*, and

$A_{label} \subseteq AV$ is a set of *labeling activities*.

Each activity arc represents a transition between two timestates, and is labeled with an unordered set of activities from the timetree's activity vocabulary. The start timestate ts_{start} represents the configuration of the modeled parties before any of the labeling activities have begun, and the end timestate ts_{end} represents the configuration after all the labeling activities have completed.

We impose several restrictions on timetrees. Since they are trees, they must have exactly one initial timestate (the root of the tree) that is the end timestate of no activity arc, and every other timestate must be the end timestate of exactly one activity arc:

$$\nexists\, a \in A: a.ts_{end} = ts_{init} \qquad (\nexists = \neg\exists, \text{ i.e. "there exists no"})$$

$$\forall a \in A: (\nexists\, b \in A: a.ts_{end} = b.ts_{end})$$

$$\forall ts \in TS, ts \neq ts_{init}: (\exists\, a \in A: a.ts_{end} = ts)$$

This precludes cycles and ensures that there is exactly one *path*, or sequence of activity arcs, from the initial timestate to each of the other timestates. We refer to this sequence of activity arcs as the *history* of the timestate. We define the history of a timestate *ts* as the ordered sequence

$$a_1, a_2, \ldots, a_d$$

where

$$\begin{aligned} a_i &\in A, \\ a_1.ts_{start} &= ts_{init}, \\ a_i.ts_{end} &= a_{i+1}.ts_{start} \text{ for } i = 1, 2, \ldots, d\text{-}1, \text{ and} \\ a_d.ts_{end} &= ts \end{aligned}$$

Since each activity arc represents a group of activities that take place between the configuration at the start timestate and the configuration at the end timestate, the history of a timestate represents the sequence of activities that led to the configuration that timestate represents. We define activities and activity arcs in more detail in Section 5.2.1, and timestates and configurations in Section 5.2.2.

The number of activity arcs composing the history of a timestate yields the *depth* of that timestate, represented above by *d*. The initial timestate has depth zero, any timestate

reached from the initial timestate by one activity arc has depth one, and so forth. Every finite tree has a *tree-depth*, defined as the maximum depth of any timestate:

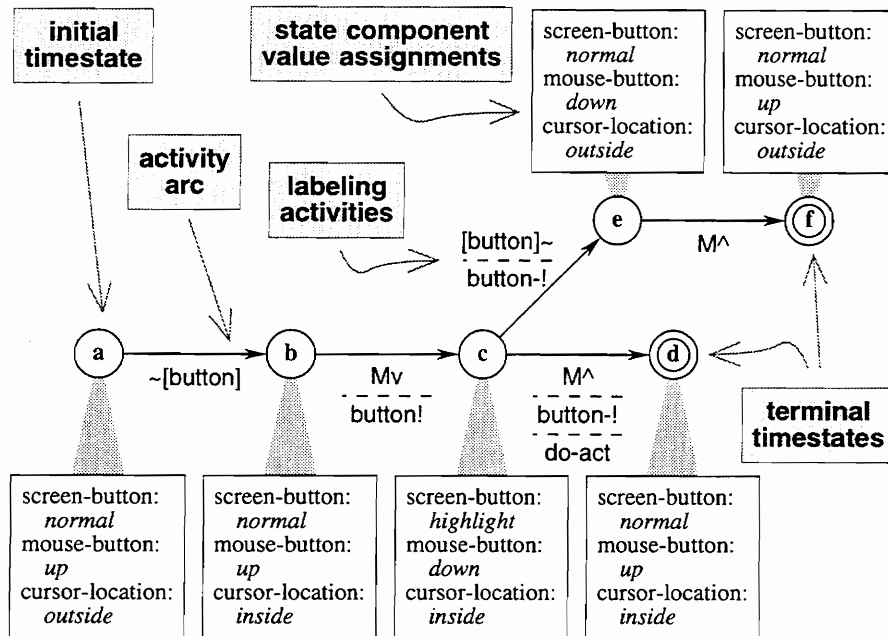
$$\begin{aligned} \mathit{tree\text{-}depth}(tt) = td\text{: } & (\exists\ ts \in tt.TS\text{: } depth(ts) = td) \wedge \\ & (\nexists\ ts \in tt.TS\text{: } depth(ts) > td) \end{aligned}$$

Since some of the behaviors we model involve indefinite repetition (for example, an action the user can repeat "zero or more times") we allow infinite paths, and thus infinite timetrees. This means that *TS*, *SCM*, *A*, and TS_{term} can be infinite. However, we require that each timestate can be the start timestate of only a finite number of activity arcs. In other words, timetrees are *finitely branching*.

We also require that the state component and activity vocabularies, *SCV* and *AV*, be finite, and that the set of activities labeling any single activity arc be finite. As we compose timetrees, *SCV* and *AV* may grow, but they cannot become infinite. Most of our composition operators combine two timetrees to yield a new one, and the state component and activity vocabularies of this tree are formed from the union of the vocabularies of the component trees; the one composition operator that does not follow this pattern, *closure*, does not change *SCV* or *AV* at all. The concurrency composition operator and some abstraction operators can add labeling activities to an activity arc, but again, the number of labeling activities will remain finite.

The elements of the state component map, *SCM*, are intended to assign at most one state component value for each state component name at each timestate; that is, for any timestate *ts* and any state component name sc_{name}, there must be at most one element $\langle ts, sc_{name}, sc_{value} \rangle$ in *SCM*. Since *TS* can be infinite, this implies that *SCM* can be infinite, as stated above; but since *SCV* is finite, there must be a finite number of state component value assignments associated with any particular timestate. In other words, only a finite amount of state information is associated with each timestate, and each state component must take on at most one value at any single timestate.

Figure 5.1 illustrates a simple timetree both as a labeled diagram and as a set of tuples.



TT: { **a**, **b**, **c**, **d**, **e**, **f** }

SCV: { < screen-button, { *highlight*, *normal* } >,
< mouse-button, { *up*, *down* } >,
< cursor-location, { *inside*, *outside* } > }

SCM: { < **a**, screen-button, *normal* >,
< **a**, mouse-button, *up* >,
< **a**, cursor-location, *outside* >,
< **b**, screen-button, *normal* >,
< **b**, mouse-button, *up* >,
< **b**, cursor-location, *inside* >,
< **c**, screen-button, *highlight* >,
< **c**, mouse-button, *down* >,
< **c**, cursor-location, *inside* >,
< **d**, screen-button, *normal* >,
< **d**, mouse-button, *up* >,
< **d**, cursor-location, *inside* >,
< **e**, screen-button, *normal* >,
< **e**, mouse-button, *down* >,
< **e**, cursor-location, *outside* >,
< **f**, screen-button, *normal* >,
< **f**, mouse-button, *up* >,
< **f**, cursor-location, *outside* > }

AV: { ~[button], [button]~, Mv, M^, button!, button-!, do-act }

A: { < **a**, **b**, { ~[button] } >,
< **b**, **c**, { Mv, button! } >,
< **c**, **d**, { M^, button-!, do-act } >,
< **c**, **e**, { [button]~, button-! } >,
< **e**, **f**, { M^ } > }

ts_{init}: **a**

ts_{term}: { **d**, **f** }

Figure 5.1. Representations of a sample timetree.

5.2 Timetree components

5.2.1 Activities

An *activity* is an event or process involving the participation of one or more of the interacting parties of interest to us — the user, the system, and possibly the environment. We can classify activities according to the parties involved. *User activities* are performed by the user alone. Similarly, *system activities* are performed by the system, and *environment activities* are performed by entities other than the user or system. Some activities involve participation by more than one party; we refer to these as *joint activities.*

Abstraction techniques (which will be presented shortly) allow us to view sequences or other combinations of activities as a single activity, which we will call a *compound activity.* A compound activity can be a user, system, environment, or joint activity, depending on the types of activities that compose it. For example, a compound activity that consists of a sequence of user activities is itself a user activity; a compound activity that consists of a user activity and a system response is a joint activity.

Timetrees model these activities with the set A of *activity arcs*, the arcs of the tree. Each member of A is uniquely identified by its source and destination, and is labeled with the activity or activities it models. The *activity vocabulary* AV is the set of activities mentioned in the tree.

The formal representation of the timetree model does not differentiate among system, user, environmental, and joint activities. Where a formal classification is necessary, we define a function $Aclass(act)$ that maps each activity act in the activity vocabulary to a nonempty subset of $\{user, system, environment\}$. Thus, if act is a user activity (say, pressing a key), $Aclass(act) = \{user\}$; if act is a joint activity between the user and the system (say, dragging an icon image across the screen), $Aclass(act) = \{user, system\}$. If act is a compound activity composed of activities act_1, act_2, ..., act_n, then

$$Aclass(act) := \bigcup_{i=1}^{n} Aclass(act_i)$$

If an activity arc is labeled with more than one activity, the temporal ordering of the activities is undetermined. In Figure 3.10, we presented a picture of a timetree in which some arcs were labeled with *sequences* of activities. This was only a notational convenience; an accurate representation of such a sequence would assign each activity to its own arc, resulting in a correspondingly more sparse timetree.

In addition to the actual activities described above, it will sometimes be convenient to augment timetrees with *virtual activities.* These activities will not reflect any action on the part of any party, but will merely serve as a kind of scaffolding during certain composition and abstraction operations. They will appear, and be addressed in more detail, in Chapter 6.

5.2.2 Timestates

A *timestate* represents the abstract configuration of all interacting parties of interest at a particular point in time. Activities take the collection of interacting parties from one timestate to another. Every activity has a single *start timestate* and a single *end timestate.*

Each timetree has a unique *initial timestate* describing the configuration of all interacting parties before any activity takes place. This initial timestate is the root of the timetree.

Each timetree with terminating sequences of activities has a set of *terminal timestates.* Each such timestate describes a possible completion of the activity or activities modeled by the timetree — each possible execution of the activity described by the timetree is described by a path from the initial to a terminal timestate. (Initial and terminal timestates of timetrees are thus analogous to start and end timestates for an activity.) Terminal timestates are not always leaf nodes of a timetree. For example, the last activity in a task may be optional; since the user can complete the task with or without performing this activity, the timestates preceding it and following it would both be terminal. Leaf nodes, on the other hand, are always terminal timestates — except in *d-restricted* timetrees, a construct which we introduce in Section 5.3.1 as a tool for defining operations over infinite timetrees.

5.2.3 State components

A *state component* is a particular aspect of system, user, or environmental state that can take on one or more values. Each component is assigned a unique *state component identifier* and a range of possible values. These identifiers and ranges form the *state component vocabulary* of a timetree.

To describe the value of a state component over time, we assign the component a particular value from its range at each timestate. The set of such *state component value assignments*, each specifying a timestate, a state component identifier, and a value from the state component's range, forms the *state component map* for a timetree.

The timetree model does not specify a particular set of data types from which state component ranges and values may be drawn. The nature of state information will vary among entities (user, system, environment), among application domains, and among other models used in conjunction with timetrees. The only requirement imposed by the timetree model on state component ranges is that they provide a union operator and tests for membership and equality among members.

The formal representation of the timetree model also does not differentiate among system, user, and environmental state. Where a formal classification is necessary, we define a function $SCclass(sc_{name})$ that maps each state component identifier sc_{name} in the state component vocabulary to one element of {*user*, *system*, *environment*}, depending on the type of state the identifier represents.

We can define a function *SCvalues(ts, tt)* to yield the finite set of state component identifiers and values associated with a particular timestate *ts* in a timetree *tt*:

$$SCvalues(ts, tt) := \{ \langle sc_{name}, sc_{value} \rangle : \langle ts, sc_{name}, sc_{value} \rangle \in tt.SCM \}$$

We also define a boolean function *state-clash* to test whether two timestates (potentially in two different trees) have incompatible state component value assignments. If two timestates map different values to the same state component, they cannot be collapsed into a single state; this will be important in some of the operations that we will define later.

$$
\begin{aligned}
&state\text{-}clash(ts_1, tt_1, ts_2, tt_2) := \\
&\qquad \exists\, sc_{name}\colon (\; \langle ts_1, sc_{name}, sc_{v1}\rangle \in tt_1.SCM \wedge \\
&\qquad\qquad \langle ts_2, sc_{name}, sc_{v2}\rangle \in tt_2.SCM \wedge \\
&\qquad\qquad sc_{v1} \neq sc_{v2}\;)
\end{aligned}
$$

That is, *state-clash* returns *true* if the two timestates have one or more conflicting value mappings, *false* otherwise.

It will sometimes be convenient to introduce *virtual* state components, analogous to the virtual activities mentioned in Section 5.2.1. Like those virtual activities, virtual state components will be addressed in Chapter 6.

5.3 Compositions

The simplest possible timetree is described by the tuple

$$\langle \{ ts_{init} \}, \varnothing, \varnothing, \varnothing, \varnothing, ts_{init}, \{ ts_{init} \} \rangle$$

It consists of one timestate, no activities, and no state mappings, and has tree-depth zero. We will refer to this timetree as the *null timetree*, called *null.*

The simplest nontrivial timetrees consist of a single activity arc with its start and end timestates. To build timetrees describing more interesting patterns of behavior, it is necessary to combine multiple activities.

5.3.1 Choice

Choice composition combines two timetrees *ta* and *tb* to yield a new timetree (*ta* | *tb*) that describes the union of all possible paths through *ta* and all possible paths through *tb*. Thus, from the initial timestate of (*ta* | *tb*), it is possible to *choose* either one of the sequences from timetree *ta* or one of the sequences from timetree *tb*. For example, this

kind of composition is common in menu-based system designs that also support keyboard shortcuts: to save a document, the user can select the "Save" entry from a menu, or type the corresponding keyboard shortcut (say, "command-S"). Either sequence of actions serves to perform the "save document" task.

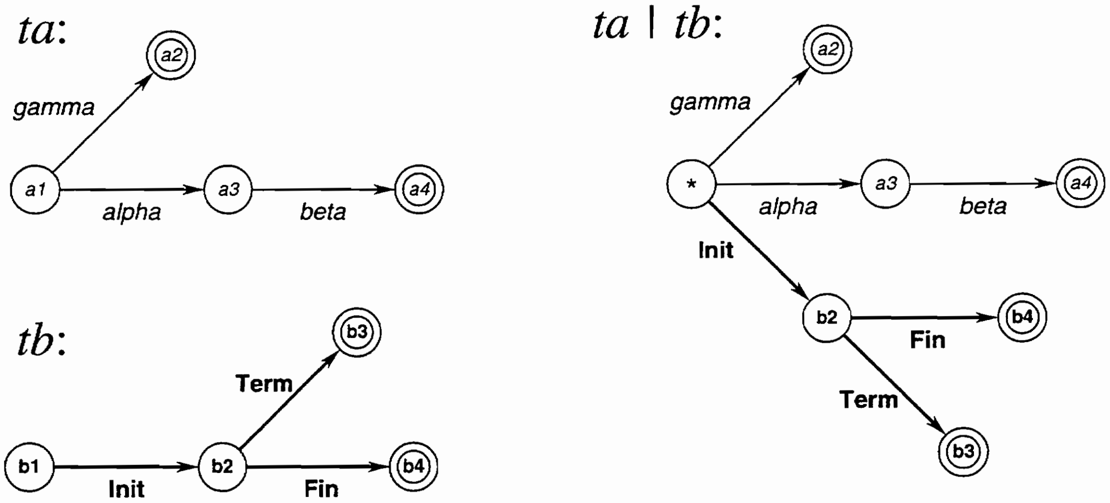


Figure 5.2. Choice composition of two timetrees.

Figure 5.2 illustrates the choice composition of two timetrees. Informally, we construct (*ta* | *tb*) by creating a new initial timestate, removing the initial timestates of *ta* and *tb*, and replacing every activity originating at the initial timestate of *ta* or *tb* with an activity originating at the new timestate. Formally, we begin by generating the new set of timestates, consisting of the union of *ta.TS* and *tb.TS*, omitting their original initial timestates, and adding a new initial timestate:

$$\begin{aligned}(ta \mid tb).TS := &\ \{ts_{init.new}\} \\ &\cup\ (ta.TS - \{ta.ts_{init}\}) \\ &\cup\ (tb.TS - \{tb.ts_{init}\})\end{aligned}$$

Next, we create the new state component vocabulary as the union of *ta.SCV* and *tb.SCV*. In the special case where *ta* and *tb* assign different ranges to a single state component identifier, we assign it the union of the two original ranges. (Thus, for example, a state

component ranging over a different set of interface objects in each timetree would range over the union of the sets in the new timetree.)

$$
\begin{aligned}
(ta \mid tb).SCV := \\
& \{ <sc_{name}, SC_{r1} \cup SC_{r2}>: \\
& \quad <sc_{name}, SC_{r1}> \in ta.SCV \wedge \\
& \quad <sc_{name}, SC_{r2}> \in tb.SCV \} \\
\cup\ & \{ <sc_{name}, SC_{r1}>: \\
& \quad <sc_{name}, SC_{r1}> \in ta.SCV \wedge \\
& \quad \nexists\ SC_{r2}: <sc_{name}, SC_{r2}> \in tb.SCV \} \\
\cup\ & \{ <sc_{name}, SC_{r2}>: \\
& \quad <sc_{name}, SC_{r2}> \in tb.SCV \wedge \\
& \quad \nexists\ SC_{r1}: <sc_{name}, SC_{r1}> \in ta.SCV \}
\end{aligned}
$$

We form the new state component map as the union of *ta.SCM* and *tb.SCM*. Map entries defining state values for the initial timestates of *ta* or *tb* are replaced by entries defining the same state values for the new initial timestate of $(ta \mid tb)$.

$$
\begin{aligned}
(ta \mid tb).SCM := \\
& \{ <ts_{init.new}, sc_{name}, sc_{value}>: \\
& \quad <ta.ts_{init}, sc_{name}, sc_{value}> \in ta.SCM \} \\
\cup\ & \{ <ts_{init.new}, sc_{name}, sc_{value}>: \\
& \quad <tb.ts_{init}, sc_{name}, sc_{value}> \in tb.SCM \} \\
\cup\ & \{ <ts, sc_{name}, sc_{value}> \in ta.SCM: \\
& \quad ts \neq ta.ts_{init} \} \\
\cup\ & \{ <ts, sc_{name}, sc_{value}> \in tb.SCM: \\
& \quad ts \neq tb.ts_{init} \}
\end{aligned}
$$

Note that this union does not consider the possibility that $ta.ts_{init}$ and $tb.ts_{init}$ have incompatible state component value assignments. It is possible to check for this situation by evaluating *state-clash*$(ta.ts_{init}, ta, tb.ts_{init}, tb)$. There are several ways to deal with a state clash in this situation. Depending on our goals, we might ignore the conflict, have the composition fail and yield an undefined result, pick one of *ta* or *tb*, or insert new activities that change the offending state component values. We will consider this situation in detail in Chapter 6.

The activity vocabulary of $(ta \mid tb)$ is simply the union of the activity vocabularies from *ta* and *tb*:

$$(ta \mid tb).AV := ta.AV \cup tb.AV$$

The set of activities for $(ta \mid tb)$ is again the union of activities from *ta* and *tb*, with activities originating at the initial timestates of *ta* or *tb* replaced by activities originating from the new initial timestate of $(ta \mid tb)$.

$$\begin{aligned}(ta \mid tb).A := \quad & ta.A \cup tb.A \\ \cup \quad & \{ \langle ts_{init.new}, ts_{end}, A_{label} \rangle : \\ & \quad \langle ta.ts_{init}, ts_{end}, A_{label} \rangle \in ta.A \} \\ \cup \quad & \{ \langle ts_{init.new}, ts_{end}, A_{label} \rangle : \\ & \quad \langle tb.ts_{init}, ts_{end}, A_{label} \rangle \in tb.A \} \\ - \quad & \{ \langle ts_{start}, ts_{end}, A_{label} \rangle : \\ & \quad ts_{start} \in \{ta.ts_{init}, tb.ts_{init}\} \}\end{aligned}$$

The initial timestate for $(ta \mid tb)$ is the one we have just created:

$$(ta \mid tb).ts_{init} := ts_{init.new}$$

Finally, the set of terminal timestates for $(ta \mid tb)$ is the union of $ta.TS_{term}$ and $tb.TS_{term}$. If the initial timestate of *ta* or *tb* is also a terminal timestate, we remove it from the set of terminal timestates, replacing it with the new initial timestate.

$$\begin{aligned}(ta \mid tb).TS_{term} := & \\ & (ta.TS_{term} - \{ta.ts_{init}\}) \\ \cup \quad & (tb.TS_{term} - \{tb.ts_{init}\}) \\ \cup \quad & \begin{cases} \{ts_{init.new}\} & \text{if } ta.ts_{init} \in ta.TS_{term} \vee tb.ts_{init} \in tb.TS_{term} \\ \varnothing & \text{otherwise} \end{cases}\end{aligned}$$

This definition is adequate for finite timetrees. For timetrees with infinite paths, though, it involves operations of union and selection over infinite sets, which are vaguely disquieting. We would prefer a definition in terms of finite, terminating entities and operations.

We can produce such a definition by introducing the concept of *d-restricted* timetrees and timetree components. Intuitively, we *restrict a tree to depth d* by removing from it all timestates with depth greater than *d*, and all activity arcs, state component value assignments, and elements of TS_{term} that refer to such timestates. Since the state component vocabulary and activity vocabulary are finite by definition and do not refer to timestates, we need not modify them; while we could delete vocabulary elements that do

not label any timestates or activity arcs of depth $\leq d$, we choose not to do so. Formally, we define a *d-restricted* timetree tt ($tt\rceil^d$) as follows:

$$tt\rceil^d.TS := \{ ts \in tt.TS: depth(ts) \leq d \}$$

$$tt\rceil^d.SCV := tt.SCV$$

$$tt\rceil^d.SCM := \{ <ts, sc_{name}, sc_{value}> \in tt.SCM: depth(ts) \leq d \}$$

$$tt\rceil^d.AV := tt.AV$$

$$tt\rceil^d.A := \{ <ts_{start}, ts_{end}, A_{label}> \in tt.A: depth(ts_{end}) \leq d \}$$

$$tt\rceil^d.ts_{init} := tt.ts_{init}$$

$$tt\rceil^d.TS_{term} := \{ ts \in tt.TS_{term}: depth(ts) \leq d \}$$

For a timetree with infinite paths, the limit of $tt\rceil^d$ as d approaches infinity is tt:

$$\lim_{d \to \infty} tt\rceil^d = tt$$

For a finite timetree of tree-depth n, $tt\rceil^d = tt$ for all $d \geq n$, and again $\lim_{d \to \infty} tt\rceil^d = tt$.

This notion of "limit" differs from the more conventional limit of a convergent series. For a timetree with infinite paths, as d goes to infinity, $tt\rceil^d$ grows indefinitely, rather than converging to one finite structure. However, as d increases, $tt\rceil^d$ encompasses more and more of the full structure of tt. This is analogous to the definition of an ω-word as the *limit* of an infinite sequence of prefixes in infinitary language theory [Hoogeboom & Rozenberg, 1986].

Note that d-restriction of an infinite timetree, or d-restriction of a finite timetree of depth greater than d, can yield a timetree with leaf nodes that are not terminal timestates. In using timetrees to describe behavior, we will usually assume that from any timestate there is some path that eventually leads to a terminal timestate, and thus to the completion of the behavior. A timestate that is not terminal and has no activities leaving it implies that a system is "stuck," comparable to the condition of deadlock in the study of concurrent systems. In our applications of d-restricted timetrees, though, this is not the case, because

a sufficient increase in d should reveal a terminal timestate at some future point. We will return to the notion of nonterminal leaf nodes in Chapters 6, 8, and 9. For now, though, it is not significant in our definitions.

Now we can recast the definition of $(ta \mid tb)$ in terms of d-restricted timetrees:

$$
\begin{aligned}
(ta \mid tb)\rceil^{d}.TS := \quad & \{ts_{init.new}\} \\
\cup \quad & (ta\rceil^{d}.TS - \{ta\rceil^{d}.ts_{init}\}) \\
\cup \quad & (tb\rceil^{d}.TS - \{tb\rceil^{d}.ts_{init}\})
\end{aligned}
$$

$$
\begin{aligned}
(ta \mid tb)\rceil^{d}.SCV := \quad & \\
& \{ <sc_{name}, SC_{r1} \cup SC_{r2}>: \\
& \qquad <sc_{name}, SC_{r1}> \in ta\rceil^{d}.SCV \wedge \\
& \qquad <sc_{name}, SC_{r2}> \in tb\rceil^{d}.SCV \} \\
\cup \quad & \{ <sc_{name}, SC_{r1}>: \\
& \qquad <sc_{name}, SC_{r1}> \in ta\rceil^{d}.SCV \wedge \\
& \qquad \nexists\, SC_{r2}: <sc_{name}, SC_{r2}> \in tb\rceil^{d}.SCV \} \\
\cup \quad & \{ <sc_{name}, SC_{r2}>: \\
& \qquad <sc_{name}, SC_{r2}> \in tb\rceil^{d}.SCV \wedge \\
& \qquad \nexists\, SC_{r1}: <sc_{name}, SC_{r1}> \in ta\rceil^{d}.SCV \}
\end{aligned}
$$

$$
\begin{aligned}
(ta \mid tb)\rceil^{d}.SCM := \quad & \\
& \{ <ts_{init.new}, sc_{name}, sc_{value}>: \\
& \qquad <ta.ts_{init}, sc_{name}, sc_{value}> \in ta\rceil^{d}.SCM \} \\
\cup \quad & \{ <ts_{init.new}, sc_{name}, sc_{value}>: \\
& \qquad <tb.ts_{init}, sc_{name}, sc_{value}> \in tb\rceil^{d}.SCM \} \\
\cup \quad & \{ <ts, sc_{name}, sc_{value}> \in ta\rceil^{d}.SCM: \\
& \qquad ts \neq ta\rceil^{d}.ts_{init} \} \\
\cup \quad & \{ <ts, sc_{name}, sc_{value}> \in tb\rceil^{d}.SCM: \\
& \qquad ts \neq tb\rceil^{d}.ts_{init} \}
\end{aligned}
$$

$$
(ta \mid tb)\rceil^{d}.AV := \quad ta\rceil^{d}.AV \cup tb\rceil^{d}.AV
$$

$$
\begin{aligned}
(ta \mid tb)\rceil^d.A := \quad & ta\rceil^d.A \cup tb\rceil^d.A \\
\cup \; & \{ <ts_{init.new}, ts_{end}, A_{label}> : \\
& \quad <ta\rceil^d.ts_{init}, ts_{end}, A_{label}> \in ta\rceil^d.A \} \\
\cup \; & \{ <ts_{init.new}, ts_{end}, A_{label}> : \\
& \quad <tb\rceil^d.ts_{init}, ts_{end}, A_{label}> \in tb\rceil^d.A \} \\
- \; & \{ <ts_{start}, ts_{end}, A_{label}> : \\
& \quad ts_{start} \in \{ta\rceil^d.ts_{init}, tb\rceil^d.ts_{init}\} \}
\end{aligned}
$$

$$(ta \mid tb)\rceil^d.ts_{init} := ts_{init.new}$$

$$
\begin{aligned}
(ta \mid tb)\rceil^d.TS_{term} := & \\
& (ta\rceil^d.TS_{term} - \{ta\rceil^d.ts_{init}\}) \\
\cup \; & (tb\rceil^d.TS_{term} - \{tb\rceil^d.ts_{init}\}) \\
\cup \; & \begin{cases} \{ts_{init.new}\} & \text{if } ta\rceil^d.ts_{init} \in ta\rceil^d.TS_{term} \vee \\ & \quad ttb\rceil^d.ts_{init} \in tb\rceil^d.TS_{term} \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}
$$

If either or both of *ta* and *tb* are infinite timetrees, $(ta \mid tb)$ is defined as $\lim_{d \to \infty} (ta \mid tb)\rceil^d$. For finite *ta* and *tb* of tree-depth *m* and *n* respectively, $(ta \mid tb)\rceil^d = (ta \mid tb)$ for $d \geq \max(m,n)$, so again $\lim_{d \to \infty} (ta \mid tb)\rceil^d = (ta \mid tb)$.

5.3.2 Sequence

Most interface designs contain tasks that require the user to perform sequences of actions, and system responses that must follow particular user actions. Behavioral representations provide sequential composition operators to represent these ordered relationships. Sequence composition combines two timetrees *ta* and *tb* to yield a new timetree $(ta \, ; tb)$ that describes all possible paths consisting of a possible path through *ta* followed by a possible path through *tb*. Informally, we construct $(ta \, ; tb)$ by making a copy of timetree *tb* for each terminal timestate of timetree *ta*, then “grafting” each new copy onto *ta* by replacing a terminal timestate in *ta* with the initial timestate of a copy of *tb*. Figure 5.3 illustrates the result of this process.

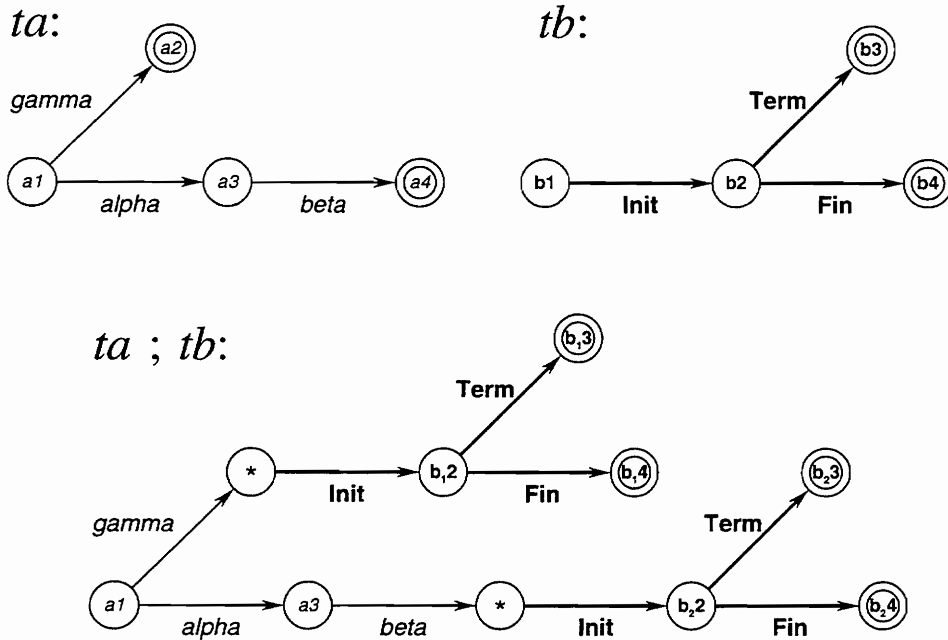


Figure 5.3. Sequence composition of two timetrees.

To perform the formal construction, we need a cloning operation to generate new instances of timestates and activity arcs for the otherwise identical timetree copies. This requires a function *tscopy* that maps an existing timestate from the old timetree to a new timestate. We use this function to create a new set of unique timestates, state component mappings, and activity arcs, and to map the initial timestate, the set of terminal timestates, and the start and end timestate of each activity arc to members of the new timestate set. Thus, we can define a new timetree *tnew* in terms of an existing timetree *tt* and *tscopy*.

First, we create a new set of timestates corresponding to the timestates of the existing timetree:

$$tnew.TS := \{ ts_{new} : ts_{new} = tscopy(ts_{old}) \wedge ts_{old} \in tt.TS \}$$

The state component vocabulary is unchanged:

$$tnew.SCV := tt.SCV$$

Next, we create a set of state component value assignments that map the new timestates to the same state component values as the corresponding timestates in the existing timetree:

$$tnew.SCM := \{ <tscopy(ts_{old}), sc_{name}, sc_{value}>: \\ <ts_{old}, sc_{name}, sc_{value}> \in tt.SCM \}$$

The activity vocabulary, like the state component vocabulary, is unchanged:

$$tnew.AV := tt.AV$$

We create a new set of activity arcs in the same way that we created the new state component value assignments, by making reference to new timestates corresponding to those in the existing timetree:

$$tnew.A := \{ <ts_{start.new}, ts_{end.new}, A_{label}> : \\ ts_{start.new} = tscopy(ts_{start.old}) \wedge \\ ts_{end.new} = tscopy(ts_{end.old}) \wedge \\ <ts_{start.old}, ts_{end.old}, A_{label}> \in tt.A \}$$

The initial timestate of the new timetree is the timestate that corresponds to the initial timestate of the old timetree:

$$tnew.ts_{init} := tscopy(tt.ts_{init})$$

Finally, we create the new timetree's TS_{term} from the new timestates corresponding to those in the old timetree's TS_{term}:

$$tnew.TS_{term} := \{ ts_{new} : ts_{new} = tscopy(ts_{old}) \wedge ts_{old} \in tt.TS_{term} \}$$

We abbreviate this construction as *tnew* := *tsclone*(*tt*). With the aid of this function, we construct a set of new timetrees *Bgrove*, one for each element of $ta.TS_{term}$. Since we will be attaching each new copy of *tb* onto *ta* at one of these terminal timestates, we refer to

the current terminal timestate in *ta* as ts_{peg}, and substitute it for the initial timestate of the new copy.

$Bgrove := \varnothing;$

for each $ts_{peg} \in ta.TS_{term}$:

$$b_{temp} := \quad tsclone(tb);$$

$$b_{new}.TS := \quad b_{temp}.TS \cup \{ts_{peg}\} - \{b_{temp}.ts_{init}\};$$

$$b_{new}.SCV := \quad b_{temp}.SCV;$$

$$\begin{aligned} b_{new}.SCM := \quad & b_{temp}.SCM \\ & \cup \{ \; <ts_{peg}, SC_{name}, SC_{value}> : \\ & \qquad <b_{temp}.ts_{init}, SC_{name}, SC_{value}> \in b_{temp}.SCM\} \\ & - \{ \; <b_{temp}.ts_{init}, SC_{name}, SC_{value}> \in b_{temp}.SCM\}; \end{aligned}$$

$$b_{new}.AV := \quad b_{temp}.AV;$$

$$\begin{aligned} b_{new}.A := \quad & b_{temp}.A \\ & \cup \{ \; <ts_{peg}, ts_{end}, A_{label}> : \\ & \qquad <b_{temp}.ts_{init}, ts_{end}, A_{label}> \in b_{temp}.A \} \\ & - \{ \; <ts_{start}, ts_{end}, A_{label}>: \\ & \qquad ts_{start} = b_{temp}.ts_{init} \}; \end{aligned}$$

$$b_{new}.ts_{init} := \quad ts_{peg};$$

$$b_{new}.TS_{term} := \begin{cases} b_{temp}.TS_{term} \cup \{ts_{peg}\} - \{b_{temp}.ts_{init}\} & \text{if } tb_{temp}.ts_{init} \in b_{temp}.TS_{term} \\ b_{temp}.TS_{term} & \text{otherwise} \end{cases}$$

$$Bgrove := \quad Bgrove \cup \{b_{new}\};$$

Here, again, state conflicts can arise between $b_{temp}.ts_{init}$ and the various elements of $ta.TS_{term}$. As with choice composition, we delay discussion of this issue until Chapter 6.

Now we can construct the timetree (*ta* ; *tb*):

$$(ta\ ;\ tb).TS := \quad ta.TS \cup \bigcup_{b_{temp} \in Bgrove} b_{temp}.TS$$

$$\begin{aligned}(ta\ ;\ tb).SCV := \quad & \{ <sc_{name}, SC_{r1} \cup SC_{r2}>: \\ & \quad <sc_{name}, SC_{r1}> \in ta.SCV \wedge \\ & \quad <sc_{name}, SC_{r2}> \in tb.SCV \} \\ \cup \quad & \{ <sc_{name}, SC_{r1}>: \\ & \quad <sc_{name}, SC_{r1}> \in ta.SCV \wedge \\ & \quad \nexists\, SC_{r2}: <sc_{name}, SC_{r2}> \in tb.SCV \} \\ \cup \quad & \{ <sc_{name}, SC_{r2}>: \\ & \quad <sc_{name}, SC_{r2}> \in tb.SCV \wedge \\ & \quad \nexists\, SC_{r1}: <sc_{name}, SC_{r1}> \in ta.SCV \}\end{aligned}$$

$$(ta\ ;\ tb).SCM := \quad ta.SCM \cup \bigcup_{b_{temp} \in Bgrove} b_{temp}.SCM$$

$$(ta\ ;\ tb).AV := \quad ta.AV \cup tb.AV$$

$$(ta\ ;\ tb).A := \quad ta.A \cup \bigcup_{b_{temp} \in Bgrove} b_{temp}.A$$

$$(ta\ ;\ tb).ts_{init} := \quad ta.ts_{init}$$

$$(ta\ ;\ tb).TS_{term} := \quad \bigcup_{b_{temp} \in Bgrove} b_{temp}.TS_{term}$$

The sets of timestates, activity arcs, and terminal timestates are collected from every timetree in *Bgrove*. The special $\cup$ notation reflects union over the set *Bgrove*, analogous to the Σ notation for summation.

Again, we can recast this definition in terms of d-restricted timetrees. If we restrict (*ta* ; *tb*) to tree-depth d, then the only terminal timestates of *ta* that will be incorporated into

(ta ; tb) are those of depth d or less. Since timetrees are finitely branching, the number of timestates of depth d or less is finite; so this constraint ensures that, for finite d, $Bgrove$ will contain a finite number of timetrees. For a particular terminal timestate in ta of depth n, only paths from tb of length $\leq d\text{-}n$ will be appended. This ensures that each b_{new}-$\in B_{grove}$ will be finite for finite d.

$tnew\rceil^d := tsclone(tt\rceil^d)$:

$$tnew\rceil^d.TS := \quad \{\ ts_{new} :\ ts_{new} = tscopy(ts_{old}) \wedge ts_{old} \in tt\rceil^d.TS\ \}$$

$$tnew\rceil^d.SCV := \quad tt\rceil^d.SCV$$

$$tnew\rceil^d.SCM := \quad \{\ \langle tscopy(ts_{old}), sc_{name}, sc_{value}\rangle : \langle ts_{old}, sc_{name}, sc_{value}\rangle \in tt\rceil^d.SCM\ \}$$

$$tnew\rceil^d.AV := \quad tt\rceil^d.AV$$

$$tnew\rceil^d.A := \quad \{\ \langle ts_{start.new}, ts_{end.new}, A_{label}\rangle :\ ts_{start.new} = tscopy(ts_{start.old}) \wedge ts_{end.new} = tscopy(ts_{end.old}) \wedge \langle ts_{start.old}, ts_{end.old}, A_{label}\rangle \in tt\rceil^d.A\ \}$$

$$tnew\rceil^d.ts_{init} := \quad tscopy(tt\rceil^d.ts_{init})$$

$$tnew\rceil^d.TS_{term} := \quad \{\ ts_{new} :\ ts_{new} = tscopy(ts_{old}) \wedge ts_{old} \in tt\rceil^d.TS_{term}\ \}$$

Bgrove:

$Bgrove := \emptyset;$

for each $ts_{peg} \in ta\rceil^{d}.TS_{term}$:

$Ad := depth(ts_{peg});$
$Bd := d - Ad;$

$b_{temp}\rceil^{Bd} := \quad tsclone(tb\rceil^{Bd});$

$b_{new}\rceil^{Bd}.\mathrm{TS} := \quad b_{temp}\rceil^{Bd}.TS \cup \{ts_{peg}\} - \{b_{temp}.ts_{init}\};$

$b_{new}\rceil^{Bd}.SCV := \quad b_{temp}\rceil^{Bd}.SCV;$

$$
\begin{aligned}
b_{new}\rceil^{Bd}.SCM := \quad & b_{temp}\rceil^{Bd}.SCM \\
\cup \; \{ \quad & \langle ts_{peg}, SC_{name}, SC_{value} \rangle : \\
& \langle b_{temp}\rceil^{Bd}.ts_{init}, SC_{name}, SC_{value} \rangle \in b_{temp}\rceil^{Bd}.SCM \} \\
- \; \{ \quad & \langle b_{temp}\rceil^{Bd}.ts_{init}, SC_{name}, SC_{value} \rangle \in b_{temp}\rceil^{Bd}.SCM \};
\end{aligned}
$$

$b_{new}\rceil^{Bd}.AV := \quad b_{temp}\rceil^{Bd}.AV;$

$$
\begin{aligned}
b_{new}\rceil^{Bd}.A := \quad & b_{temp}\rceil^{Bd}.A \\
\cup \; \{ \quad & \langle ts_{peg}, ts_{end}, A_{label} \rangle : \\
& \langle b_{temp}\rceil^{Bd}.ts_{init}, ts_{end}, A_{label} \rangle \in b_{temp}\rceil^{Bd}.A \; \} \\
- \; \{ \quad & \langle ts_{start}, ts_{end}, A_{label} \rangle : \\
& ts_{start} = b_{temp}\rceil^{Bd}.ts_{init} \; \};
\end{aligned}
$$

$b_{new}\rceil^{Bd}.ts_{init} := \quad ts_{peg};$

$$
b_{new}\rceil^{Bd}.TS_{term} :=
\begin{cases}
b_{temp}\rceil^{Bd}.TS_{term} \cup \{ts_{peg}\} - \{b_{temp}\rceil^{Bd}.ts_{init}\} & \text{if } tb_{temp}.ts_{init} \in b_{temp}\rceil^{Bd}.TS_{term} \\
b_{temp}\rceil^{Bd}.TS_{term} & \text{otherwise}
\end{cases}
$$

$Bgrove := Bgrove \cup \{b_{new}\rceil^{Bd}\};$

$(ta \; ; \; tb)\rceil^d$:

$$(ta \; ; \; tb)\rceil^d.TS := \quad ta\rceil^d.TS \cup \bigcup_{b_{temp} \in Bgrove} b_{temp}.TS$$

$$\begin{aligned}(ta \; ; \; tb)\rceil^d.SCV := \quad & \{ \, <sc_{name}, SC_{r1} \cup SC_{r2}>: \\ & \quad <sc_{name}, SC_{r1}> \in ta\rceil^d.SCV \wedge \\ & \quad <sc_{name}, SC_{r2}> \in tb\rceil^d.SCV \, \} \\ \cup \quad & \{ \, <sc_{name}, SC_{r1}>: \\ & \quad <sc_{name}, SC_{r1}> \in ta\rceil^d.SCV \wedge \\ & \quad \nexists \; SC_{r2}: <sc_{name}, SC_{r2}> \in tb\rceil^d.SCV \, \} \\ \cup \quad & \{ \, <sc_{name}, SC_{r2}>: \\ & \quad <sc_{name}, SC_{r2}> \in tb\rceil^d.SCV \wedge \\ & \quad \nexists \; SC_{r1}: <sc_{name}, SC_{r1}> \in ta\rceil^d.SCV \, \}\end{aligned}$$

$$(ta \; ; \; tb)\rceil^d.SCM := \quad ta\rceil^d.SCM \cup \bigcup_{b_{temp} \in Bgrove} b_{temp}.SCM$$

$$(ta \; ; \; tb)\rceil^d.AV := \quad ta\rceil^d.AV \cup tb\rceil^d.AV$$

$$(ta \; ; \; tb)\rceil^d.A := \quad ta\rceil^d.A \cup \bigcup_{b_{temp} \in Bgrove} b_{temp}.A$$

$$(ta \; ; \; tb)\rceil^d.ts_{init} := \quad ta\rceil^d.ts_{init}$$

$$(ta \; ; \; tb)\rceil^d.TS_{term} := \quad \bigcup_{b_{temp} \in Bgrove} b_{temp}.TS_{term}$$

If either or both of *ta* and *tb* are infinite timetrees, $(ta\ ;\ tb)$ is defined as $\lim_{d \to \infty} (ta\ ;\ tb)\rceil^d$. For finite *ta* and *tb* of tree-depth *m* and *n* respectively, $(ta\ ;\ tb)\rceil^d = (ta\ ;\ tb)$ for $d \geq m + n$, so again $\lim_{d \to \infty} (ta\ ;\ tb)\rceil^d = (ta\ ;\ tb)$.

5.3.3 Interleaving

Many contemporary interface styles allow a user to switch back and forth among tasks in progress, instead of requiring the user to finish one task before starting another. For example, in systems with multiple windows, the user may switch back and forth between (say) a mail program in one window, a calendar manager in another, and a spreadsheet in a third. The UAN provides an *interleavability* operator to describe this kind of task composition.

The timetree model uses an *interleaving* operator to represent this composition form. Interleaving composition combines two timetrees *ta* and *tb* to yield a new timetree $(ta \Leftrightarrow tb)$ that describes all possible interleavings of all possible sequences of activity described by *ta* and *tb*. This pattern of combination is similar to serializable concurrency; one can visualize *ta* and *tb* as describing two tasks whose activities can be interleaved (but cannot overlap) over time, and $(ta \Leftrightarrow tb)$ as describing the possible absolute orderings of their activities.

Timetrees to describe interleaving may potentially grow very quickly. A first intuitive glance might indicate that such a combination of two timetrees could be formed by taking the cross product of the timestate sets of *ta* and *tb* to yield a new set of timestates, each encoding a timestate of *ta* and *tb*, then connecting these timestates with activities from *ta* and *tb*. But this does not work; if, for example, *ta* matches only the activity *X* and *tb* matches only the activity *Y*, $(ta \Leftrightarrow tb)$ could match *X; Y* or *Y; X*, and must have a different terminal timestate for each path. A simple cross-product encoding would yield only one terminal timestate.

The construction of $(ta \Leftrightarrow tb)$ is more complex than our choice or sequence constructions. We use recursion over nodes of the new tree; at each timestate, we create arcs and destination nodes to reflect the activities possible at that timestate, then recurse

on the new destination nodes. Each node in the new tree encodes a pair of states from *ta* and *tb*, as we alluded above; but any given pair of states may be encoded by more than one timestate in ($ta \Leftrightarrow tb$), reflecting different possible interleavings of the activities leading up to that timestate. The recursion terminates at nodes reflecting combinations of leaf nodes from *ta* and *tb*; once all paths to every pairing of leaf nodes have been generated, the algorithm itself terminates. Figures 5.4-5.7 illustrate this process step by step.

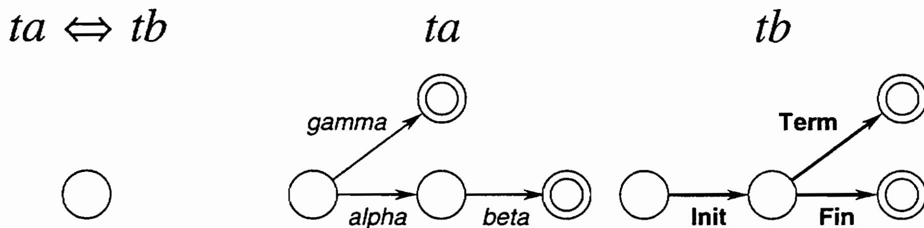


Figure 5.4. Timetree interleaving before first step.

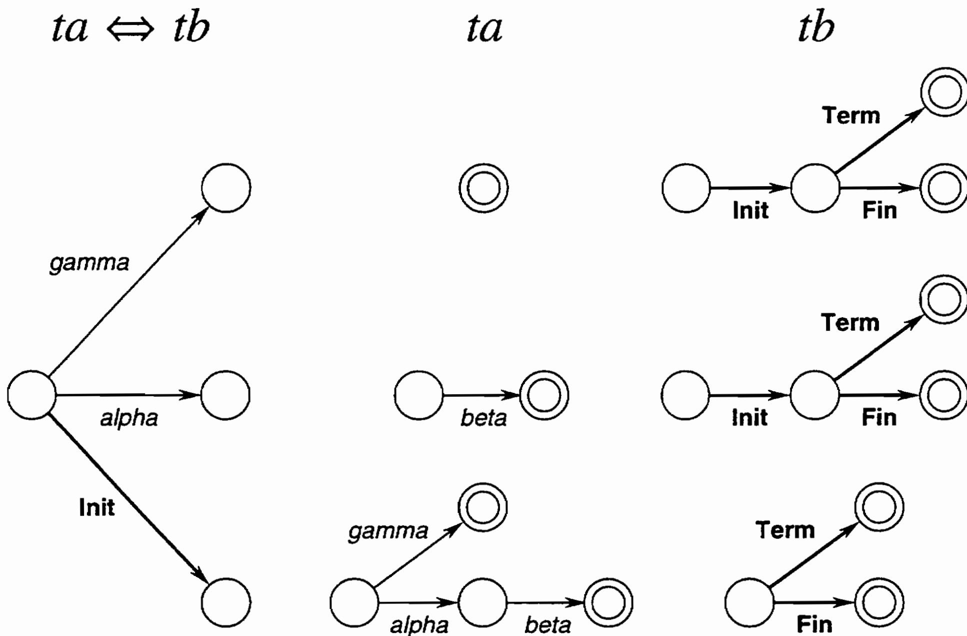


Figure 5.5. Timetree interleaving after one step.

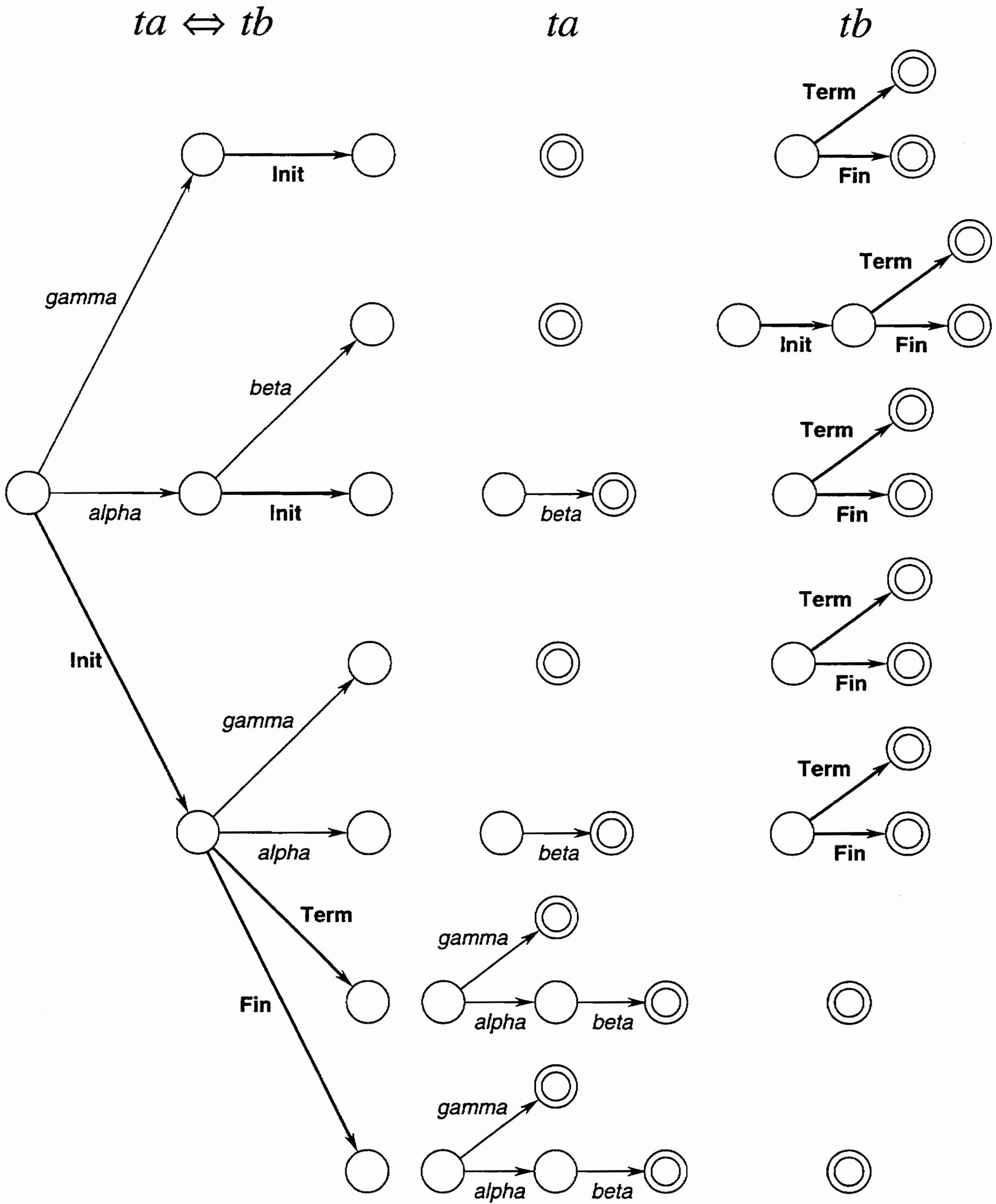


Figure 5.6. Timetree interleaving after two steps.

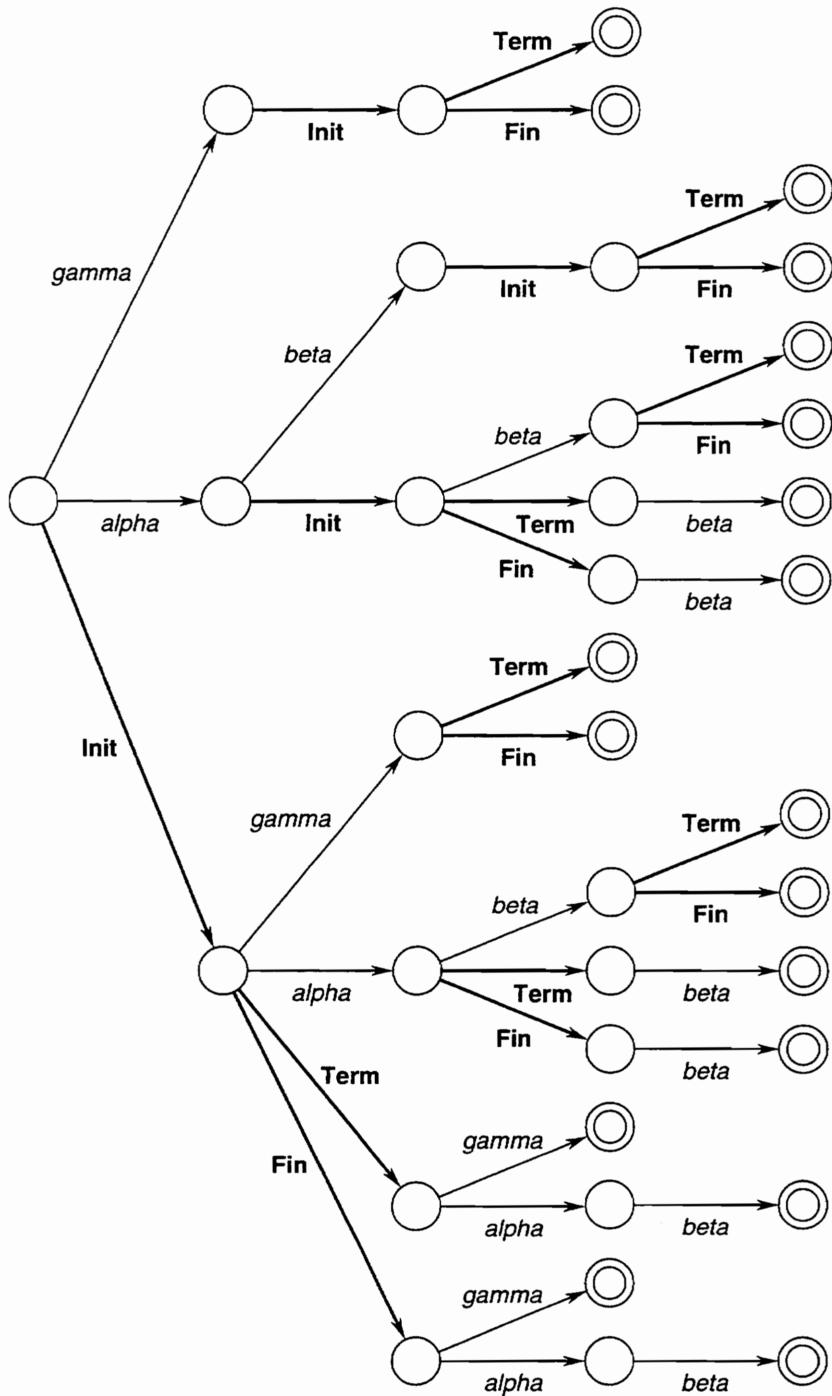


Figure 5.7. Timetree interleaving after completion.

We initialize ($ta \Leftrightarrow tb$) to contain an initial timestate, and form its state component vocabulary and activity vocabulary from *ta* and *tb*. All other components are empty at the start.

$$(ta \Leftrightarrow tb).TS := \{ts_{init.new}\}$$

$$\begin{aligned}(ta \Leftrightarrow tb).SCV := \\ & \{ <sc_{name}, SC_{r1} \cup SC_{r2}>: \\ & \qquad <sc_{name}, SC_{r1}> \in ta.SCV \wedge \\ & \qquad <sc_{name}, SC_{r2}> \in tb.SCV \} \\ \cup & \{ <sc_{name}, SC_{r1}>: \\ & \qquad <sc_{name}, SC_{r1}> \in ta.SCV \wedge \\ & \qquad \nexists\, SC_{r2}: <sc_{name}, SC_{r2}> \in tb.SCV \} \\ \cup & \{ <sc_{name}, SC_{r2}>: \\ & \qquad <sc_{name}, SC_{r2}> \in tb.SCV \wedge \\ & \qquad \nexists\, SC_{r1}: <sc_{name}, SC_{r1}> \in ta.SCV \}\end{aligned}$$

$$(ta \Leftrightarrow tb).SCM := \varnothing$$

$$(ta \Leftrightarrow tb).AV := ta.AV \cup tb.AV$$

$$(ta \Leftrightarrow tb).A := \varnothing$$

$$(ta \Leftrightarrow tb).ts_{init} := ts_{init.new}$$

$$(ta \Leftrightarrow tb).TS_{term} := \varnothing$$

Next, we define the recursive interleaving function. It refers to components from *ta*, *tb*, and ($ta \Leftrightarrow tb$). Parameter *istate* is the current timestate in ($ta \Leftrightarrow tb$), *astate* is a timestate in *ta*, and *bstate* is a timestate in *tb*. Function *tsnew*( ) creates a new timestate, similarly to *tscopy* in sequential combination.

```
interleave(istate, astate, bstate):

    -- first, add the current timestate to TSterm if it corresponds to terminal
       timestates in both source timetrees:
    if (astate ∈ ta.TSterm) ∧ (bstate ∈ tb.TSterm)
        then (ta ⇔ tb).TSterm := (ta ⇔ tb).TSterm ∪ {istate};

    -- initialize a set of argument lists for recursive calls at the end of this
       iteration:
    Statelabels := ∅;

    -- build state information for this timestate from the associated timestates
       in the source timetrees:
    (ta ⇔ tb).SCM := (ta ⇔ tb).SCM
                ∪ { <istate, sc_name, sc_value>:
                       <astate, sc_name, sc_value> ∈ ta.SCM }
                ∪ { <istate, sc_name, sc_value>:
                       <bstate, sc_name, sc_value> ∈ tb.SCM }

    -- for each arc leaving the current timestate in ta, create a new timestate in (ta ⇔
       tb) and an arc going to it. Add to argument list for recursion at end.
    for each <astate, adest, Label> ∈ ta.A
        newdest        := tsnew( );
        (ta ⇔ tb).TS   := (ta ⇔ tb).TS ∪ { newdest };
        (ta ⇔ tb).A    := (ta ⇔ tb).A ∪ { <istate, newdest, Label> };
        Statelabels    := Statelabels ∪ { <newdest, adest, bstate> };

    -- Perform the same iteration for each arc leaving the current timestate in
       tb:
    for each <bstate, bdest, Label> ∈ tb.A
        newdest        := tsnew( );
        (ta ⇔ tb).TS   := (ta ⇔ tb).TS ∪ { newdest };
        (ta ⇔ tb).A    := (ta ⇔ tb).A ∪ { <istate, newdest, Label> };
        Statelabels    := Statelabels ∪ { <newdest, astate, bdest> };

    -- recurse on each timestate added to (ta ⇔ tb). If astate and bstate were
       both leaf nodes, Statelabels is empty, and the recursion terminates.
    for each <newdest, newA, newB> ∈ Statelabels
        interleave(newdest, newA, newB);
```

To start the interleave operation, we invoke it with the roots of the three trees:

$interleave((ta \Leftrightarrow tb).ts_{init}, ta.ts_{init}, tb.ts_{init})$

In this algorithm, once more, state conflicts can arise: in this case, state conflicts between timestates from *ta* and *tb* that are represented by a single timestate in $(ta \Leftrightarrow tb)$. The same solutions mentioned for choice composition also apply here, and again will be discussed in Chapter 6.

To recast the definition of $(ta \Leftrightarrow tb)$ in terms of *d*-restricted timetrees, we need only rewrite *interleave*; the initial state of $(ta \Leftrightarrow tb)\rceil^{d}$ is identical to that of $(ta \Leftrightarrow tb)$.

$interleave\rceil^{d}(istate, astate, bstate)$:

> -- *first, get the depths of* astate *and* bstate*:*
> $Ad := depth(astate);$
> $Bd := depth(bstate);$
>
> -- *add the current timestate to* $\mathrm{TS}_{\mathrm{term}}$ *if it corresponds to terminal timestates in both source timetrees:*
> **if** $(astate \in ta\rceil^{Ad}.TS_{term}) \wedge (bstate \in tb\rceil^{Bd}.TS_{term})$
>   **then** $(ta \Leftrightarrow tb)\rceil^{d}.TS_{term} := (ta \Leftrightarrow tb)\rceil^{d}.TS_{term} \cup \{istate\};$
>
> -- *build state information for this timestate from the associated timestates in the source timetrees:*
> $(ta \Leftrightarrow tb)\rceil^{d}.SCM := (ta \Leftrightarrow tb)\rceil^{d}.SCM$
> $\quad\quad \cup \; \{ \; <istate, sc_{name}, sc_{value}>: \; <astate, sc_{name}, sc_{value}> \in ta\rceil^{Ad}.SCM \; \}$
> $\quad\quad \cup \; \{ \; <istate, sc_{name}, sc_{value}>: \; <bstate, sc_{name}, sc_{value}> \in tb\rceil^{Bd}.SCM \; \}$
>
> -- *quit if* istate *has depth* d*:*
> **if** $(depth(istate) = d)$
>   **then return**;
>
> -- *initialize a set of argument lists for recursive calls at the end of this iteration:*
> $Statelabels := \varnothing;$

-- *for each arc leaving the current timestate in* ta, *create a new timestate in* (ta $\Leftrightarrow$ tb) *and an arc going to it. Add an argument list for recursion at end.*

for each *<astate, adest, Label>* $\in ta\rceil^{Ad+1}.A$

newdest := *tsnew*( );
$(ta \Leftrightarrow tb)\rceil^{d}.TS := (ta \Leftrightarrow tb)\rceil^{d}.TS \cup \{\ newdest\ \};$
$(ta \Leftrightarrow tb)\rceil^{d}.A := (ta \Leftrightarrow tb)\rceil^{d}.A \cup \{$ *<istate, newdest, Label>* $\};$
Statelabels := *Statelabels* $\cup$ { *<newdest, adest, bstate>* };

-- *Perform the same iteration for each arc leaving the current timestate in* tb*:*

for each *<bstate, bdest, Label>* $\in tb\rceil^{Bd+1}.A$

newdest := *tsnew*( );
$(ta \Leftrightarrow tb)\rceil^{d}.TS := (ta \Leftrightarrow tb)\rceil^{d}.TS \cup \{\ newdest\ \};$
$(ta \Leftrightarrow tb)\rceil^{d}.A := (ta \Leftrightarrow tb)\rceil^{d}.A \cup \{$ *<istate, newdest, Label>* $\};$
Statelabels := *Statelabels* $\cup$ { *<newdest, astate, bdest>* };

-- *recurse on each timestate added to* (ta $\Leftrightarrow$ tb)*. If* astate *and* bstate *were both leaf nodes,* Statelabels *is empty, and the recursion terminates.*

for each *<newdest, newA, newB>* $\in$ *Statelabels*

interleave(*newdest*, *newA*, *newB*);

Again, we start the operation by invoking it on the three trees:

$$interleave\rceil^{d}((ta \Leftrightarrow tb)\rceil^{d}.ts_{init},\ ta\rceil^{d}.ts_{init},\ tb\rceil^{d}.ts_{init})$$

As usual, if either or both of *ta* and *tb* are infinite timetrees, $(ta\ ;\ tb)$ is defined as $\lim_{d\to\infty} (ta\ ;\ tb)\rceil^{d}$. For finite *ta* and *tb* of tree-depth *m* and *n* respectively, $(ta\ ;\ tb)\rceil^{d} = (ta\ ;\ tb)$ for $d \ge m + n$, so again $\lim_{d\to\infty} (ta\ ;\ tb)\rceil^{d} = (ta\ ;\ tb)$.

5.3.4 Concurrency

In interleaved task performance, the ordering of actions can vary, but actions cannot overlap one another. In some interface forms, this restriction is unacceptable. For example, an interface might allow a user to perform two tasks simultaneously, using two separate input devices (say, two pointing devices, or a pointer and a foot pedal). If the user can manipulate both devices at the same time, and the system can respond to both devices at the same time, an interface description for that system must represent true concurrency.

[Hoare, 1985] views concurrency much as we presented interleaving in the previous section. In his model (CSP), events are instantaneous, and so it is possible to neglect the issue of "true" simultaneity. In fact, modern physics teaches us that the concept of simultaneity in the real world is chimerical.

On the other hand, Hoare's approach also allows him to discount the possibility of *overlapping* events. We do not have this luxury, because, as we alluded previously, the activities represented in timetrees need not be instantaneous. They can have a significant duration, and so they may overlap. However, we can take advantage of Hoare's prescription for representing extended actions:

> Extended or time-consuming actions should be represented by a pair of events, the first denoting its start and the second denoting its finish. The duration of an action is represented by the interval between the occurrence of its start event and the occurrence of its finish event; during such an interval, other events may occur. Two extended actions may overlap in time if the start of each one precedes the finish of the other. [p. 24]

To adapt this approach to the timetree model, we adopt a *tripartite activity view*, in which we represent an activity by its *initiation*, *continuation*, and *termination*. To prepare a timetree for concurrent composition, we first replace each activity a in AV with three new activities, $a_{initiate}$, $a_{continue}$, and $a_{terminate}$. We next replace each activity arc labeled with $\{A\}$ with *two* arcs, labeled $\{a_{initiate}\}$ and $\{a_{terminate}\}$, and joined by a new timestate. We do not allow this process on timetrees containing arcs labeled with multiple activities; since such activities are temporally unspecified, no detailed representation of their individual initiation and termination can be guaranteed accurate. To remedy this problem, the multiply labeled arc may be expanded to a level of detail sufficient to reveal the temporal relationship of the activities labeling it, or the arc may be relabeled with a single, abstract, compound activity hiding the detail of the multiple activities.

Once we have two trees with their activities represented by initiation and termination, we compose them by the interleaving operation of the previous section. This yields a tree in which activities can overlap, since the initiation of one activity can be interleaved between the initiation and termination of another.

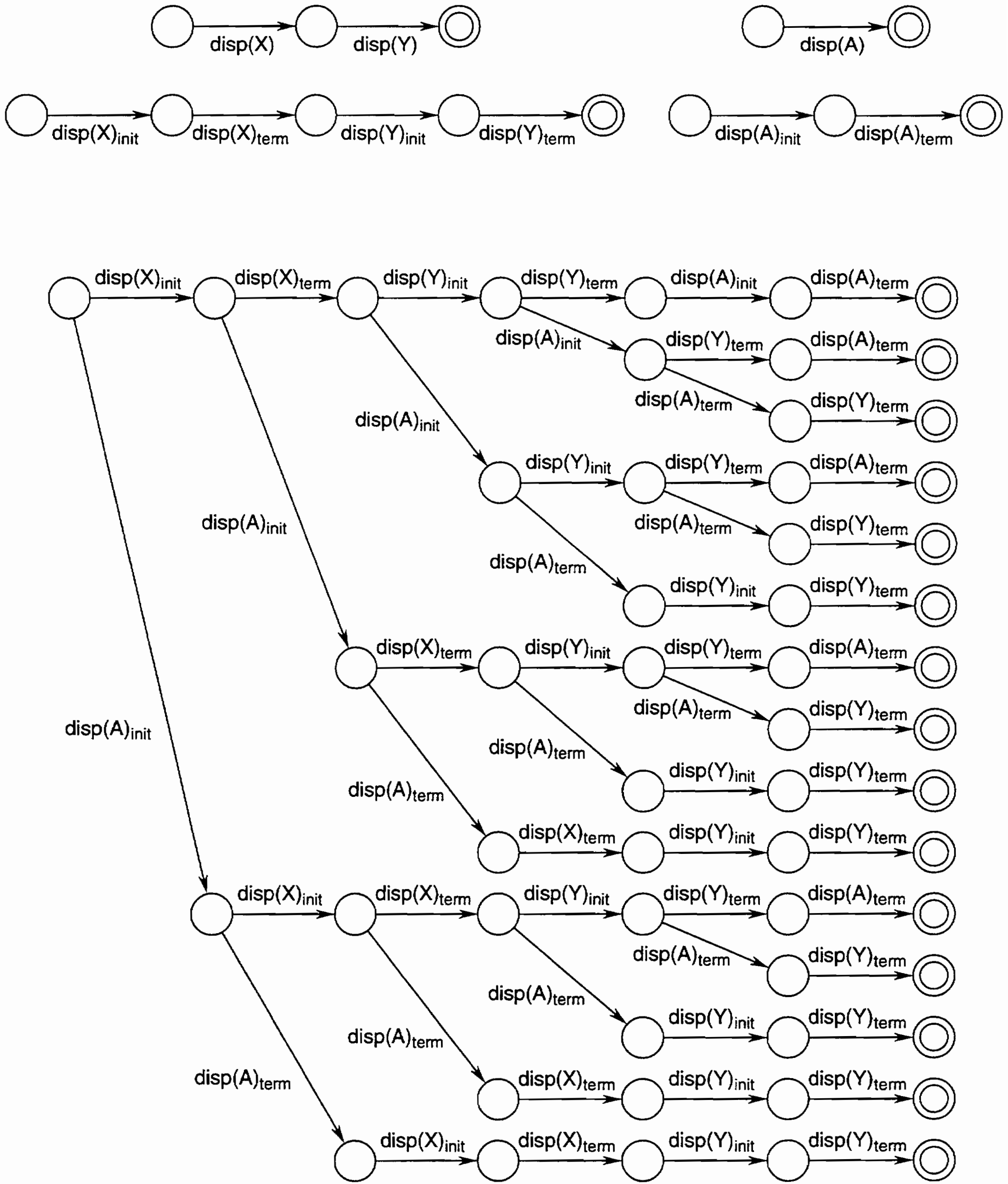


Figure 5.8. Partitioning and interleaving of concurrent activities.

Figure 5.8 illustrates the result of this composition for two timetrees, one with two sequential activities, the other with a single activity.

Finally, to indicate the continuing performance of an activity between its initiation and termination, we add continuations along all paths from a given activity's initiation to its termination. We do this by adding $a_{continue}$ to the label of each activity occuring between $a_{initiate}$ and $a_{terminate}$. While there is not room for this information in Figure 5.8, Figure 5.9 illustrates some of its paths with the added labels.

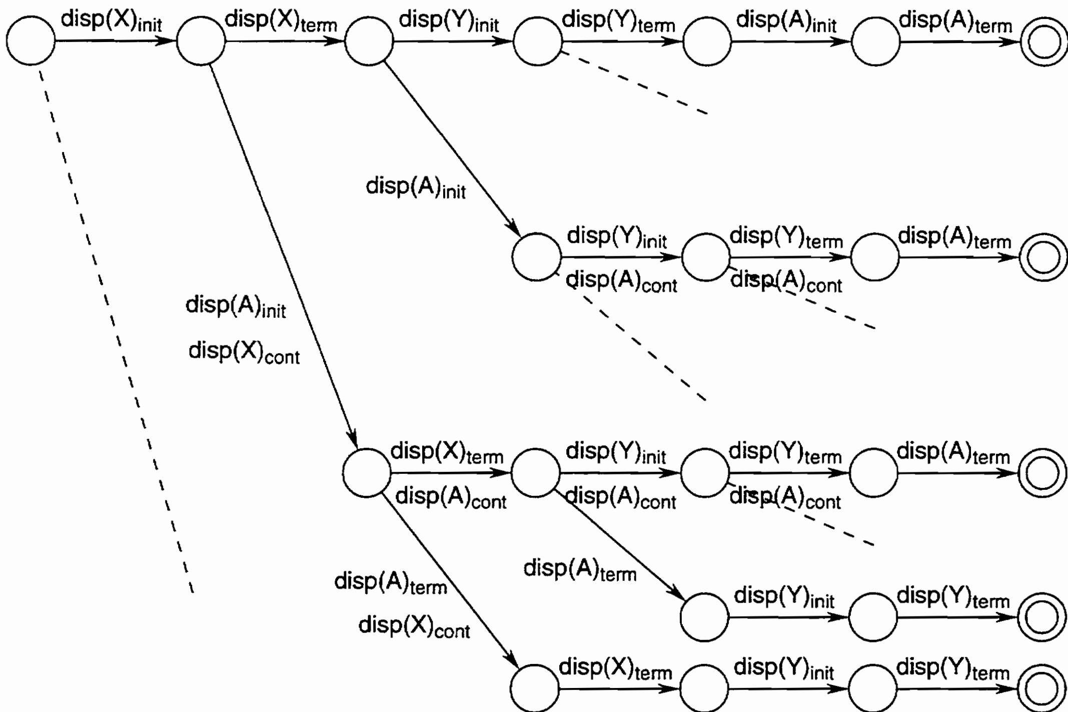


Figure 5.9. Concurrent composition with propagation of continuations.

The function *partition-activities*(*tt*) rewrites the timetree presented as its argument, replacing each activity in the vocabulary with initiation, continuation and termination activities, expanding each activity arc to a pair of arcs (for initiation and termination), and instantiating a new timestate at the junction of each arc pair. This function does not propagate state information into the new timestates, because state information *during* an

activity cannot generally be inferred from state information before or after the activity. Specifying state information *during* an activity requires a lower-level view of the activity itself. An example of this phenomenon appears in Section 6.3.

partition-activities(*tt*):

> -- *first, generate new activity vocabulary, and replace the old:*
> *NewAV* := { $a_{initiate}$, $a_{continue}$, $a_{terminate}$: $a \in tt.AV$ };
> *tt.AV* := *NewAV*;
>
> -- *replace each activity arc with pair of arcs for initiate and terminate,*
>   *and instantiate new timestates:*
> **for each** <*startstate*, *endstate*, { *act* }> ∈ *tt.A*
>   *interstate* := *tsnew*( );
>   *tt.A* := *tt.A* – { <*startstate*, *endstate*, { *act* }> }
>   ∪ { <*startstate*, *interstate*, { $act_{initiate}$ }> }
>   ∪ { <*interstate*, *endstate*, { $act_{terminate}$ }> };
>   *tt.TS* := *tt.TS* ∪ { *interstate* };

To insert continuation activities after two timetrees have been combined, *propagate-continuations*(*tt*, *state*) examines the label of each arc starting at one timestate in *tt*, then modifies the labels of the arcs leaving that arc's end timestate. If an arc's label contains $a_{initiate}$ or $a_{continue}$, and the following arc's label does not contain $a_{terminate}$, then $a_{continue}$ is added to that following arc's label. The algorithm then recurses on the children of the current timestate. The algorithm is started by invoking it on the initial timestate of the timetree: *propagate-continuations*(*tt*, $tt.ts_{init}$).

```
propagate-continuations(tt, state):

    -- initialize set of descendants of current node:
    Stateset := ∅;

    -- iterate over each activity arc starting at the current timestate:
    for each <state, endstate, Curlabel> ∈ tt.A

        -- add destination of current arc to Stateset:
        Stateset := Stateset ∪ { endstate };

        -- iterate over arcs starting at endstate:
        for each nextarc: <endstate, nextstate, Nextlabel> ∈ tt.A
            -- rewrite label of nextarc:
            Auglabel := Nextlabel ∪
                    { a_continue:    (a_initiate ∈ Curlabel ∨
                                      a_continue ∈ Curlabel)
                                   ∧ a_terminate ∉ Nextlabel };
            -- replace nextarc in tt.A:
            tt.A := tt.A - nextarc ∪ { <endstate, nextstate, Auglabel> };

    -- recurse on descendants of current node:
    for each tstate ∈ Stateset
        propagate-continuations(tt, tstate);
```

These algorithms allow us to define a new timetree (*ta* || *tb*) representing the concurrent composition of *ta* and *tb*. We apply *partition-activities* to *ta* and *tb*, build an intermediate representation of (*ta* || *tb*) using the interleave operator of the previous section, and complete the representation by applying *propagate-continuations* to the result.

```
partition-activities(ta);
partition-activities(tb);
(ta || tb) := (ta ⇔ tb);
propagate-continuations((ta || tb), (ta || tb).ts_init);
```

The *d*-restricted statement of this composition operator is complicated by *partition-activities*, which doubles the tree-depth of a finite timetree by replacing each activity arc with a sequence of two arcs. Thus, *partition-activities*($tt\rceil^{d}$) yields a timetree of tree-depth $2d$. But no modifications to *partition-activities* or *propagate-continuations* are necessary; the former simply doubles the depth of each existing timestate in the

timetree, and since the latter only rewrites arc *labels*, it does not change the depth of any timestate.

We can state the d-restricted version of concurrent composition as follows:

> $partition\text{-}activities(ta\rceil^{d})$;
> $partition\text{-}activities(tb\rceil^{d})$;
> $(ta \parallel tb)\rceil^{2d} := (ta \Leftrightarrow tb)\rceil^{2d}$;
> $propagate\text{-}continuations((ta \parallel tb)\rceil^{2d}, (ta \parallel tb)\rceil^{2d}.ts_{init})$;

Where either or both of ta and tb are infinite timetrees, $(ta \parallel tb)$ is defined as $\lim_{d \to \infty} (ta \parallel tb)\rceil^{d}$. Since concurrent composition uses the same algorithm as interleaved composition to generate a combined timetree, it exhibits similar depth limits when applied to finite timetrees: for ta and tb of tree-depth m and n respectively, $(ta \parallel tb)\rceil^{d} = (ta \parallel tb)$ for $d \geq 2m + 2n$, and so again $\lim_{d \to \infty} (ta \parallel tb)\rceil^{d} = (ta \parallel tb)$.

5.3.5 Disambiguation

The composition operators described above can lead to *ambiguous* timetrees: timetrees in which some pair of arcs leaving a single timestate share the same set of labeling activities. (Equivalently, we can say that there is at least one sequence of activities that matches more than one path from the initial timestate, or from any other single timestate.) Consider, for example, the choice operator applied to two tasks that share a common prefix. The resulting timetree will have two branches with identical labels originating at the initial timestate. In Section 3.2, Figures 3.8 and 3.9, we presented an example of an ambiguous timetree, and showed how it could be collapsed into an unambiguous form.

We formalize this operation in the function *disambiguate*. This function is applied to the root of a timetree. For each timestate, it condenses all equivalent activities and their destination timestates; it then recurses on each of the condensed destination timestates. Note that this is not strictly a *composition* operator, since it does not compose multiple timetrees or multiple copies of a single timetree. We present it here because it operates in

conjunction with the preceding composition operators, and because the concept of disambiguation affects our presentation of the composition operators that follow.

disambiguate(*tt*, *state*):

```
-- initialize a set of arcs leaving this timestate:
Arcset := ∅;

-- for each arc leaving the current timestate, see if it is present in Arcset. If it is,
   condense it with the arc already in Arcset; if it is not, add it to Arcset.
for each tarc: <state, endstate, Label> ∈ tt.A
    if ∃ arc ∈ Arcset: arc.A_label = Label
        then replace(tt, endstate, arc.ts_end)
        else Arcset := Arcset ∪ { tarc };

-- recurse on the end timestate of each arc in Arcset.
for each arc ∈ Arcset
    disambiguate(tt, arc.ts_end);
```

The *replace* function removes the redundant arc and its end timestate, replaces arcs originating at the redundant timestate with arcs originating at the retained timestate, and updates TS_{term} where necessary.

replace(*tt*, *losestate*, *keepstate*):

> -- *remove the redundant arc leading to* losestate.
> *tt.A* := *tt.A* - { <ts_{start}, *losestate*, A_{label}> };
>
> -- *replace each arc leaving* losestate *with an identically labeled arc leaving* keepstate.
> **for each** *tarc*: <*losestate*, *deststate*, *Label*> ∈ *tt.A*
>   *tt.A* := *tt.A* - { *tarc* };
>   *tt.A* := *tt.A* ∪ { <*keepstate*, *deststate*, *Label*> };
>
> -- *copy* losestate*'s state component value assignments to* keepstate.
> **for each** <*losestate*, sc_{name}, sc_{value}> ∈ *tt.SCM*
>   *tt.SCM* := *tt.SCM* - { <*losestate*, sc_{name}, sc_{value}> };
>   *tt.SCM* := *tt.SCM* ∪ { <*keepstate*, sc_{name}, sc_{value}> };
>
> -- *if* losestate *was a terminal timestate, remove it from* TS_{term}. *If so, and* keepstate *was not already a terminal timestate, make it one.*
> **if** *losestate* ∈ $tt.TS_{term}$ **then**
> $tt.TS_{term}$:= $tt.TS_{term}$ - { *losestate* };
>   **if** *keepstate* ∉ $tt.TS_{term}$
>   **then** $tt.TS_{term}$:= $tt.TS_{term}$ ∪ { *keepstate* };
>
> -- *remove* losestate *from* TS.
> *tt.TS* := *tt.TS* - { *losestate* };

This algorithm disregards the possibility of state conflicts among the timestates that get condensed. We address this issue in detail in Chapter 6.

Neither *disambiguate* or *replace* need be modified to support *d*-restriction. Simply passing them a *d*-restricted timetree will suffice. Since timetrees are finitely branching, *disambiguate* applied to a *d*-restricted timetree only needs to process a finite number of arcs and timestates; since *replace* does not add timestates or change the depth of any existing timestate, it does not increase the tree-depth of the timetree. In other words, *disambiguate*($tt\rceil^{d}$, $tt.ts_{init}$) still terminates after a finite number of steps to yield a disambiguated timetree *tt* that still has tree-depth $\leq d$.

Note that this operation does not change the behavior that a timetree describes. A timetree describes a set of possible sequences of activity; each path from the initial timestate to a terminal timestate represents one possible sequence. In other words, the

history (Section 5.1) of each terminal timestate represents a possible sequence of activities. Although *disambiguate* collapses activities together breadthwise, it does not alter sequences of activity, and so it *does not alter the history of any timestate*, and so it does not alter the behavior described by a timetree.

5.3.6 Closure

Some interactive tasks allow a user to perform an action any number of times. For example, in a word processor, the user can usually change the font or type size of a selected block repeatedly; each change may erase or add to the effects of previous changes. To describe this situation, behavioral notations must have a way to specify that an action can take place "zero or more times." (In conjunction with sequence, this notation also allows specification of "one or more times" and other related forms.)

In a regular expression, Kleene star-closure denotes zero or more repetitions of a pattern or element. Similarly, *closure* composition over a timetree *ta* yields a new timetree ta^* that describes all possible paths consisting of zero or more concatenated possible paths through *ta*. We construct ta^* using repeated concatenation and choice. Informally, ta^* is the limit of the series

$$Null \mid (ta \mid ((ta \; ; \; ta) \mid ((ta \; ; \; ta \; ; \; ta) \mid \ldots$$

The timetree corresponding to this structure has an infinite depth and an infinite number of terminal timestates, even if *ta* is finite. The only exception is the null timetree; $Null^*$ is still *Null*. (More precisely, this is true of any timetree with $TS = \{ts_{init}\}$, $TS_{term} = \{ts_{init}\}$, and $A = \varnothing$; if the lone timestate has state information associated with it, or if the state component vocabulary or activity vocabulary are nonempty, the timetree is still unchanged by closure.)

The description above suggests that the timetree for ta^* must be infinitely branching, since ta^* is described by an infinite series of choices. In fact, though, disambiguation collapses these choices back to a single sequence, with each iteration in the sequence starting at a terminal timestate. To avoid infinite branching, we generate ta^* in disambiguated form; since ta^* is by definition nonterminating, we present only a *d*-restricted generating algorithm.

We adapt the algorithm for (*ta* ; *tb*), since closure is closely related to sequence. Our generation of (*ta* ; *tb*) involved construction of *Bgrove*, a set of timetrees representing the copies of *tb* to be appended to each terminal timestate of *ta*. For ta^*, we similarly generate *Agrove*, representing copies of *ta* to be appended to the terminal timestates of *ta*. Here, though, we iteratively add copies of *ta* to each frontier terminal timestate in *Agrove*. This process stops when we have appended a copy of *ta* to every terminal node of depth less than *d* in *Agrove*.

We maintain a set *Curterms* of the terminal timestates in *Agrove* that must be extended; within an iteration, we build up the set of terminals for the next iteration in *Newterms*. When terminal timestates from a new copy of *ta* are added to *Newterms*, we omit the initial timestate of the new copy, even if it is a member of TS_{term}. (Effectively, we are replacing the initial timestate of the new copy with the terminal timestate to which it is being attached.) For this reason, the depth of the terminal timestates being added to *Newterms* increases with each iteration; eventually, there will be no terminal timestates of depth less than *d*, *Curterms* will be empty, and the algorithm will terminate.

Once we have constructed *Agrove*, we use it to build ta^* analogously to the construction of (*ta* ; *tb*).

Agrove:

$Agrove := \emptyset;$
$Curterms := ta\rceil^{d}.TS_{term};$

while $Curterms \neq \emptyset$ **do**

$Newterms := \emptyset;$
for each $ts_{peg} \in Curterms$:
$Ad := d - depth(ts_{peg});$
$a_{temp}\rceil^{Ad} := tsclone(ta\rceil^{Ad});$

$a_{new}\rceil^{Ad}.TS := a_{temp}\rceil^{Ad}.TS \cup \{ts_{peg}\} - \{a_{temp}.ts_{init}\};$

$a_{new}\rceil^{Ad}.AV := a_{temp}\rceil^{Ad}.AV;$

$$
\begin{aligned}
a_{new}\rceil^{Ad}.A := \; & a_{temp}\rceil^{Ad}.A \\
\cup \; & \{ <ts_{peg}, ts_{end}, A_{label}> : \\
& \quad <a_{temp}\rceil^{Ad}.ts_{init}, ts_{end}, A_{label}> \in a_{temp}\rceil^{Ad}.A \} \\
- \; & \{ <ts_{start}, ts_{end}, A_{label}>: \\
& \quad ts_{start} = a_{temp}\rceil^{Ad}.ts_{init} \};
\end{aligned}
$$

$a_{new}\rceil^{Ad}.ts_{init} := ts_{peg};$

$$
a_{new}\rceil^{Ad}.TS_{term} :=
\begin{cases}
a_{temp}\rceil^{Ad}.TS_{term} \cup \{ts_{peg}\} - \{a_{temp}\rceil^{Ad}.ts_{init}\} & \text{if } a_{temp}.ts_{init} \in a_{temp}\rceil^{Ad}.TS_{term} \\
a_{temp}\rceil^{Ad}.TS_{term} & \text{otherwise}
\end{cases}
$$

-- add terminal timestates other than the initial timestate to Newterms*:*
$Newterms := Newterms \cup a_{temp}\rceil^{Ad}.TS_{term} - \{a_{temp}\rceil^{Ad}.ts_{init}\};$

$Agrove := Agrove \cup \{a_{new}\rceil^{Ad}\};$

-- replace list of current terminal timestates for next iteration:
$Curterms := Newterms;$

$(ta^*)\rceil^d$:

$$(ta^*)\rceil^d.TS := \quad ta\rceil^d.TS \cup \bigcup_{a_{temp} \in Agrove} a_{temp}.TS$$

$$(ta^*)\rceil^d.AV := \quad ta\rceil^d.AV$$

$$(ta^*)\rceil^d.A := \quad ta\rceil^d.A \cup \bigcup_{a_{temp} \in Agrove} a_{temp}.A$$

$$(ta^*)\rceil^d.ts_{init} := \quad ta\rceil^d.ts_{init}$$

$$(ta^*)\rceil^d.TS_{term} := \quad \{\ ta\rceil^d.ts_{init}\ \} \cup ta\rceil^d.TS_{term} \cup \bigcup_{a_{temp} \in Agrove} a_{temp}.TS_{term}$$

In this composition operation, as in the previous ones, ta^* is defined as $\lim_{d \to \infty} (ta^*)\rceil^d$. If ta contains no activities, $ta.TS_{term}$ can contain only $ta.ts_{init}$, so the algorithm to compute $Agrove$ will terminate after one iteration and $ta^*\rceil^d = ta^* = ta$ for all $d \geq 0$. Otherwise, if ta contains terminal timestates of nonzero depth, there is no finite d for which $ta^*\rceil^d = ta^*$.

5.4 Abstractions

These composition operators preserve most of the structure of their constituent timetrees, although the disambiguation operator can remove some of it. It is sometimes useful to reduce the size or complexity of a timetree. We now present several operators to rewrite a timetree into a simpler form. We have already hinted at some of these operators in Chapter 3.

5.4.1 Path pruning

This is the simplest way to reduce the size of a timetree. In its most basic form, we pick one timestate and eliminate the activity leading to that timestate and the subtree rooted at it. This eliminates all possible sequences of behavior following the activity.

prune-state(*tt*, *state*):

-- *remove the activity arc leading to this timestate:*
$tt.A := tt.A - \{ <ts_{start}, state, A_{label}> \};$

-- *remove this timestate from* TS, *and from* TS_{term} *if it is present:*
$tt.TS := tt.TS - \{ state \};$

$tt.TS_{term} := tt.TS_{term} - \{ state \};$

-- *remove the state information associated with this timestate:*
$tt.SCM := tt.SCM - \{ <state, sc_{name}, sc_{value}> \in tt.SCM \};$

-- *for each arc leaving this timestate, remove the arc, then recursively prune the subtree rooted at its end timestate.*

for each *tarc*: $<state, endstate, Label> \in tt.A$
$tt.A := tt.A - \{ tarc \};$
prune-state(*tt*, *endstate*);

To *d*-restrict this operation, we simply apply it to a depth-restricted timetree; since it does not add timestates, it cannot increase the depth of the timetree. Thus, *prune-state*($tt\rceil^{d}$, *state*) yields a pruned timetree *tt* that is still restricted to tree-depth *d*.

As an alternative, we can define an operation that removes a terminal timestate and all activities and timestates that lead only to it. If the terminal timestate is not a leaf — that is, if there are other activities leading *from* it — we only remove it from TS_{term}, without disrupting the rest of the timetree. This eliminates one possible behavior sequence described by the timetree.

prune-path(*tt*, *state*):

-- *remove this timestate from* TS_{term}:
$tt.TS_{term} := tt.TS_{term} - \{ state \}$;

-- *back up along the path that leads to* state *until we reach an earlier terminal timestate along the path or a timestate with other branches.*
while $state \notin tt.TS_{term} \wedge \nexists <state, ts_{end}, A_{label}> \in tt.A$

-- *remove the state information associated with this timestate:*
$tt.SCM := tt.SCM - \{ <state, sc_{name}, sc_{value}> \in tt.SCM \}$;

-- *remove this timestate from* TS:
$tt.TS := tt.TS - \{ state \}$;

-- *remove the arc leading to this timestate*:
$tt.A := tt.A - \{ <startstate, state, A_{label}> \}$;

-- *repeat at the preceding timestate*:
state := *startstate*;

This algorithm is inherently *d*-restricted; since it works backward from *state*, it sees no part of *tt* of depth greater than *d*.

5.4.2 Task-like abstractions

The UAN supports hierarchical structure by allowing a group of actions or subtasks to be abstracted into a higher-level task. We can represent this in a timetree by applying two abstraction techniques. The first, *condense-seq*, replaces a sequence of activities with a single activity, labeled by a new name; the second, *condense-choice*, replaces all the activities starting from a single timestate by a single activity, labeled by a new name.

The algorithm for *condense-seq* resembles that for *prune-path* (Section 5.4.1). We start at the end timestate of the last activity in the sequence and work backward, deleting activities and timestates not shared by other branches of the timetree. We preserve the end timestate, and add a new activity arc leading to it from the start timestate of the first activity in the sequence. Again, since the algorithm works backward through *Path*, it is inherently *d*-restricted.

condense-seq(*tt*, *Path*, *taskname*):

-- Path *is a set of activity arcs from tt.*A. *First, find the first and last timestates along the path.*

firststate := (*state*: $\exists$ <*state*, ts_{end}, A_{label}> $\in$ *Path* $\wedge$
$\nexists$ <ts_{start}, *state*, A_{label}> $\in$ *Path*);

finalstate := (*state*: $\exists$ <ts_{start}, *state*, A_{label}> $\in$ *Path* $\wedge$
$\nexists$ <*state*, ts_{end}, A_{label}> $\in$ *Path*);

-- *find our starting point for the walkback, then delete the last activity in* Path.
curstate := *state*: $\exists$ <*state*, *finalstate* , A_{label}> $\in$ *Path*;
tt.A := *tt.A* - { <*curstate*, *finalstate*, A_{label}> $\in$ *Path* };

-- *back up along the path that leads to* finalstate *until we reach* firststate.
repeat

-- *don't remove things if there are alternate paths from this node.*
if $\nexists$ <*curstate*, ts_{end}, A_{label}> $\in$ *tt.A*
then

-- *remove the arc in* Path *leading from this timestate*:
tt.A := *tt.A* - { <*prevstate*, *curstate*, A_{label}> };

-- *remove the state information associated with this timestate:*
tt.SCM := *tt.SCM* - { <*curstate*, sc_{name}, sc_{value}> $\in$ *tt.SCM* };

-- *remove this timestate from tt.*TS:
tt.TS := *tt.TS* - { *curstate* };

-- *repeat at the preceding timestate*:
curstate := *prevstate*;

until *curstate* = *firststate*;

-- *add* taskname *to the activity vocabulary.*
tt.AV := *tt.AV* $\cup$ { *taskname* };

-- *add the abstracted activity labeled with* taskname.
tt.A := *tt.A* $\cup$ { <*firststate*, *finalstate*, { *taskname* }> };

Note that the new, abstract activity may give no indication whether it shares a common prefix with any other activities. This is consistent with the overall approach of the UAN, and for that matter any other representation; common prefixes can be obscured at higher levels of abstraction. Further, this is not normally an issue, since *condense-seq* is ordinarily applied to every path (or at least a prefix of every path) leaving one particular

timestate. This is clearest if we consider applying *condense-seq* to every path from the initial timestate to a terminal timestate in an entire timetree. The product will be a timetree in which every path is a single activity, starting at the initial timestate and ending at a terminal timestate. For example, applying *condense-seq* to every path from the initial timestate to a terminal timestate in the timetree of Figure 3.9 yields the timetree of Figure 3.6.

The algorithm for *condense-choice* resembles *disambiguate* from Section 5.3.5. It involves replacing a set of activities and their end timestates with a single abstract activity and end timestate. The algorithm assumes that all the activities start at a single timestate. We use the *replace* function to remove the activities that are abstracted away, and *disambiguate* the tree rooted at the end timestate of the new activity. As in the disambiguation algorithm, we postpone discussion of state conflicts.

condense-choice(tt, Arcset, taskname):

> -- *pick one arc from* Arcset, *a set of arcs from* tt *all starting at the same timestate. We will substitute* taskname *for this arc's original label, keep its end timestate and the associated state information of its end timestate.*
>
> *newarc* := <*startstate, endstate, Label*> ∈ *Arcset*;
> *Arcset* := *Arcset* - *newarc*;
>
> -- *change the label for the new activity.*
> *tt.A* := *tt.A* - { *newarc* } ∪ { <*startstate, endstate*, { *taskname* }>};
>
> -- *add the new* taskname *to the activity vocabulary.*
> *tt.AV* := *tt.AV* ∪ { *taskname* };
>
> -- *for each arc leaving the current timestate, see if it is present in* Arcset. *If it is, condense it with the arc already in* Arcset*; if it is not, add it to* Arcset.
>
> **for each** <*startstate, losestate, Label*> ∈ *Arcset*
>   *replace(tt, losestate, endstate)*;
>
> -- *disambiguate the subtree now rooted at* endstate, *composed from the trees rooted at the end timestates of each arc in* Arcset.
> *disambiguate(tt, endstate)*;

To *d*-restrict this algorithm, we simply apply it to a *d*-restricted tree. This restricts the calls to *replace* and *disambiguate*; the rest of the algorithm is inherently bound to the same depth as the arcs in *Arcset*, that is, the depth of the endstates of those arcs.

5.4.3 State component propagation

The definitions of state component vocabulary and state component map do not require that every state component be assigned a value at every timestate. It is possible to define timetrees in which some state components have undefined values at certain timestates. In other words, it is not necessary that for every timestate $ts \in TS$ and every state component range assignment $\langle sc_{name}, SC_{range} \rangle \in SCV$ there is a state component mapping $\langle ts, sc_{name}, sc_{value} \rangle \in SCV$.

Some timetree applications require a way to *propagate* values from timestates where they are defined to timestates where they are undefined. The *propagate-forward* algorithm propagates values forward from timestates in which they are defined. This is suited to an interpretation in which a state component retains its value until some activity changes it. Parameter *tt* is the timetree on which to operate, *state* is the timestate at which to start (typically the root), and *name* is the name of the state component whose value is propagated.

propagate-forward(tt, state, name):

> -- *get the current value of state component* name *at timestate* state
> $curval := sc_{value}: \langle state, name, sc_{value} \rangle \in tt.SCM$;
>
> -- *copy the value to each successor timestate at which* name *is undefined;*
> *recurse on each successor timestate*
> **for each** $tarc: \langle state, endstate, Label \rangle \in tt.A$
>   **if** $\nexists \langle endstate, name, sc_{value} \rangle \in tt.SCM$ **then**
> $tt.SCM := tt.SCM \cup \{ \langle endstate, name, curval \rangle \}$;
>   *propagate-forward(tt, endstate, name)*;

Like *prune-state* (Section 5.4.1), we *d*-restrict this operation by applying it to a depth-restricted timetree.

We can propagate values for all state components by applying this operation once for each state component having a defined value at a specified state:

> **for each** name: <state, name, sc_{value}> ∈ tt.SCM
>
> *propagate-forward*(*tt*, *state*, *name*);

5.4.4 State component pruning

To generate higher-level representations of an interaction, it is occasionally convenient to ignore some components of state explicitly represented in a timetree. We can easily remove all mention of an unwanted state component from *SCV* and *SCM*.

SC-remove(*tt*, sc_{name}):

> -- *remove* scname*'s state component range assignment and state component value assignments.*
>
> *tt.SCV* := *tt.SCV* - { <sc_{name}, SC_{range}> ∈ *tt.SCV* };
>
> *tt.SCM* := *tt.SCM* - { <*ts*, sc_{name}, sc_{value}> ∈ *tt.SCM* };

Here again, we accomplish depth restriction by *d*-restricting *tt*.

5.4.5 State component equivalence classes

At other times we want not to discard a state component, but to represent it at a higher level of abstraction. For example, we might want to represent cursor location in terms of the context or on-screen item (if any) in which it resides, rather than by an absolute location (see Sections 6.1.2 and 6.2.2). We can do this by defining equivalence classes over the range of the state component, and a function to map values from that range into their appropriate equivalence class. Assume a set $SC_{classes}$ representing the equivalence classes for our state component, which we will again call sc_{name}. Define a function *equiv*(value) that maps each value from SC_{range} to an element of $SC_{classes}$. Then we can replace the old elements of *SCV* and *SCM* mentioning sc_{name} with new elements referring to equivalence classes:

$$SCV := SCV - \{ <sc_{name}, SC_{range}> \in SCV \} \cup \{ <sc_{name}, SC_{classes}> \};$$

$$SCM := SCM - \{ <ts, sc_{name}, sc_{value}> \in SCM \} \cup \{ <ts, sc_{name}, equiv(sc_{value})>: <ts, sc_{name}, sc_{value}> \in SCM \};$$

Note that this approach maps each value from SC_{range} to a *single* equivalence class. If a value can match multiple classes — for example, where the classes indicate contexts of icons that can overlap — $SC_{classes}$ must contain sets of classes as its elements, and *equiv* must map values to sets. Since this operation only changes the number of members of *SCV*, and rewrites some members of *SCM* without changing the set's cardinality, it can be *d*-restricted by *d*-restricting the timetree being rewritten.

The components and operations described in this chapter provide a framework sufficient for modeling the entities and operations of the behavioral realm. In the next chapter, we demonstrate this sufficiency by using the timetree model to define the semantics of the UAN.

6 Formal Definition of UAN

Previous expositions of the UAN have demonstrated its use, defined in prose some of its components [Hartson, Siochi, & Hix, 1990], and formally defined some aspects of its temporal operators [Hartson & Gray, 1992]. So far, though, there has been no framework suitable for a unified formal definition of UAN components and operators. In this chapter, we show how the timetree model provides such a framework.

We address components of the UAN in terms of several categories. First, we describe how interface entities that appear in UAN descriptions — input devices, display objects, and so forth — are represented in terms of state components. Next, we give representations of user actions and system feedback in terms of timetree activities. We use both state components and activities to represent interface state and connection to computation. Finally, we represent viability conditions and UAN composition operators in terms of timetree composition operations.

6.1 Entities

UAN descriptions refer to certain *entities* that are manipulated by and perceived by one or more interacting parties. Input, the transmission of information from a user to the system, takes place through *input devices* such as keys or a mouse; output, the presentation of information from the system to a user, is specified in terms of *display objects* such as icons, cursors, and windows.

These views of input and output are not symmetric. This asymmetry is rooted in the nature of the systems the UAN evolved to describe. Such systems traditionally provide a keyboard, mouse, and display screen for user-computer communication. The physical functions that define the behavior of keys and locators (press and release, move) are very similar to the user actions these devices support (press or release a key, move a pointer); thus, the user actions associated with these systems are most naturally specified in terms of particular input *devices*.

System output, on the other hand, almost always appears through a visual display. (Other output modalities have not been significant in most UAN descriptions, beyond the occasional beep.) The visual display usually presents a rectangular array of pixels, each of which can be set independently to various values (on or off, black or white, or one color from a palette or color space, depending on the system). UAN descriptions generally do not specify visual output at this level of detail. Instead, they refer to visual *objects* that *appear* on the screen. The system designer and the end-user both think in terms of such objects, not in terms of individual pixels on a display. In fact, software implementations usually refer to individual pixels only at the very lowest level of abstraction; at all higher levels, they manipulate software constructs corresponding to visual objects.

The UAN is not *constrained* to this sort of interface, and as it is extended to new interaction styles and devices, this dichotomy between input and output may begin to fade. But the version of the UAN that we consider here does view input and output very differently, and our timetree-based definition of the UAN will reflect this difference.

6.1.1 Input devices

UAN descriptions refer to input devices by name. For example, pressing the button of a single-button mouse is represented by **Mv**, where **M** refers to the mouse button and **v** refers to the action of pressing. Pressing the "X" key on a keyboard is represented by **Xv**. (This scheme must be modified when one wants to denote pressing the "v" key, or worse yet, the "M" key.) Existing UAN descriptions refer to only one pointing device, so ~ is used to represent movement of the (single) pointer, and no explicit name for the device exists.

These input devices have state, and we represent this state in the timetree model with state components. To represent the state of the "X" key, for example, we use a component named *X-key-state*, with the range {*down*, *up*}. All key-like devices (keyboard keys, mouse buttons, etc.) are represented with similarly constructed names. (In choosing the names *down* and *up* we assume the usual orientation of a keyboard; other names, such as *on* and *off*, might be used instead if they would improve overall clarity.)

The pointing device has an associated two-dimensional location, usually indicated by a cursor. We represent this location with a state component named *cursor-location*, with a range representing all possible two-dimensional locations. We allow the elements of this range to take on whatever data type is used in a particular interface specification. For example, a drafting program might use units of fractional inches, while a paint program might use units of pixels. Other applications might call for more abstract representations, which we discuss in the next section.

6.1.2 Display objects

Display objects, like input devices, are referenced by name in the UAN. For example, the UAN representation of highlighting an icon is **icon-name!**. The action of moving the cursor to an icon is represented by **~[icon-name]**; in this example, the square brackets indicate the *context* of the icon. Informally, the context of an object is the area on the screen associated with an object for purposes of manipulation — one "points to" an object by moving the cursor to the object's context. On the Macintosh, the context of a display object can be (for example) the object itself, or a rectangle circumscribed around the object, or a set of small handles at the corners of the object.

In UAN descriptions of "point-and-click" interfaces, these contexts may be the only kind of cursor location ever mentioned. In such descriptions, it is sometimes convenient to group the values of *cursor-location* into equivalence classes corresponding to the contexts of visible items (Section 5.4.5). This yields a more abstract representation of cursor location, hiding irrelevant detail about precise cursor location. While the user still moves the cursor smoothly across the screen, the only movements of significance to the design are movements into or out of visible items.

UAN descriptions do not specify the exact nature of an object's appearance, the aspects of appearance changed by highlighting, or the area constituting the object's context. Instead, the UAN uses abstract references, leaving visual specifications to other representation techniques (such as storyboards, screen sketches, or textual descriptions).

The UAN provides notations for displaying an object at a particular location or erasing an object. These operations involve two more characteristics of objects: whether they are visible at all, and where they appear.

We represent attributes of display objects at a level of abstraction compatible with that of the UAN. The timetree model could in principle accomodate precise visual appearance as a component of state; an *icon-image* component, for example, could have as its range the set of possible icon images, or arrays representing bitmaps. For now, though, we leave these representations abstract.

The state components associated with a display object called *item* are called *item-form*, *item-context*, *item-location*, *item-visible*, and *item-highlighted.* The first two components, *item-form* and *item-context*, represent the abstract form and context of *item*. Component *item-location* represents the location of *item* on the screen, and is a location like *cursor-location*; in particular, it might be a pair of coordinates, or it might be an abstract position. Components *item-visible* and *item-highlighted* have boolean values, indicating respectively whether *item* is visible and whether it is highlighted. Items with multiple highlighting styles, indicated in the UAN by ! and !!, have separate components for each highlighting style, named (for example) *item-highlighted* and *item-highlighted2.*

The chart of Figure 6.1 summarizes these entities and their representations.

Entity	State component name	State component range
Input devices:		
key or button X	*X-key-state*	{ *down*, *up* }
pointing device	*cursor-location*	two-dimensional or abstract locations
Display object *item*:		
physical appearance	*item-form*	abstract
context	*item-context*	abstract
location	*item-location*	two-dimensional or abstract locations
visibility	*item-visible*	{ *true*, *false* }
highlight status	*item-highlighted*	{ *true*, *false* }

Figure 6.1. UAN entities and state representations.

6.2 User actions

UAN descriptions at low levels of abstraction refer to physical user actions applied to physical devices. Since the commonly used input devices are keys or buttons and pointing devices, these low-level actions involve pressing and releasing keys or moving the cursor. The results of key presses are very simple, but our representation of cursor movement must be slightly more involved to account for different location representations (Section 6.1.2).

6.2.1 Keys and buttons

A key affords two actions, *pressing* and *releasing*. Pressing the "X" key (for example) changes its associated state component, *X-key-state*, from *up* to *down*. Releasing it changes the state from *down* back to *up*. We represent each of these user actions by an activity arc, labeled with a single activity, and with start and end timestates containing the proper associated state values. We call the activities associated with pressing and releasing the "X" key **Xv** and **X^**, for convenience and consistency with the UAN.

The UAN also provides shorthand notations for entering a string or typing a value for a string variable. For example, **K"abc"** describes the user action of typing the literal string

abc, and **K(user-id)** describes the user action of typing a value for a string variable named **user-id**. The first example describes a simple concatenation of individual key presses and releases (ignoring issues such as key rollover), and can be represented by a linear sequence of activities, or by a single compound activity at a higher level of abstraction. The second example is complicated by the addition of a "string variable" presumably associated with interface or system state. This activity could be represented by a sequence of key presses and releases and associated incremental changes to the string variable, but it is generally more useful to represent it by a single, abstract, compound activity. The end timestate of such an activity associates the entered value with the state component corresponding to the string variable. (We specifically address state components associated with interface state and computation in Section 6.4.)

The actions we have considered so far, and most actions that we consider in subsequent sections, change the values of one or more state components. In situations where it is necessary to reason about state values across several actions, we require that every such change of value be specified explicitly in the activity definition. This allows us to assume that any state components not explicitly mentioned in the definition of an activity are not changed by that activity.

6.2.2 Cursor movement

The UAN represents user actions on pointing devices by their effect on cursor position. It may sometimes be important to consider these actions at a lower level of detail; for example, to predict whether a mouse, a joystick, a trackball, or a touch screen is more suitable for a particular application, it may be necessary to discuss the different physical actions associated with each device. The UAN, though, abstracts away this level of detail, and our model-based definition of the UAN does the same.

Cursor movement is most commonly specified in the UAN in terms of a destination. The user may move the cursor to an arbitrary location outside any visible object (~**[x,y]**), into the context of a visible object **X** (~**[X]**), or to an arbitrary location within a visible object (~**[x,y in X]**). The cursor is understood to remain within a context until it is explicitly moved elsewhere. Movement *out of* the context of object **X** is indicated by **[X]**~, the one movement specifier that does not specify a destination. This movement specifier can be

understood to leave the cursor at some arbitrary position outside object **X**, like the specifier **~[x,y]**. In practice, movement out of contexts is frequently omitted in UAN specifications; moving out of one context (say **Y**) and into another (**X**) is frequently represented by **~X**, instead of the explicit **Y~ ~X**. Since it is sometimes necessary to consider context-leaving operations explicitly, we model them explicitly. We discuss this issue further in Chapter 9.

Like the actions of Section 6.2.1, we represent each of these movement specifiers by an activity arc labeled with an activity whose name is derived from the corresponding UAN expression. The differences between these activities are reflected by the value of *cursor-location* at their start and end timestates. The chart in Figure 6.2 summarizes these state values.

Action	***cursor-location* value (start timestate)**	***cursor-location* value (end timestate)**
~[x,y]	any location	[*x*,*y*] (or any location)
~[X]	any location outside **[X]**	any location inside **[X]**
~[x,y in X]	any location inside **[X]**	[*x*,*y*] inside **[X]** (or any location inside **[X]**)
[X]~	any location inside **[X]**	any location outside **[X]**

Figure 6.2. Cursor movement activities and their associated state information.

While the notation **x,y** suggests movement to a particular point on the screen, it actually is almost never used in that sense. Instead, **x,y** indicates any arbitrary location satisfying the actual constraints of the particular idiom — either an arbitrary location outside any object context or an arbitrary location within an object context. Movement to a particular point is indicated by **~[x',y']** instead of **~[x,y]**. In general, for interfaces based on direct manipulation of windows, icons, and menus, only movement into or out of contexts is significant; explicit references to movement within a context or outside all contexts are limited to situations in which feedback depends on such motion. We address this situation in the next section.

6.3 System feedback actions

The UAN provides notation for several common types of system feedback. It includes specifiers for displaying and erasing objects, highlighting or unhighlighting objects, and moving objects in response to user actions. We represent each feedback action by an activity arc labeled with the appropriate system activity. As with user actions, we derive names for these actions from the UAN notation. We represent the results of these activities in terms of changes to the state components associated with a display object. Figure 6.3 summarizes simple feedback actions and their effects on state.

Action	State component values associated with X (start timestate)	State component values associated with X (end timestate)
X!	*X-visible = true* *X-highlighted = false*	*X-visible = true* *X-highlighted = true*
X-!	*X-visible = true* *X-highlighted = true*	*X-visible = true* *X-highlighted = false*
X!!	*X-visible = true* *X-highlighted2 = false*	*X-visible = true* *X-highlighted2 = true*
display(X)	*X-visible = false*	*X-visible = true* *X-highlighted = false*
erase(X)	*X-visible = true*	*X-visible = false*

Figure 6.3. System feedback activities and their associated state information.

As with cursor movement, common interpretations of the UAN highlighting notation often omit some of these requirements or leave some information implicit. For example, the specification **X!** frequently is meant to leave **X** highlighted *regardless* of its previous condition. In other words, **X!** is often interpreted as "if **X** is not highlighted, highlight it; if it is already highlighted, leave it unchanged." The definition we adopt here still allows specification of this behavior in terms of its explicit expansion.

The **display(X)** specifier can accept modifiers to specify a location. The form **display(X) @ x,y** displays item **X** at the specified location, and we represent this by specifying *X-location = x,y* in the activity's end timestate. The form **display(X) @ Y** displays item X at item Y, and we represent this by *X-location = Y-location* in the end timestate. There

is no precise definition of "at" in this usage; we assume that the abstract specification of an icon's appearance also includes some reference point for positioning.

A related idiom, **outline(X)**, specifies an "outline" of item **X**. This outline is a display object in its own right; its *form* is related to *X-form*, but since these forms are usually defined outside the UAN, we make no formal statement of the relationship. The Delete File task of Section 2.2 demonstrates the **outline(X)** idiom.

There are some forms of feedback that inherently involve compound or joint activity. For example, the UAN idiom **!-!** specifies *blinking* the highlight of an object. This can be represented by a sequence of three system activities: highlight the object, wait for an interval, unhighlight the object.

The idiom **X > ~**, defined as "item **X** follows the cursor," is more complicated. It is associated with a user action involving cursor movement, like **~[x,y]**. But while most other feedback activities occur *after* their corresponding user actions, **X > ~** occurs *during* the corresponding user action. This behavior is generally *implemented* as a tight loop in which a machine samples the cursor position and updates the display object's position frequently enough to preserve the illusion of smooth movement. But the designer should not need to consider this level of detail; all that matters is the user's perception that an object follows the cursor's movement, *concurrent with* that movement.

We usually represent this situation with a joint activity arc, labeled with the user action and the system feedback, and leave the temporal relation between the two activities implicit, as the current UAN does. However, it is possible to make some statements about the temporal ordering of the activities; obviously, since the user action *causes* the system response, onset of the system response must *follow* onset of the user action. In the most common interpretation of this UAN idiom, termination of the system response also follows termination of the user action. We can represent this level of detail using the representation of concurrency from Section 5.3.4. We break **~[x,y]** and **X > ~** into initiate, continue, and terminate stages, and order them as described above. This yields the timetree of Figure 6.4.

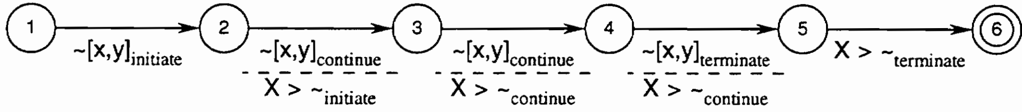


Figure 6.4. Detailed timetree for "cursor-following" behavior.

We can assign values to *cursor-location* and *X-location* at each timestate to reflect the changes caused by each activity. We assume that the cursor is originally at some location $[x,y]$ and X is at location $[a,b]$, and that the cursor moves over a distance $[dx_1,dy_1]$ in the second activity and $[dx_2,dy_2]$ in the third, leaving the cursor at $[x',y']$. (Since *initiate* and *terminate* are abstract activities marking the beginning and end of actual activities, we choose to view them as having no duration, and so no actual change in cursor or icon location results from them. We adress this issue in more detail in Section 6.6.8.) Figure 6.3 shows the resulting values for the two state components at each timestate.

Timestate	***cursor-location***	***X-location***
1	$[x,y]$	$[a,b]$
2	$[x,y]$	$[a,b]$
3	$[x+dx_1,y+dy_1]$	$[a,b]$
4	$[x+dx_2,y+dy_2]$	$[a+dx_1,b+dy_1]$
5	$[x+dx_2,y+dy_2] = [x',y']$	$[a+dx_2,b+dy_2]$
6	$[x',y']$	$[a+dx_2,b+dy_2]$

Figure 6.5. Detailed state information for "cursor-following" behavior.

Actually, the third activity, labeled by { **~[x,y]**$_{\textbf{continue}}$, **X>~**$_{\textbf{continue}}$ }, may be repeated any number of times as the user drags the cursor (and **X**) through various points on the screen. We could represent this by the same sort of branching structure used to represent *-closure (Section 5.3.6). But repeating this activity simply moves the cursor and **X** to other arbitrary intermediate locations, with no effect on the end result of the activity sequence, so we lose no information by abstracting these branches away. This reflects common UAN usage, which views **~[x,y]*** and **~[x,y]** as equivalent.

The UAN also provides a related idiom, **X** >> ~, to indicate that **X** is "rubber-banded" as it follows the cursor. This usually indicates that the object "stretches" between a fixed point and the cursor. Examples include drawing lines or ovals in MacDraw, resizing windows in Motif, and dragging out frames in Word for Windows. The temporal behavior of this idiom can be described as above, but the details of **X**'s changing appearance must generally be specified outside the UAN.

6.4 Interface state and connection to computation

The preceding sections have shown how timetree state components can represent the state of visible objects and input devices that are part of an interface. Most of these state components represent aspects of interface state that are directly perceptible to the user — key position, cursor location, or appearance of a visible object.

There are some aspects of state that do not correspond to any single perceptible condition. For example, the Delete File task of Section 2.2 refers to a "selected" file. This task specification does provide a visual indication of the selected file: its *icon* is highlighted. But the file *itself* is not visible to the user; rather, it is an abstract object maintained by the operating system, and the interface must translate between the user's manipulations of the file icon and operating system manipulations of the file. In addition, it is convenient to describe the behavior of the interface in terms of an abstract variable, *selected,* that takes on values indicating individual files or icons. While this variable is closely related to what the user sees, it is itself abstract and not perceptible.

The UAN provides a separate *interface state* column for the designer to specify abstract aspects of interface state. Our experience with the UAN has shown that interface designers typically create interface state variables to represent abstract interface state associated with operations like selection, item locking, or item lists. The designers of the UAN have not imposed a formal structure on interface state entries, and so the format of these entries varies to suit designer preference and convenience. Since this is the case, we do not provide a formal structure into which *all* UAN interface state specifications must fit.

But while the format of entries in the interface state column is not rigidly defined, the details these entries convey can be important to design specification. For example, many conditions of viability (Sections 3.6 and 6.5) depend on abstract interface states, and these conditions in turn control the availability of tasks and functions. Any useful formal model of interactive behavior must deal with these state components and conditions.

To represent information from the UAN's interface state column in the timetree model, we impose the following requirement:

> If an entry in the interface state column is to be modeled (for example, if it is used in a condition of viability), all variables, values, and operations in the entry must be representable in terms of state components, state component values, and activities.

Preceding sections of this chapter have shown how other components of interface state can be represented in terms of state components; abstract state components specified in the interface state column of a UAN description can be represented analogously. While it is impossible for us to foresee every kind of state information that designers may specify, our experience with UAN descriptions has not yielded interface state column entries that violate this requirement.

We adopt a similar approach for the UAN's *connection to computation* column. This column provides a means for the interface designer to refer to system functions or data structures that are not part of the interface. For example, the Finder is fundamentally a tool for manipulating and visualizing a file structure, but files themselves are not a part of the interface — the user manipulates icons in windows, which *represent* files and directories. A specification for a Finder task, like the example of Section 2.2, describes these manipulations, but must also specify the system actions that are in some sense the goal of the task.

The content of the connection to computation column is even more system- and domain-dependent than that of the interface state column. In the Delete File specification from Section 2.2, the comment "mark file for deletion" is an abstract, high-level description of the file system change that actually results from "dragging a file to the trash." In other

UAN descriptions, entries in this column can contain other high-level descriptions, procedure names, function calls, or lines of code. Such entries may also refer to variables within the code of the non-interface part of the system.

Since the connection to computation column can contain arbitrarily high- or low-level references to system activity, we again do not impose a formal structure on these references. Like entries in the interface state column, though, these entries can be an important part of an interface specification. For this reason, we adopt the same approach we use for interface state: if connections to computation are to be modeled with timetrees, all the variables and values they mention must be representable as state components and state component values. We can represent procedure calls, high-level system functions, and other such computational actions as single system activities. If it is important to view these activities at lower levels of abstraction, we must turn to design and implementation languages appropriate for non-interactive systems; representations in this domain are outside the purview of the timetree model.

6.5 Conditions of viability

The availability of a particular task frequently depends on some aspect of interface or system state. The notion of system or interface *modes* describes situations in which the same user action can have different effects when a system is in different states. In an older interface, for instance a command line interface to an operating system, an attempt to perform an unavailable task might generate an error message. In more contemporary direct-manipulation or menu-driven interfaces, it is considered good practice to *disable* commands that invoke unavailable tasks — for example, by graying out a button or a menu item. Thus, the **Empty Trash** menu item on the Macintosh is grayed out when the Trash is empty, indicating that the task of emptying the Trash is not available when there are no items in the Trash.

The UAN provides *conditions of viability* to represent these dependencies. Conditions of viability are usually propositions — expressions, stated in terms of interface state components, that yield boolean values. The user can begin to perform a task or action only when its condition of viability is *satisfied*, that is, when evaluation of the expression

yields *true*. We presented an example of this in Section 3.6, where file manipulation tasks are only available if a file's icon is visible.

Formally, we state that the action or task associated with a condition of viability can take place only if the condition is *true* at the time the action or task is to take place. If the condition is *false*, the action or task in question cannot take place as part of the task in which its precondition is specified.

These explicit conditions of viability can appear in the heading of a task description, in which case they control the viability of the entire task, or in front of an action or subtask, in which case they control the viability of that single activity. In addition, some activities have *implicit* conditions of viability — conditions that must be true for the action to take place, even though the design does not specify them explicitly. For example, according to the usual meaning of context, it is not possible to move the cursor to the context of an item that is not visible. More trivially, it is not possible to press a key that is already down, or to release one that is already up.

It is important to understand that these conditions of viability represent assertions about the behavior of an interactive system, and that they do *not* necessarily represent activities on the part of the user or the system. For example, neither the system nor the user "checks" that a key is down before it is released. The key simply "works that way." Similarly, the UAN statement **icon visible: ~[icon]** does not imply that the system or the user must check the icon's visibility. Rather, it simply states that if the icon is not visible, the action **~[icon]** cannot and will not take place.

We have already indicated that we can use timetree state components to represent aspects of interface and system state. To reflect viability conditions in timetrees representing tasks, we make assertions about state values at the beginning of a task or activity, that is, in the task's initial timestate. We then compare these state values to the state values at the end of the preceding task or activity to determine whether the new task can execute. This can be done in the timetree composition operators if we modify them to support *conditional composition* based on state information.

In some situations, the result of a composition depends on a state component whose value is not known. In fact, viability conditions are generally used in UAN task descriptions to represent situations where the state conditions determining viability may vary — otherwise, the designer would simply describe the actions that happen under the (known) state conditions, and perhaps at most include a comment about those conditions. In principle, a timetree representing all possible sequences of activity through an *entire* system design would contain the state information necessary to evaluate viability conditions — but only if such a timetree were constructed from the initial timestate forward, since the viability of an activity depends on the state at the end of the *preceding* activity. Further, in practice, a timetree of this size and detail is prohibitively large for non-trivial systems.

It would be possible to represent a set of timetrees, one for each set of state conditions, and pick the proper one for composition when the pre-existing state becomes known. But all the algorithms and operators we have presented so far deal with single timetrees (except, of course, those that compare or combine two timetrees), and changing the logic of these algorithms to deal with sets of timetrees — "timeforests" — would introduce a great deal of additional complexity.

We choose instead to represent this situation of uncertain state values by introducing a new kind of activity arc, the *virtual activity*. The virtual activity starts at a timestate at which a certain state component is undefined, and ends at a timestate at which that state component has a value assignment. Thus, each condition of viability is represented by a virtual activity arc that leads to a timetree branch in which that condition is met. In this way, we preserve the separation of timetrees that have different requirements for initial state, but still provide a way to manipulate them as a single timetree. We discuss virtual activities in more detail in Section 6.5.2.

Composition operations on timetrees handle these virtual activities just like normal activity arcs representing user, system, and environmental activity. But it is important to remember that they, like viability conditions in the UAN, do not represent actual activity on the part of any party. It is easy to interpret them as "the system checking a state value." Indeed, in some system designs that may happen — specifically, when a system action depends on some state value — but, if so, this check must itself be represented by

a system activity, and determining which viability conditions are checked in this way is a task for the system designer. In fact, they can sometimes represent "the *user* checking a state value." We discuss these issues further in Chapters 8 and 9.

6.5.1 Conditional composition

As we presented composition operators for timetrees (Section 5.3), we alluded to the situation in which two timestates that are to be combined into one have incompatible state component value assignments. Timestates are combined in every composition operation. In choice composition, the initial timestates of two timetrees are replaced by a single new timestate. In sequence composition, the terminal timestates of the first timetree are replaced with initial timestates of copies of the second timetree; these new timestates represent both the old terminal timestate of the first timetree and the old initial timestate of the second timetree. Closure composition follows a similar pattern. Finally, in interleaved and concurrent composition, each timestate in the product timetree represents a pair of timestates in the two component timetrees.

In composition operations that build a new timetree from past to future, a start timestate of one activity is combined with an end timestate of a preceding activity. In these situations, it is possible to compare the state configurations of the end timestate and the start timestate, and combine them only if there is no state conflict. If there is a state conflict, we eliminate the timetree branch that we would otherwise construct from the new activity. Figure 6.6 illustrates an instance of this process.

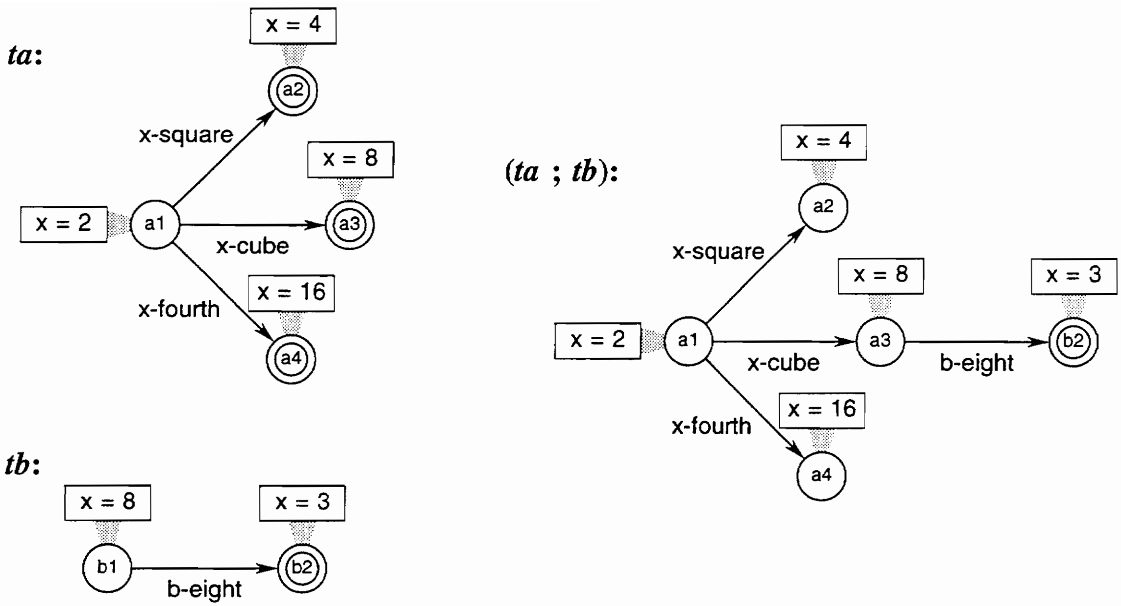


Figure 6.6. Conditional composition of timetrees with state conflicts.

As Figure 6.6 indicates, conditional composition can yield a timetree with leaf nodes that are not terminal timestates. When *ta* and *tb* are composed sequentially to form (*ta* ; *tb*), two of the terminal timestates of *ta* (*a2* and *a4*) have state conflicts with the initial timestate of *tb*. As a result, the activity of *tb* cannot follow the activities (*x-square* and *x-fourth*) that led to the conflicting timestates in *ta*. But *a2* and *a4* are *not* terminal timestates in (*ta* ; *tb*) — they represent points at which activities of *ta* have been completed, but not the activity of *tb*, and so cannot represent a completion of (*ta* ; *tb*).

The definition and interpretation of timestates presented in Section 5.2.2 requires that every leaf node be a terminal timestate, except in *d*-restricted timetrees. Less formally, we said that each terminal timestate "describes a possible completion of the activity or activities modeled by the timetree." Since *a2* and *a4* do not represent completions of (*ta* ; *tb*), they should not be terminal timestates. But since *x-square* and *x-fourth* cannot lead to terminal timestates, they cannot be part of the activity (*ta* ; *tb*) describes. Thus, we can prune them off without affecting the behavior of (*ta* ; *tb*). This yields the timetree of Figure 6.7.

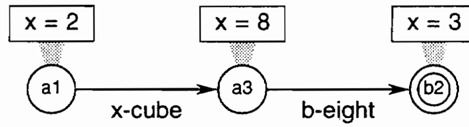


Figure 6.7. (*ta* ; *tb*) after pruning non-terminal leaf timestates.

Since we require that timetrees in general have no non-terminal leaf nodes, we provide an algorithm to prune back branches that end on non-terminal timestates. We use a variation of *prune-path* (Section 5.4.1), applying it to each nonterminal timestate that is a leaf node.

for each *leafstate* $\in$ *tt.TS* : *leafstate* $\notin$ $tt.TS_{term} \wedge \nexists$ <*leafstate*, ts_{end}, A_{label}> $\in$ *tt.A*
prune-nonterm-path(*tt*, *leafstate*);

prune-nonterm-path(*tt*, *state*):

-- *back up along the path that leads to* state *until we reach an earlier terminal timestate along the path or a timestate with other branches.*
while *state* $\notin$ $tt.TS_{term} \wedge \nexists$ <*state*, ts_{end}, A_{label}> $\in$ *tt.A*

-- *remove the state information associated with this timestate:*
tt.SCM := *tt.SCM* - { <*state* , sc_{name}, sc_{value}> $\in$ *tt.SCM* };

-- *remove this timestate from* TS:
tt.TS := *tt.TS* - { *state* };

-- *remove the arc leading to this timestate*:
tt.A := *tt.A* - { <*startstate*, *state*, A_{label}> };

-- *repeat at the preceding timestate*:
state := *startstate*;

We present modified conditional composition operators for timetrees in Section 6.6, in conjunction with the UAN composition operators they model.

6.5.2 Virtual activities

There are many situations in which the viability of an activity depends on some state component whose value is not known. For example, many task descriptions in the UAN use viability conditions not only to describe conditions that must hold before the task can

be performed, but also to indicate activities within the task that only take place under certain circumstances. When such a task is composed with others, more state information may become available, but we want a way to represent the task *without* specifying that information.

We introduce the virtual activity as a way to maintain different timetree branches with different initial state component values, while allowing those branches to be part of one timetree with one initial timestate. Virtual activities start at that initial timestate, at which the state component of interest is undefined, and end at the initial timestate of each branch, each of which defines its own value for the state component. Figure 6.8 presents a UAN description with viability conditions and a timetree that models it using virtual activities.

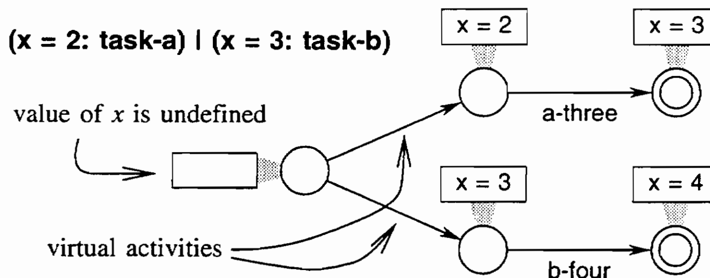


Figure 6.8. Timetree with virtual activities.

We represent virtual activities with activity arcs, like actual activities, but we need some method for distinguishing them from actual activities. We can extend *Aclass* (Section 5.2.1) to include a *virtual* class as well as *user*, *system*, and *environment.* But distinguishing virtual activities is usually quite simple, because we generally give them *empty* labels, signifying that they represent no actual activity. For clarity, we sometimes label them in illustrations with names corresponding to the preconditions they help represent, but the algorithms we describe here do not need recourse to those labels.

Since we leave virtual activities unlabeled by convention, the *disambiguate* algorithm of Section 5.3.5 does not handle them very gracefully. Recall that *disambiguate* condenses equivalent activity arcs, that is, activity arcs with the same label originating at the same timestate. The algorithm also disregards the possibility of state conflicts among the end

timestates of these arcs. In contrast to actual activities, virtual activities starting at a common timestate are distinguished only by the state configurations of their end timestates.

We can modify *disambiguate* to handle virtual activities analogously to actual activities. Where the previous version of the algorithm condensed activities based on their labels, this new version also condenses virtual activities based on the state configurations of their end timestates. We introduce a variable *Stateset* to keep track of end timestates of virtual activity arcs encountered so far at the current level, analogous to the variable *Arcset* for keeping track of actual arcs. For actual activities, the logic is unchanged; for virtual activities, the logic is again analogous, substituting a check of the end timestate's state configuration for a check of the arc's label.

disambiguate(tt, state):

> *-- initialize a set of arcs leaving this timestate:*
> *Arcset* := ∅;
>
> *-- initialize a set of end timestates of virtual activity arcs leaving this timestate.*
> *Stateset* := ∅;
>
> *-- for each arc leaving the current timestate, see if it is equivalent to an arc already encountered. If it is, condense it with the previous arc; if it is not, add it to* Arcset, *or if it is a virtual activity arc, add its end timestate to* Stateset.
> **for each** *tarc*: *<state, endstate, Label>* ∈ *tt.A*
>   **if** *Label* = ∅ **then**
>   *-- this is a virtual activity; condense based on state configuration of* endstate
>   **if** ∃ *tstate* ∈ *Stateset*: *SCvalues(tt,endstate)* = *SCvalues(tt,tstate)*
>   **then** *replace(tt, endstate, tstate)*
>   **else** *Stateset* := *Stateset* ∪ { *endstate* }
>   *-- this is an actual activity; condense based on arc label*
>   **else if** ∃ *arc* ∈ *Arcset*: $arc.A_{label}$ = *Label*
>   **then** *replace(tt, endstate,* $arc.ts_{end}$*)*
>   **else** *Arcset* := *Arcset* ∪ { *tarc* };
>
> *-- recurse on the end timestate of each arc in* Arcset.
> **for each** *arc* ∈ *Arcset*
>   *disambiguate(tt,* $arc.ts_{end}$*)*;

We also modify the *replace* algorithm to check for state conflicts between its arguments. Such a condition will arise only when two arcs start from the same timestate, bear the same nonempty set of labeling activities, and yet yield different values for some state component at the terminal timestate. Since this presumably reflects an error in specification, we fail if it occurs. We discuss this situation further in Section 8.4.3.

replace(*tt*, *losestate*, *keepstate*):

-- *remove the redundant arc leading to* losestate.
tt.A := *tt.A* - { <ts_{start}, *losestate*, A_{label}> };

-- *replace each arc leaving* losestate *with an identically labeled arc leaving* keepstate.
for each *tarc*: <*losestate*, *deststate*, *Label*> ∈ *tt.A*
tt.A := *tt.A* - { *tarc* };
tt.A := *tt.A* ∪ { <*keepstate*, *deststate*, *Label*> };

-- *If there is a state conflict between* losestate *and* keepstate, *fail the composition.*
if *state-clash*(*losestate*, *tt*, *keepstate*, *tt*) **then fail;**

-- *copy* losestate*'s state component value assignments to* keepstate. *If there is a state conflict, fail the composition.*
for each <*losestate*, sc_{name}, sc_{value}> ∈ *tt.SCM*
tt.SCM := *tt.SCM* - { <*losestate*, sc_{name}, sc_{value}> };
tt.SCM := *tt.SCM* ∪ { <*keepstate*, sc_{name}, sc_{value}> };

-- *if* losestate *was a terminal timestate, remove it from* TS_{term}. *If so, and* keepstate *was not already a terminal timestate, make it one.*
if *losestate* ∈ *tt.*TS_{term} **then**
tt.TS_{term} := *tt.*TS_{term} - { *losestate* };
if *keepstate* ∉ *tt.*TS_{term}
then *tt.*TS_{term} := *tt.*TS_{term} ∪ { *keepstate* };

-- *remove* losestate *from* TS.
tt.TS := *tt.TS* - { *losestate* };

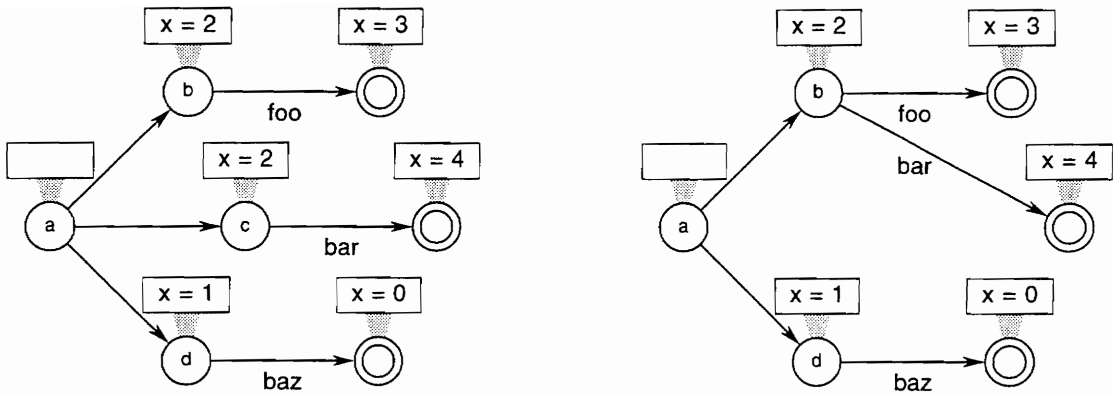


Figure 6.9. Timetree with virtual activities before and after disambiguation.

Figure 6.9 illustrates the disambiguation of a timetree with virtual activities. The timetree on the left is ambiguous according to our new criterion; it has two virtual activities starting at the same timestate (a) and leading to timestates with identical state configurations (b and c). After disambiguation, timestate c is collapsed onto timestate b — in other words, it is replaced by timestate b. Activity *bar*, which originated at c, now originates at b.

If a timetree containing virtual activities is composed with other timetrees, additional knowledge about state component values may make it possible to determine which branches are available and which are not. In such cases, it is possible to collapse out virtual activities by either pruning off the subtrees that descend from them (if there is a state conflict) or combining their start and end timestates.

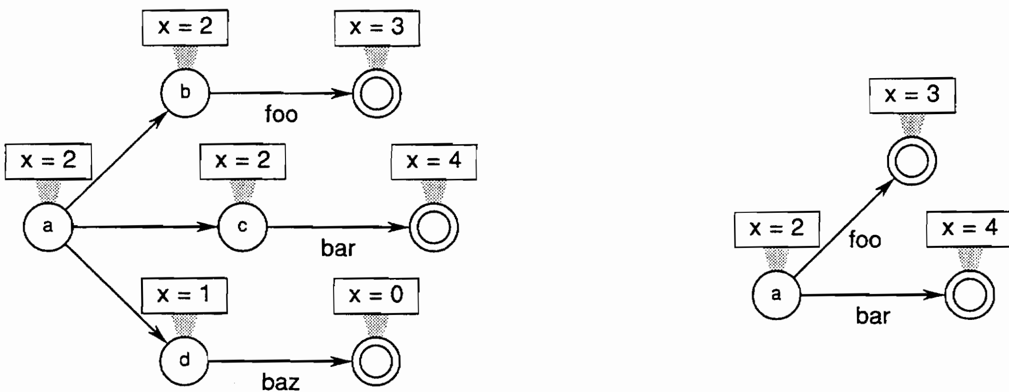


Figure 6.10. Timetree before and after collapsing virtual activities.

Figure 6.10 illustrates this process. Note that the timetree at the left, unlike the timetrees of Figure 6.9, defines a value for *x* at the initial timestate *a*. As a result of this definition, timestates *a*, *b*, and *c* have identical state configurations, and timestate *d* has a state conflict with *a*. This allows us to collapse *b* and *c* onto *a*, and to prune off the subtree delimited by *d*. These manipulations yield the timetree at the right.

We define an algorithm, *collapse-VAs*, to perform these operations. The algorithm is based on *disambiguate*.

collapse-VAs(*tt*, *state*):

> -- *for each virtual activity arc leaving the current timestate, check for state conflicts between its start and end timestates. If there is a conflict, prune off* endstate, *the subtree rooted at it, and the arc leading to it. If there isn't a conflict, recurse on* endstate, *then see if there are any state components whose values are defined in* endstate *but not in* state. *If there are not, remove the arc and combine* state *and* endstate; *if there are, leave the arc unchanged.*
>
> **for each** *tarc*: <*state*, *endstate*, ∅> ∈ *tt.A*
>
>   **if** *state-clash(tt, state, tt, endstate)*
>
>   **then** *prune-state(tt, endstate)*
>
>   **else**
>
>   *collapse-VAs*(*tt*, *endstate*);
>
>   **if** ∄ *name*: (∃ <*endstate*, *name*, sc_{value}> ∈ *tt.A* ∧
>
>   ∄ <*state*, *name*, sc_{value}> ∈ *tt.A*)
>
>   **then** *replace*(*tt*, *endstate*, *state*);

Again, the *replace* algorithm is the same one presented earlier in this section.

It is possible to define an infinite timetree which this algorithm will attempt to reduce to an infinitely branching timetree of finite depth, which is not allowed under the conditions presented in Section 5.1. It does not appear at this time that such timetrees can be constructed using the operators we have defined. We address this situation in more detail in Chapter 9.

6.5.3 State propagation

In the preceding sections we have used state components and state component values to represent input devices, display objects, results of user and system activity, and interface and computational state. In the current section, we have shown how this representation allows us to incorporate viability conditions into the timetree model. But we have not yet dealt explicitly with the persistence of state information, and state persistence is critical to the techniques we have described.

Consider the UAN sequence **X!** ; **Y!**. According to the definitions of Sections 6.3 and 5.3.2, we can illustrate the timetrees associated with these two actions and their sequential composition as follows.

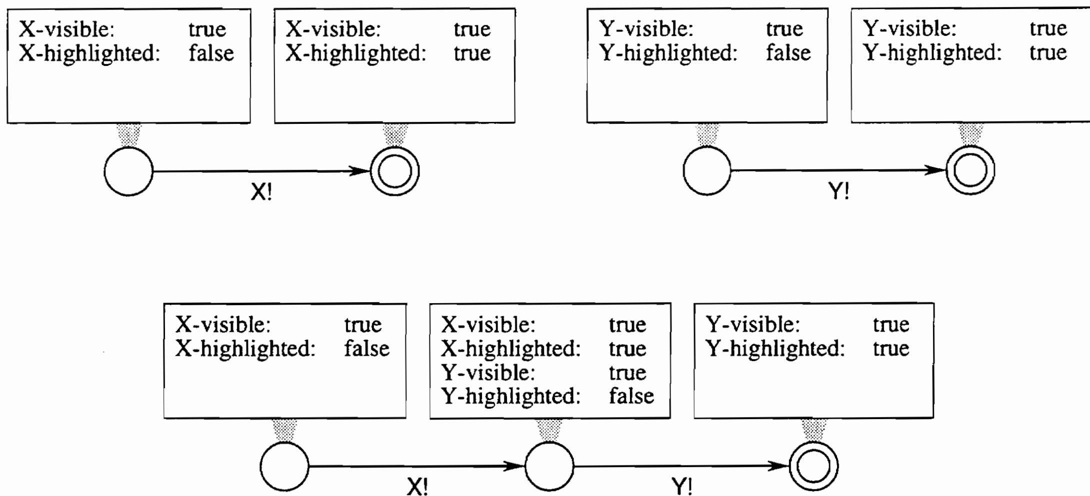


Figure 6.11. Sequential composition with disjoint state components.

The definition of **X!** involves only the state components *X-visible* and *X-highlighted*, and likewise **Y!** involves only *Y-visible* and *Y-highlighted*. Since the two do not mention any common state components, composition of these two activities cannot yield a state conflict. Instead, the timestate that joins the two activities defines values for all four state components.

At the initial and terminal timestates, though, some state components are undefined. While this is perfectly legal under the timetree model, it does not fit with the interpretation of state we have described in this chapter. Specifically, we "assume that

any state components not explicitly mentioned in the definition of an activity are not changed by that activity" (Section 6.2.1). In other words, according to our interpretation of state in our definition of the UAN, state values must persist until they are changed explicitly.

To ensure that state components retain their last value until something explicitly changes that value, we can use the *propagate-forward* operator of Section 5.4.3. The rules specified in Section 6.2.1 require that every value changed by an activity be specified in the definition of that activity, and so any activity that changes a state value must have a value assignment for that state component *in its end timestate*. An activity that leaves a state value unchanged will not mention that state value in its end timestate. The *propagate-forward* operator propagates a state value only into timestates which do *not* already have value assignments for that state component. Upon reaching a timestate which *does* define a new value for the state component, *propagate-forward* begins propagating that new value instead. This provides the persistence we require — once a state value is changed, it retains that new value until some activity explicitly changes it again.

This approach still leaves one source of ambiguity. The composed timetree in Figure 6.11 labels the end timestate of **X!** with state assignments for *Y-visible* and *Y-highlighted*. While this does not explicitly violate the rules we have stated, it makes it appear that **X!** has assigned a value to *Y-visible* and *Y-highlighted*, and that **Y!** requires certain values for *X-visible* and *X-highlighted.* We would prefer to keep the information about viability conditions for **Y!**, indicated by state component values at its start timestate, separate from the information about state effects contained in **X!**'s end timestate.

We can maintain this separation by interposing a virtual activity between **X!** and **Y!** when we compose them. This allows us to retain an explicit representation of **X!**'s effects on state with no confounding from **Y!**'s viability conditions. Figure 6.12 shows the timetree that results from this interposition.

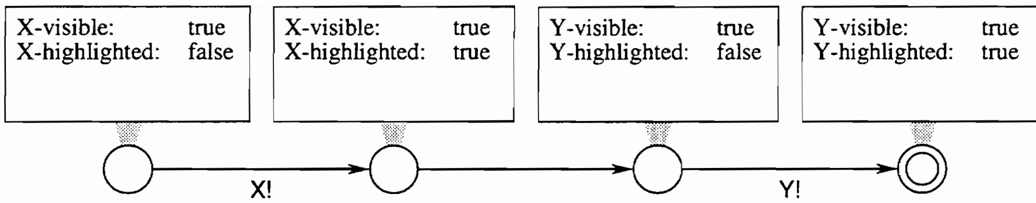


Figure 6.12. Timetree for **X! ; Y!** with interposed virtual activity.

Ordinarily, the viability conditions for these activities would be satisfied as a result of some earlier activity. We can illustrate this situation by composing **X! ; Y!** with another activity that serves only to establish the state values required by the viability conditions of **X!** and **Y!**. We call this activity **set-preconds**. Figure 6.13 shows the timetree for **set-preconds ; (X! ; Y!)**, Figure 6.14 shows the result after propagating states forward in this timetree, and Figure 6.15 shows the result of collapsing virtual activities after state propagation.

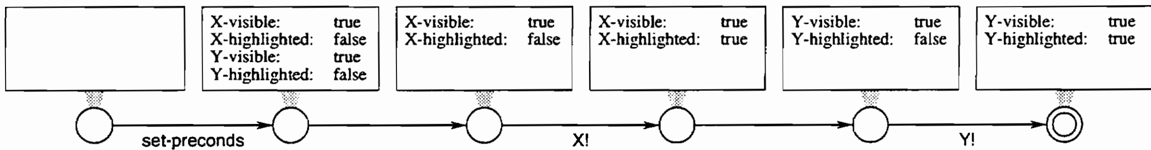


Figure 6.13. Timetree for **set-preconds ; (X! ; Y!)**.

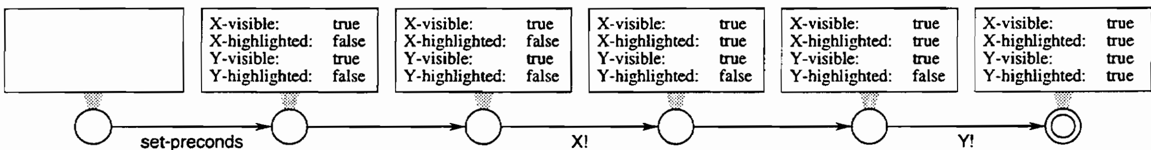


Figure 6.14. Timetree for **set-preconds ; (X! ; Y!)** after state propagation.

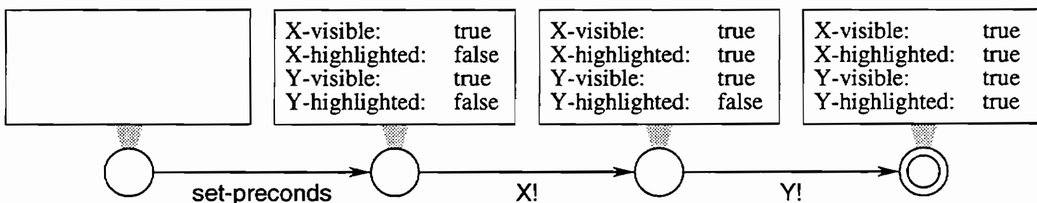


Figure 6.15. Timetree for **set-preconds ; (X! ; Y!)** after propagating state values and collapsing virtual activities.

So far, we have discussed the representation of individual UAN entities and activities, and some of the issues involved in composing activities that include conditions of viability and effects on state. We have shown how to construct timetrees and state assignments to represent individual elements of the UAN. In the next section, we use composition operations on these timetrees to model UAN composition operators.

6.6 Composition

The UAN provides a variety of composition operators to combine individual actions or tasks into larger task descriptions. Since we can represent individual activities in terms of timetrees, we represent UAN composition operators in terms of timetree composition operators.

As we have stated, the composition operators as presented in Section 5.3 do not take state conflicts into consideration. But virtual activities, as introduced in Sections 6.5.2 and 6.5.3, allow us to keep conflicting states separate during composition. Once composition is complete, a distinct reduction operation using *propagate-forward*, *collapse-VAs*, *disambiguate*, and *prune-nonterm-path* allows us to reduce the resulting timetrees to a consistent, unambiguous, and non-redundant form.

reduce(*tt*):

> **for each** name: $\langle \text{state, name, } sc_{value} \rangle \in$ tt.SCM
>   *propagate-forward*(tt, $tt.ts_{init}$, *name*);
> *collapse-VAs*(tt, $tt.ts_{init}$);
> *disambiguate*(tt, $tt.ts_{init}$);
> **for each** $leafstate \in tt.TS$: ($leafstate \notin tt.TS_{term} \wedge$
> $\nexists \langle leafstate, ts_{end}, A_{label} \rangle \in tt.A$)
>   *prune-nonterm-path*(tt, *leafstate*);

The first step, propagating state values forward, ensures that the timetree reflects persistence of state. This step must come first because it propagates state knowledge that is necessary for collapsing virtual activities, the next step. After virtual activities are collapsed wherever possible, *disambiguate* resolves ambiguities among remaining virtual

activities (as well as actual activities), and *prune-nonterm-path* removes paths that do not lead to terminal timestates.

We state that this operation is performed after composition is complete, even though it may seem that it could be applied at any stage during timetree composition. As we indicated in Section 6.5.3, uncontrolled propagation of state information at an early stage of composition can obscure important knowledge about which activities affect which state components. This is particularly a problem in interleaving and related operations, in which new activities can be inserted "in the middle of" an existing timetree. We discuss this issue in more detail in our treatment of interleaving.

6.6.1 Sequence

Sequential composition is the default form of composition in UAN specifications. A list of tasks or actions appearing from left to right within a column, or in successive rows within a column, specifies sequential performance of those tasks or actions from left to right and top to bottom. We represent this form of composition using the timetree sequence composition operator.

The sequence composition operator of Section 5.3.2 combines two timetrees *ta* and *tb* by appending a copy of *tb* to each terminal timestate of *ta*. More precisely, the initial timestate of each copy of *tb* is replaced by a terminal timestate of *ta*. But all the state information associated with the initial timestate of *tb* is also added to each terminal timestate of *ta* during this operation. As we indicated, this can lead to state conflicts.

To avoid state conflicts and preserve state information as discussed in Section 6.5.3, we interpose a virtual activity between each copy of *tb* and *ta*. In fact, we can do this without introducing any new timetree operators, by prepending a virtual activity to *tb* and then composing this new timetree with *ta*.

The existing sequence composition operator composes two timetrees, not an unattached activity arc and a timetree. So to prepend a virtual activity to *tb*, we must make a timetree that consists only of one virtual activity, then compose this with *tb*. We define this special timetree *spacer* as follows:

$spacer.TS := \{ a, b \}$

$spacer.SCV := \emptyset$

$spacer.SCM := \emptyset$

$spacer.AV := \emptyset$

$spacer.A := \{ <a, b, \emptyset> \}$

$spacer.ts_{init} := a$

$spacer.TS_{term} := \{ b \}$

That is, *spacer* consists of two timestates with *no* associated state information, joined by a virtual activity arc.

Now, we can add a virtual activity in front of *tb* by composing *spacer* and *tb* to yield (*spacer* ; *tb*). As we said at the beginning of this section, sequential composition combines state information from terminal timestates of the first timetree and the initial timestate of the second timetree, and this can lead to state conflicts. But since the terminal timestate of *spacer* has no associated state information, no state conflicts can result when we compose it with *tb*. Similarly, this new timetree's initial timestate has no associated state information, and so no state conflicts can occur when we compose it sequentially with *ta*.

We can illustrate this process using the timetrees from Section 6.5.3 (Figures 6.11 and 6.12). In Figure 6.16, *ta* is the timetree describing **X!**, and *tb* is the timetree describing **Y!**.

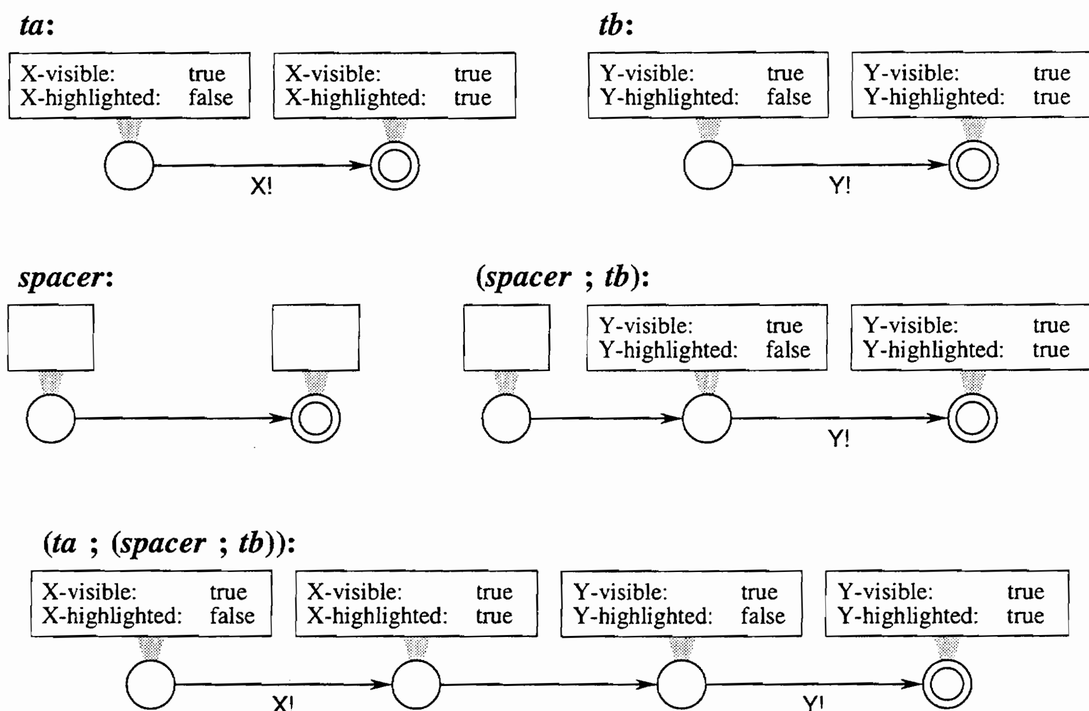


Figure 6.16. Sequential composition with a virtual activity.

Timetrees generated in this way always contain virtual activities between the first timetree and copies of the second, whether or not there is a state conflict between the two. The *collapse-VAs* operator of Section 6.5.2 can remove virtual activities where there is no state conflict and prune copies of *tb* where there is a state conflict. This pruning may leave paths that do not lead to terminal timestates; *prune-nonterm-path* can remove these. In some cases, it may be necessary to propagate state information from earlier timestates to determine whether a state conflict will occur, and this must be done before collapsing and pruning can take place.

These three steps appear in the *reduce* operator from the beginning of Section 6.6. The other step, *disambiguate*, is necessary for choice composition, which we consider next.

6.6.2 Choice

The UAN provides a choice operator for situations in which one of several actions or tasks may occur. The keyword **OR** or the vertical bar symbol (|) specify choice. We use the timetree choice composition operator to represent this form of composition.

Like the sequence operator, the choice composition operator of Section 5.3.1 combines timestates with potentially conflicting state value assignments. In this case, the initial timestate of the new timetree (*ta* | *tb*) gets state value assignments from the initial timestates of both *ta* and *tb*. The technique of the previous section can be modified to handle this situation as well.

Since the definition of choice composition treats *ta* and *tb* symmetrically (that is, neither timetree is "first" or "second"), we prepend a virtual activity to *each* timetree, yielding (*spacer* ; *ta*) and (*spacer* ; *tb*). We then combine these timetrees to yield ((*spacer* ; *ta*) | (*spacer* ; *tb*)). Again, since the initial timestate of *spacer* has no associated state information, the initial timestates of (*spacer* ; *ta*) and (*spacer* ; *tb*) also have none, and no state conflicts can arise when we combine them. Figure 6.17 illustrates this process with the timetrees from Section 6.5.3.

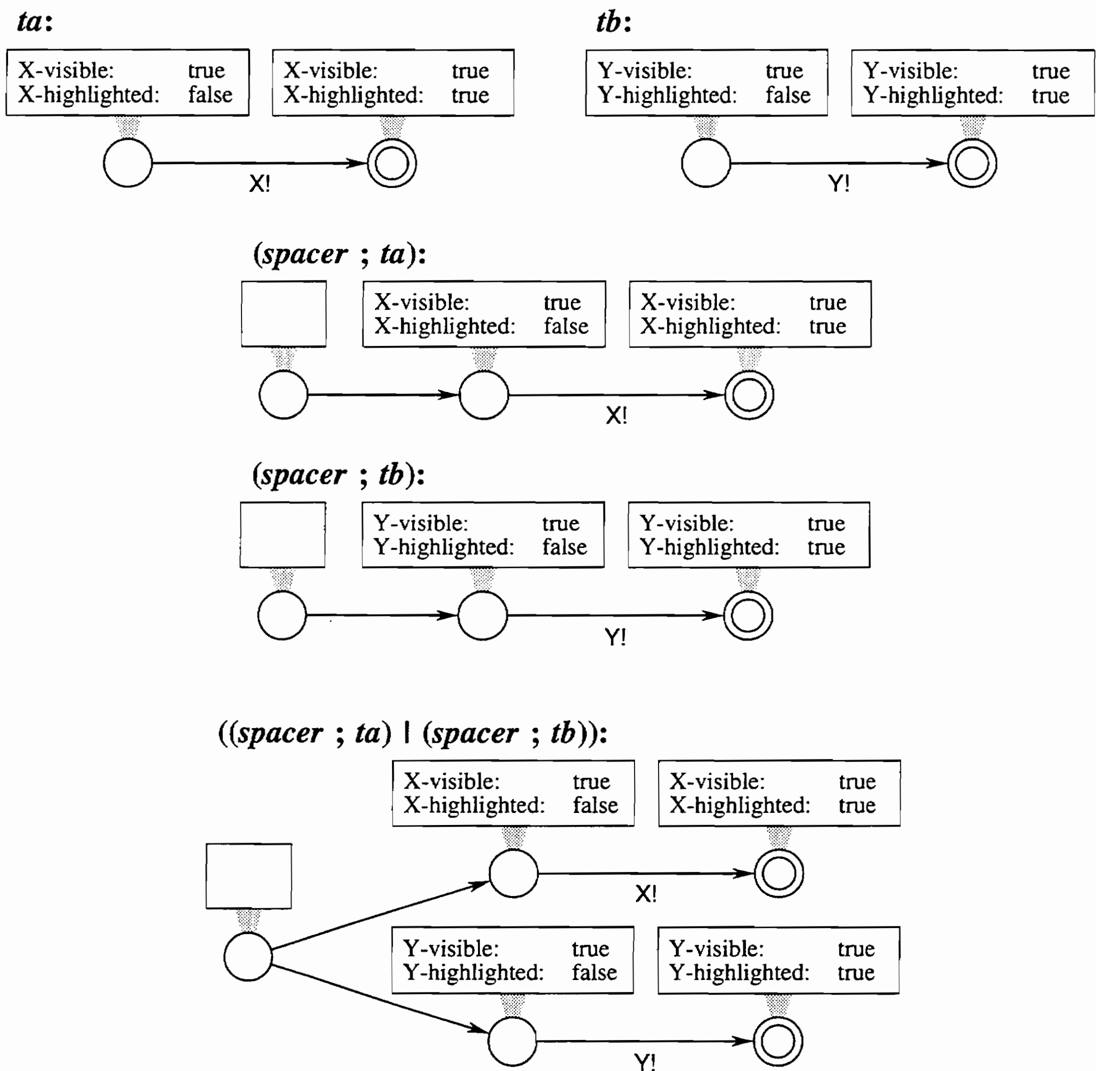


Figure 6.17. Choice composition with virtual activities.

This form of composition always yields a timetree whose initial timestate has no associated state information, and only virtual activity arcs originate at this timestate. As with sequential composition, these virtual activities appear whether or not there is a state conflict between the initial timestates of *ta* and *tb*. The *disambiguate* operator of Section 6.5.2 can combine these virtual activities when there is no such conflict. This still leaves an initial virtual activity where there was none before; this virtual activity can remain, or may be removed by some future application of *reduce* after more compositions take place.

6.6.3 Grouping

The UAN uses parentheses as grouping operators for actions or subtasks. For example, if a viability condition specifier is followed by a group of actions in parentheses, the viability condition applies to that entire group of actions; without the parentheses, it would apply only to the first action. Parentheses can also group actions or subtasks to indicate precedence during composition, much as they are used in algebraic notation.

We handle grouped activities or subtasks by rearranging the order of composition operations according to the precedence that the parentheses specify. We combine timetrees representing actions and subtasks within parentheses first, then combine these timetrees according to the preconditions or composition operators that apply to the group. Figure 6.18 illustrates this process for two different groupings of a series of compositions.

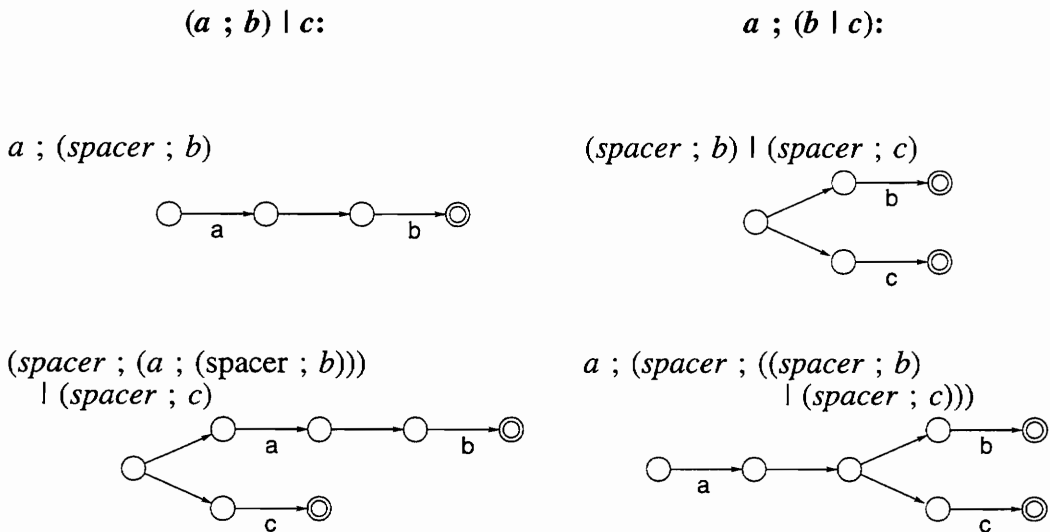


Figure 6.18. Grouping and composition.

6.6.4 Repetition

In Section 5.3.6, we informally defined the timetree closure operator in terms of repeated choice and sequence. This conceptual view reflects the definition of the UAN's star, plus, and numeric-superscript operators. **A*** means zero or more repetitions of task **A**, $\mathbf{A}^{+}$ means one or more repetitions, and $\mathbf{A}^{\mathbf{n}}$ means exactly **n** repetitions.

The timetree closure operator generates a timetree *ta** by recursively appending new copies of *ta* to each terminal timestate of the original *ta*, and adding $ta.ts_{init}$ to $ta^{*}.TS_{term}$. This can lead to state conflicts in the same way as sequence composition. The same solution we adopted for sequence and choice also works in this case; instead of applying the timetree closure operator to *ta* directly, we apply it to (*spacer* ; *ta*). Figure 6.19 illustrates this process for a small timetree.

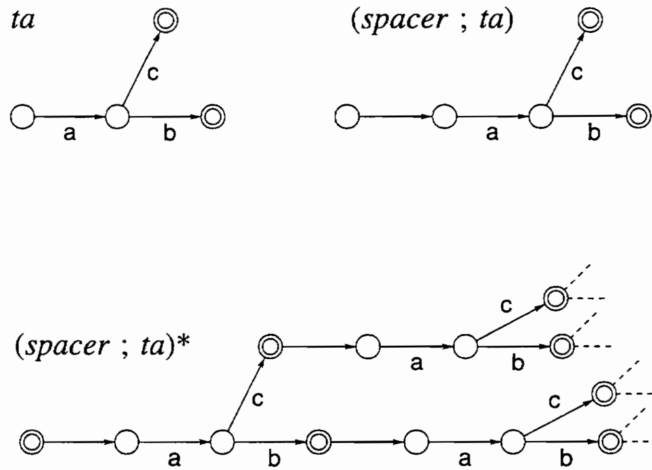


Figure 6.19. Star closure for timetrees representing UAN tasks.

Note that this operator, like the choice operator, yields a timetree whose initial timestate has no associated state information. To understand this, consider again the informal definition of closure from Section 5.3.6. We defined the closure operator in terms of a series of choice compositions:

Null | (*ta* | ((*ta* ; *ta*) | ((*ta* ; *ta* ; *ta*) |...

If we abstract the entire non-null part of this series into one timetree, we can express this composition as

Null | $ta_{non\text{-}null}$

Using the choice operator of Section 6.6.2, we can represent this composition with the timetree of Figure 6.20 (a). But since timestates 1 and 2 have identical associated state information (specifically, none at all), we can condense the virtual activity that joins them, yielding the timetree of Figure 6.20 (b).

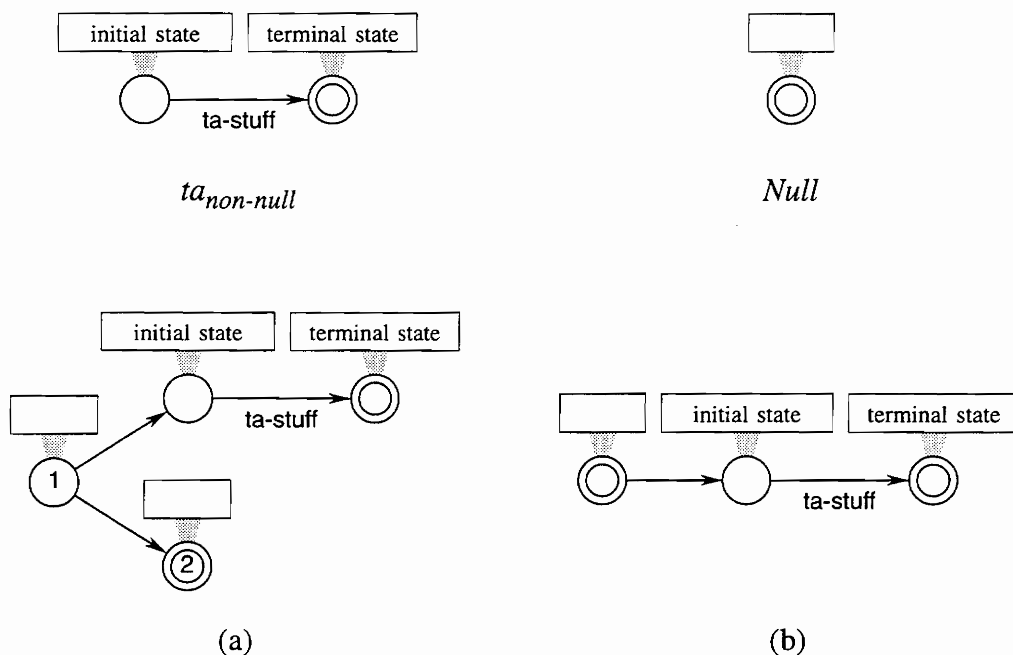


Figure 6.20. Choice composition with the null timetree.

This is the same kind of structure that results from applying the timetree closure operator to *ta*, as illustrated in Figure 6.19 — an initial timestate that is terminal, with one virtual activity that leads to the rest of the timetree.

We do not explicitly generate the timetree for (*spacer* ; *ta*)* by a series of choice and sequence compositions, for the same reason we presented in Section 5.3.6: an infinite series of choice compositions would yield an infinitely branching timetree, which our disambiguation algorithm is not designed to process. Instead, we instead generate the timetree breadth-first.

The definition of $\mathbf{A}^+$ differs from that of $\mathbf{A}^*$ only in that **A** must occur at least once. We can define this operator in terms of sequence and star-closure: $\mathbf{A}^+ := \mathbf{A} ; \mathbf{A}^*$. We can then generate the timetree for $\mathbf{A}^+$ using the operators we have already defined. Similarly, we can generate the timetree for $\mathbf{A}^\mathbf{n}$ by $\mathbf{n}$-1 applications of sequential composition. (This is true for any finite **n**. Any infinite **n** can be considered equivalent to *, since issues of cardinality have not presented a problem to date among UAN designers or users, and it seems unlikely that they ever will.)

6.6.5 Optional performance

The UAN notation **{ A }** indicates that **A** may be performed once or not at all. We can represent this in terms of a choice between the null timetree and the timetree representing **A**: (*spacer* ; *Null*) | (*spacer* ; *ta*). Figure 6.20 in the previous section illustrates the result of this composition.

6.6.6 Order independence

In some situations, the order in which two tasks occur is not important, but it is necessary that both occur. The UAN provides the order-independent composition operator, **&**, to represent sequential composition in which order is not significant.

If two tasks **A** and **B** can execute in any order, but both must execute, it is obvious that the two possible sequences of behavior are **A B** and **B A**. So the UAN expression **A & B** can be rewritten as **(A ; B) | (B ; A)**. We define the timetree representing **A & B** using this technique and the composition and choice operators we have already defined. For two timetrees *ta* and *tb*, we generate the timetree representing their order-independent occurrence as (*spacer* ; (*ta* ; (*spacer* ; *tb*))) | (*spacer* ; (*tb* ; (*spacer* ; *ta*))).

Unlike choice and sequence, order-independent composition is not associative [Hartson & Gray, 1992]. For example, A & (B & C) is not the same as (A & B) & C; the first allows the sequence A C B, but the second does not. The UAN allows n-ary order-independent composition, and while this cannot be reduced to repeated applications of the binary order-independent operator, we can represent it by enumerating all possible sequential orderings of the tasks. (This enumeration is necessarily finite, since the n-ary form still can only take a finite number of arguments.)

6.6.7 Interruptibility and interleavability

The UAN provides some facilities for describing *interruptible* and *interleavable* task performance. The UAN expression **A** $\rightarrow$ **B** indicates that task **A** can interrupt task **B** — that is, that the user can suspend task **B**, perform task **A** to completion, and resume task **B**. The expression **A** $\Leftrightarrow$ **B** indicates that the user can switch back and forth between tasks

A and **B** — that is, any part of task **A** can interrupt task **B**, and any part of task **B** can interrupt task **A**.

The definitions of interruption and interleaving in [Hartson & Gray, 1992] refer to the concept of *primitive actions*, single indivisible actions that cannot be interrupted. These actions represent the finest grain possible for interleaved execution; execution can be interleaved *among* primitive actions, but not *within* primitive actions. Current practice in the UAN treats individual user actions as primitives, with the occasional exception of path-independent cursor movement (~) (since path-independent cursor movement can be interpreted to include an implicit ~**[x,y]***). Previous work on the definition of interleaving has focused on user actions, so the notion of primitive system feedback or other system activity has not been addressed in detail, but it is consistent with current practice to treat individual system actions as primitives as well.

The UAN also allows specification of uninterruptible tasks or subtasks. Angle brackets (< and >) enclosing part of a task description indicate that that part of the task cannot be interrupted by other user actions at any level. This notation is frequently used in descriptions of dialogue boxes and other modal dialogues.

To model interruption, interruptibility and interleavability using timetrees, we use the interleaved composition operator of Section 5.3.3. This operator interleaves timetrees down to the level of individual activity arcs and no lower. This characteristic of the operator affords one simple treatment of primitive actions and uninterruptible tasks.

To generate a timetree representing the interleavability of two tasks represented by timetrees, we can require that subtrees representing uninterruptible subtasks be abstracted to subtrees of depth 1 (using, for example, the *condense-seq* operator of Section 5.4.2), and that the rest of the timetrees be expanded to the level of primitive actions. This restriction ensures that the interleaving algorithm will not insert activity arcs within any path through an uninterruptible task, and that all possible interleavings among other activity sequences will be generated.

Sometimes, though, it may be desirable to indicate that a particular path or subtree of arbitrary depth represents an uninterruptible sequence. Since interruption and

interleaving involve inserting new activities *between* two activities, interruptibility can be viewed as a characteristic of *junctions* of activities — in other words, a characteristic of timestates.

To represent this characteristic, we introduce the concept of a *virtual state component*, analogous to virtual activities. Virtual state components do not represent an aspect of user, system, or environmental state; instead, they represent information significant only during timetree composition. In this instance, we introduce a state component called *(uninterruptible)*, with range {*first*, *internal*, *last*}. The timestate at which an uninterruptible sequence starts is labeled with *(uninterruptible) = first*. Timestates within the sequence are labeled with *(uninterruptible) = internal*, and the timestate at which the sequence ends is labeled with *(uninterruptible) = last*. Note that these labels can be applied to a number of paths forming a subtree, to represent an uninterruptible subtask with several possible outcomes.

Upon encountering these markings, our interleaving operator will copy the entire marked subtree without interleaving any other activities into it. This yields a timetree representing uninterrupted performance of the subtask. For primitive and abstracted activities, the interleaving operator yields the desired result without any additional markings, as described previously.

So far, no definitive list of UAN primitive actions exists; previous treatments have only presented examples, such as key presses or mouse movement. For now, we adopt a subset of user actions from Section 6.2 and system feedback actions from Section 6.3 as our canonical list of primitive actions. The user actions of key (or button) press and release are primitive; so are cursor movements, with the caveat that path-independence implies possible additional movements before or after the specific movement denoted. Highlighting, unhighlighting, displaying and erasing are also primitive.

It is worth noting that each of these activities can, in principle, be broken down into smaller sub-activities at lower levels of abstraction. But traditional applications of the UAN have viewed these activities as the lowest level of abstraction that is of interest to the designer. While it is possible to adapt the UAN for work at other levels of detail, for

example representing physiological limitations or cognitive activities, we do not consider such areas in this work.

The UAN interruptibility operator indicates that one task *can* interrupt another, but not that one *must* interrupt another. For instance, the canonical example of interruptibility [Hartson & Gray, 1992], [Hix & Hartson, 1993] is the expression **help → edit document**. The standard interpretation of this expression is "...a user can invoke the help task at any time during editing, but closure of the help task is required before editing can continue." [Hix & Hartson, 1993] (p. 198) In effect, then, **edit document** is actually interrupted zero or more times by the **help** task — in other words, there is an implicit closure operator applied to the first argument of →.

The situation is somewhat more obscure for mutual interleavability (⇔). Some examples of UAN interleavability from the literature appear to introduce some inconsistencies with previous definitions of composition operators, and in an effort to define a single consistent framework of definitions, we must clarify our view of the interleavability operator.

An example from [Hix & Hartson, 1993] (p. 198) defines the task **manage calendar** as

(**access appointment**
⇔ **add appointment**
⇔ **update appointment**
⇔ **delete appointment**
⇔ **establish alarm**) $^{+}$

The sense of this example seems to be that the user must perform at least one of these tasks at least once, but beyond that can repeat any number of the tasks, any number of times, interleaved with one another. In fact, the discussion of the example explicitly states that this construct also allows *multiple instances of the same task at once.*

The UAN does not have an explicit operator for "at least one" — choice specifies that exactly one task is performed, and sequence and order independence specify that exactly all tasks are performed. It is possible, though, to translate the "at least one" modality into an explicit choice among each possible combination of arguments.

But if we impose this interpretation on the interleavability operator, it leaves us unable to specify the case in which *all* argument tasks must be performed, but may be interleaved. If, instead, we interpret the operator to mean that *all* the interleaved tasks must be performed, we can still specify the situation where any number of them may be performed through explicit choice, as described above. For this reason, we choose here to model the latter interpretation of interleavability. If two (or more, since the operator is associative) tasks are specified as interleavable, they must both be performed to completion. Note that they still do not *need* to interrupt one another — the sequences **A B** and **B A** both satisfy **A** $\Leftrightarrow$ **B**.

The original interpretation of the **manage calendar** example also involves a new interpretation of the $^+$ closure operator. It indicates that multiple instances of the same task may execute in an interleaved fashion, and implies that new instances of a particular task may begin while old instances are not yet finished. This is very different from the usual interpretation of $^+$ or *, which involves distinct repetitions of a well-defined task. In a sense, the example above has task structure "escaping from the parentheses" to interact between iterations. It is unclear how such an interpretation of closure could be applied in other situations.

The concept of multiple interleaved instances of a single task does not seem unusual. It is certainly not unusual to encounter multiple occurrences of a single task in sequence — indeed, that is just what closure operators usually describe — and the UAN provides clearly defined facilities for specifying such a situation. It is also not unusual to encounter behaviors that seem to reflect multiple interleaved instances of a single task. In the calendar management example above, one interface design has each instance of a subtask executing in its own window. The intent of the design is that the user could have several **access appointment** windows, for example, each referring to a separate appointment.

Unfortunately, the UAN does not yet provide composition modalities to represent this kind of structure explicitly. As we have illustrated above, stretching existing operators to represent new structures leads to ambiguities, and can actually *reduce* the potential expressiveness of the UAN in some cases. Neither the interval-based definition of

interleaving from [Hartson & Gray, 1992] nor the definition of interleaving in this section prohibit self-interleaving of multiple instances of a task. Indeed, the UAN currently allows structures like (**A** ⇔ **A**); it is only the interleaved equivalent of closure that the UAN cannot represent.

Since the original interpretation of closure in the UAN has a clear formal definition, and since the interpretation from the example above does not, we elect to retain the original interpretation of closure. We have already provided a timetree-based definition of this operator in Section 6.6.4.

Under the interpretations we have chosen to use, the UAN cannot express the sort of behavior the example above was originally intended to represent — specifically, a form of interleaving that allows multiple instances of one task. Work is currently under way to add explicit notations for these additional modalities to the UAN, and we address some issues associated with these modalities in Chapter 9.

Neglecting the possibility of multiple instances of the same task, we can rewrite the **manage calendar** example under our current interpretation as

(**access appointment**
  | **add appointment**
  | **update appointment**
  | **delete appointment**
  | **establish alarm**) ⇔
(**access appointment***
  ⇔ **add appointment***
  ⇔ **update appointment***
  ⇔ **delete appointment***
  ⇔ **establish alarm***)

This definition captures the requirements that the user must perform at least one subtask at least once, and that any number of any of the other subtasks may be interleaved with it and with each other.

If we introduce an operator *⇔ similar to closure, but allowing instances of its argument to be interleaved rather than sequential, we can capture the original intent of the example as

(access appointment
| add appointment
| update appointment
| delete appointment
| establish alarm) ⇔

(access appointment*⇔
⇔ add appointment*⇔
⇔ update appointment*⇔
⇔ delete appointment*⇔
⇔ establish alarm*⇔)

Admittedly, this is more unwieldy than the original example, but it is also unambiguous and consistent with previous definitions. Since this is still an ongoing area of UAN research, we do not provide a formal definition of *⇔ at this time.

Like the other composition operators of Chapter 5, the interleaving operator of Section 5.3.3 does not include logic for resolving state conflicts. The simple approach of prepending *spacer*, that is, adding a virtual activity at the beginning of each timetree, will not suffice in this case, because the interleaving operator combines internal timestates as well as initial and terminal timestates. Instead, we must prepend a copy of *spacer* to *each activity arc* as we build the interleaved timetree. In other words, where the algorithm of Section 5.3.3 adds copies of individual arcs to the new timetree, our modified algorithm will effectively add a copy of *spacer* followed by a copy of an individual arc. This will preserve the state information associated with both start and end timestates of each activity arc, since the algorithm will combine these timestates with the end and start timestates of *spacer*, which themselves have no associated state information.

We present here the modified version of the depth-restricted interleaving operator from Section 5.3.3. Since the modified algorithm prepends a virtual activity to each actual activity, the resulting tree can reach depth $d + 1$ rather than d. For simplicity, we do not modify the algorithm to generate a tree of depth d.

The loops to iterate over arcs in the source timetrees now contain additional logic to copy uninterruptible subtrees. The first timestate in an uninterruptible subtree is not treated specially. From such a timestate, it is possible to select an activity from the other source timetree instead of selecting an activity from the uninterruptible subtree. Encountering an *internal* timestate indicates that we have entered an uninterruptible subtree, and at this point the algorithm invokes *copy-unint* to copy the rest of the subtree as discussed previously. Since *copy-unint* copies until it reaches *last* timestates, effectively consuming all the *internal* timestates in the subtree, it is guaranteed that *interleave* will only see the first level of internal timestates in the subtree. Thus, *interleave* will not improperly interleave parts of an uninterruptible subtree before invoking *copy-unint*. The frontier of the subtree is added to *Statelabels* so that normal interleaving can resume afterward.

We form the initial version of $(ta \Leftrightarrow tb)$ as before, combining the state component vocabularies and activity vocabularies of *ta* and *tb*, generating an initial timestate, and leaving other components empty. We then apply the recursive interleaving function.

$interleave\rceil^{d}(istate, astate, bstate)$:

> -- *first, get the depths of* astate *and* bstate*:*
> $Ad := depth(astate)$;
> $Bd := depth(bstate)$;
>
> -- *add the current timestate to* $\mathrm{TS}_{\mathrm{term}}$ *if it corresponds to terminal timestates in both source timetrees:*
> **if** $(astate \in ta\rceil^{Ad}.TS_{term}) \wedge (bstate \in tb\rceil^{Bd}.TS_{term})$
> **then** $(ta \Leftrightarrow tb)\rceil^{d}.TS_{term} := (ta \Leftrightarrow tb)\rceil^{d}.TS_{term} \cup \{istate\}$;
>
> -- *quit if* istate *has* depth $\geq$ d*:*
> **if** $(depth(istate) \geq d)$
> **then return**;
>
> -- *initialize a set of argument lists for recursive calls at the end of this iteration:*
> $Statelabels := \emptyset$;

-- *for each arc leaving the current timestate in* ta, *create in (*ta ⇔ tb*) a virtual activity arc, an actual activity arc, and timestates corresponding to the start and end timestates of the current arc. Copy state information from start and end timestates in* ta *to (*ta ⇔ tb*).*

for each $\langle astate, adest, Label \rangle \in ta\rceil^{Ad+1}.A$

$newstart := tsnew();$

$newend := tsnew();$

$(ta \Leftrightarrow tb)\rceil^{d}.TS := (ta \Leftrightarrow tb)\rceil^{d}.TS \cup \{ newstart, newend \};$

$(ta \Leftrightarrow tb)\rceil^{d}.A := (ta \Leftrightarrow tb)\rceil^{d}.A \cup \{ \langle istate, newstart, \varnothing \rangle, \langle newstart, newend, Label \rangle \};$

$(ta \Leftrightarrow tb)\rceil^{d}.SCM := (ta \Leftrightarrow tb)\rceil^{d}.SCM$

$\cup \{ \langle newstart, sc_{name}, sc_{value} \rangle : \langle astate, sc_{name}, sc_{value} \rangle \in ta\rceil^{Ad+1}.SCM \}$

$\cup \{ \langle newend, sc_{name}, sc_{value} \rangle : \langle adest, sc_{name}, sc_{value} \rangle \in ta\rceil^{Ad+1}.SCM \}$

-- *if this is the second timestate in an uninterruptible subtree, copy the subtree without interleaving; otherwise, add to argument list for recursion at end.*

if $\exists \langle adest, (uninterruptible), internal \rangle \in ta\rceil^{Ad+1}.SCM$ **then**

$term\text{-}p := (bstate \in tb\rceil^{Bd}.TS_{term});$ -- true *if* bstate *is terminal*

$Lastlabels := \varnothing;$

$copy\text{-}unint\rceil^{d}(ta\rceil^{d}, adest, term\text{-}p, Lastlabels, (ta \Leftrightarrow tb)\rceil^{d}, depth(newend));$

for each $\langle newend, adest \rangle \in Lastlabels$

$Statelabels := Statelabels \cup \{ \langle newend, adest, bstate \rangle \};$

else

$Statelabels := Statelabels \cup \{ \langle newend, adest, bstate \rangle \};$

-- *Perform the same iteration for each arc leaving the current timestate in* tb*:*

for each $\langle bstate, bdest, Label \rangle \in tb\rceil^{Bd+1}.A$

$newstart := tsnew();$

$newend := tsnew();$

$(ta \Leftrightarrow tb)\rceil^{d}.TS := (ta \Leftrightarrow tb)\rceil^{d}.TS \cup \{ newstart, newend \};$

$(ta \Leftrightarrow tb)\rceil^{d}.A := (ta \Leftrightarrow tb)\rceil^{d}.A \cup \{ \langle istate, newstart, \varnothing \rangle, \langle newstart, newend, Label \rangle \};$

$(ta \Leftrightarrow tb)\rceil^{d}.SCM := (ta \Leftrightarrow tb)\rceil^{d}.SCM$

$\cup \{ \langle newstart, sc_{name}, sc_{value} \rangle : \langle bstate, sc_{name}, sc_{value} \rangle \in tb\rceil^{Bd+1}.SCM \}$

$\cup \{ \langle newend, sc_{name}, sc_{value} \rangle : \langle bdest, sc_{name}, sc_{value} \rangle \in tb\rceil^{Bd+1}.SCM \}$

```
        if ∃ <bdest, (uninterruptible), internal> ∈ tb⌉^(Bd+1).SCM then
            term-p := (astate ∈ ta⌉^(Ad).TS_term); -- true if astate is terminal
            Lastlabels := ∅;
            copy-unint⌉^d(tb⌉^d, bdest, term-p, Lastlabels, (ta ⇔ tb)⌉^d,
                depth(newend));
            for each <newend, bdest> ∈ Lastlabels
                Statelabels:= Statelabels ∪ { <newend, astate, bdest> };
        else
            Statelabels := Statelabels ∪ { <newend, astate, bdest> };

    -- recurse on each timestate added to (ta ⇔ tb). If astate and bstate were
       both leaf nodes, Statelabels is empty, and the recursion terminates.
    for each <newdest, newA, newB> ∈ Statelabels
        interleave(newdest, newA, newB);


copy-unint⌉^d(tt, ts_cur, term-p, Lastlabels, tdest, depth):

    if depth ≥ d return;

    -- For each arc leaving the current timestate in tt, copy the arc, its end timestate,
       and associated information into tdest. If we were at a terminal timestate in
       the other tree (term-p = true), add terminal end timestates to tdest.TS_term.
       For timestates marked with (uninterruptible) = internal, recurse; for timestates
       marked last, add to Lastlabels; for timestates with no value assigned to
       (uninterruptible), add to Lastlabels.
    for each <ts_cur, ts_end, Label> ∈ tt..A
        newend        := tsnew( );
        tdest.TS      := tdest.TS ∪ { newend };
        if (ts_end ∈ tt.TS_term) ∧ term-p
            then tdest.TS_term := tdest.TS_term ∪ { newend };
        tdest.A       := tdest.A ∪ { <ts_cur, newend, Label> };
        tdest.SCM     := tdest.SCM
              ∪       {       <newend, sc_name, sc_value>:
                              <ts_end, sc_name, sc_value> ∈ tt.SCM };
        if ∃ <ts_end, (uninterruptible), internal> ∈ tt.SCM
            then copy-unint(tt, ts_end, term-p, Lastlabels, tdest)
            else Lastlabels := Lastlabels ∪ { <newend, ts_end> };
```

Information about possible interleavings of timetrees can be lost if state has been propagated through one or both of the timetrees, as noted at the beginning of Section 6.6. Consider the timetree representing the interleaving of two tasks, **X! Y!** and **X-!**. (For

notational convenience, we indicate the values of state components *X-highlighted* and *Y-highlighted* as X!, X-!, Y!, and Y-!.)

Figure 6.21. Timetrees for **X! Y!**, **X-!**, and **(X! Y!) ⇔ X-!**.

Figure 6.21 shows the timetrees for **X! Y!**, **X-!**, and **(X! Y!) ⇔ X-!**. In the first (topmost) branch, **X-!** occurs before **X!**. In the second branch, **X-!** occurs after **X!**, but before the virtual activity connecting **X!** and **Y!**. In the third branch, **X-!** occurs after that virtual activity, and in the bottom branch, it occurs after **X! Y!** has completed.

Figures 6.22, 6.23, and 6.24 show the result of propagating state through this timetree, collapsing its virtual activities, and pruning nonterminal paths that are left after the collapse step.

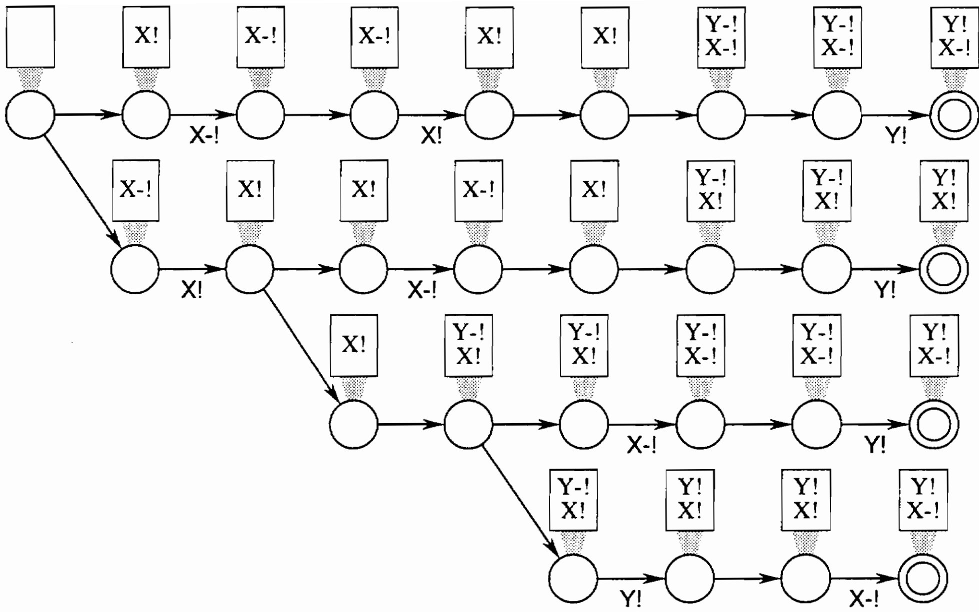


Figure 6.22. Timetree for **(X! Y!)** ⇔ **X-!** after state propagation.

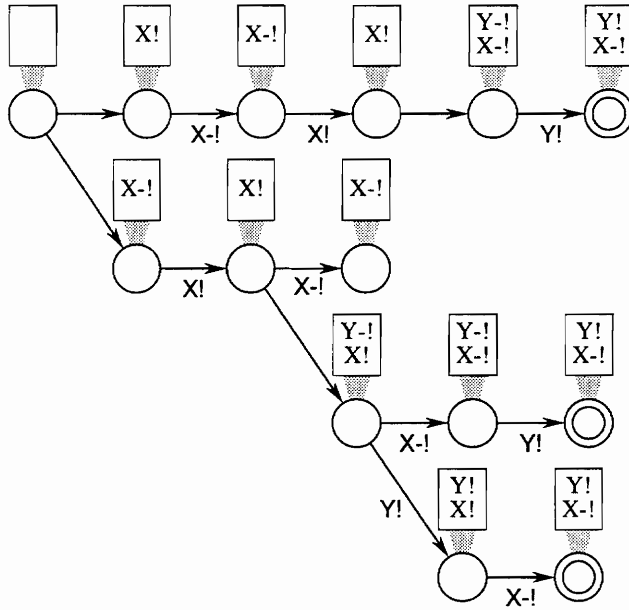


Figure 6.23. Timetree for **(X! Y!)** ⇔ **X-!** after collapsing virtual activities.

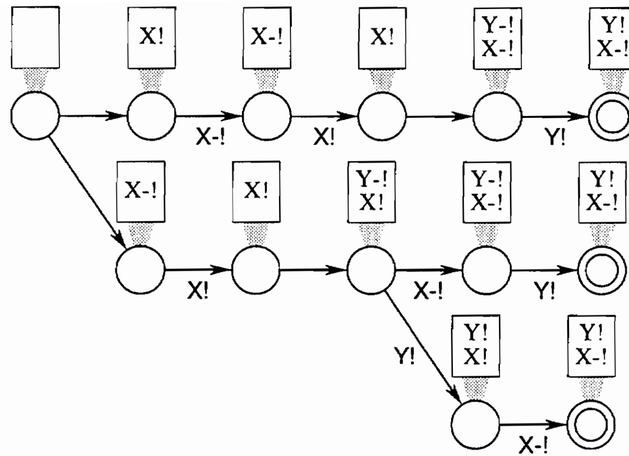


Figure 6.24. Timetree for (**X! Y!**) ⇔ **X-!** after pruning nonterminal paths.

In Figure 6.24, we see the timetree that represents all possible interleavings of **X! Y!** and **X-!**. There are three possible outcomes: **X-!** can occur before **X! Y!**, between **X!** and **Y!**, or after **X! Y!**. Note that the availability of these paths still depends on the values of *X-highlighted* and *Y-highlighted* before the task begins. If *X-highlighted* is *false*, that is, if **X** is not highlighted, then the first path cannot occur, since its first actual activity requires that *X-highlighted* have the value *true*. Conversely, if *X-highlighted* is *true*, the second and third paths cannot occur. Finally, if *Y-highlighted* is initially *true*, indicating that **Y** is initially highlighted, then *none* of the paths can occur, since all of the contain a step that depends on *Y-highlighted* having an initial value of *false*. One can see where these state conflicts arise by assigning appropriate values to *X-highlighted* and *Y-highlighted* in the initial timestate and then propagating state information forward through the timetree.

Now consider the situation in which state information is propagated through the component timetrees *before* they are composed with the interleaving operator.

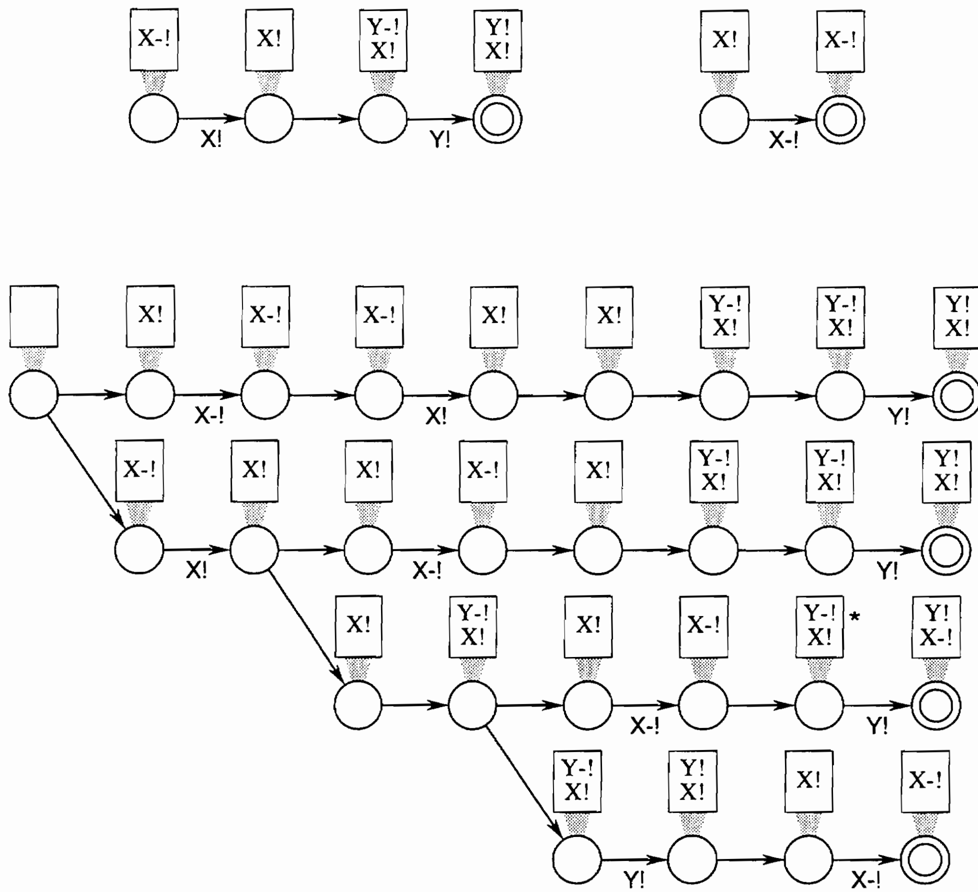


Figure 6.25. Timetrees for **X! Y!**, **X-!**, and **(X! Y!)** ⇔ **X-!** with state propagated before composition.

Figure 6.25 show the result of this composition. The timestate in the third branch marked with an asterisk indicates a state conflict that was not present in the timetree of Figure 6.21. Effectively, the propagation of state through the timetree for **X! Y!** has made it appear that *X-highlighted* must be *true* for **Y!** to occur. In other words, the premature propagation of state has generated a spurious viability condition for **Y!**. Figures 6.26 and 6.27 illustrate the result of collapsing virtual activities and pruning nonterminal paths in this timetree.

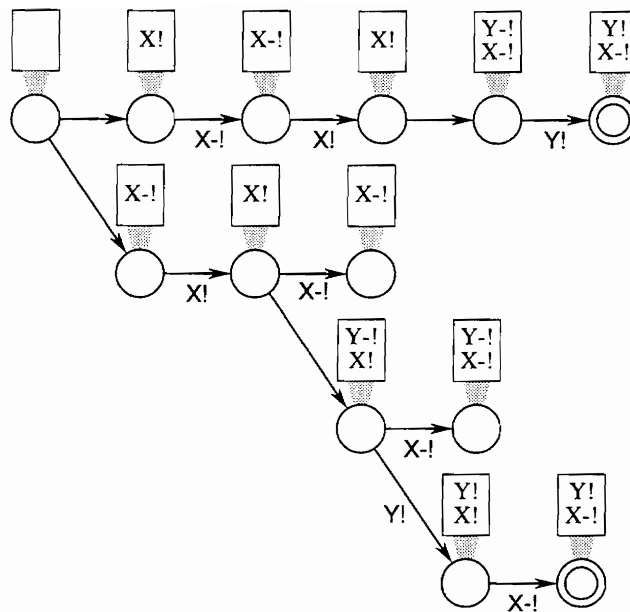


Figure 6.26. Timetree for **(X! Y!)** ⇔ **X-!** with state propagated before composition, state propagated after composition, and virtual activities collapsed.

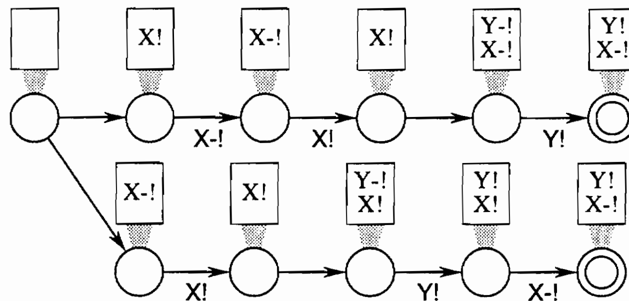


Figure 6.27. Timetree from Figure 6.26 for **(X! Y!)** ⇔ **X-!** after pruning nonterminal paths.

The timetree of Figure 6.27 describes only two possible paths, **X-!** occuring before **X! Y!** and after **X! Y!**. The path along which **X-!** occurs between **X!** and **Y!** has been pruned because of the spurious viability condition associated with **Y!**.

This situation arises because our interpretation of state values gives them two slightly different meanings. When we use them to represent conditions of viability in task definitions, and when we use them to indicate the effect of activities on state components

in activity definitions, they represent values of state that hold at the beginning or the end of *any* instance of the task or activity. But state propagation reflects the persistence of state values until they are specifically changed, and so after state propagation, the set of state component value assignments at the beginning or end of an activity arc represents what is known about state at the beginning or end of *that particular instance* of the activity. It is this dual interpretation that allows us to use activity and task definitions to determine state conflicts and pare off conflicting timetree branches. When composing timetrees, we insert virtual activites and preserve knowledge about the relations between state values and activites. After composition, we propagate state information to reflect state persistence, then prune paths that lead to state conflicts.

6.6.8 Concurrency

The UAN provides a concurrent composition operator analogous to its interleavability operator. The UAN expression **A || B** indicates that actions from task **A** can take place *simultaneously* with actions from task **B** — a primitive action from one task can begin before a primitive action in the other task has ended. This differs from interleavability, in which actions from two tasks can alternate, but cannot overlap in time. Concurrent activities might take place, for example, in an interface that allows a user to manipulate separate controls simultaneously with both hands.

Existing discussions of UAN concurrency raise the same issues of interpretation as we discussed for interleaving. It is unclear whether (**A || B**) requires *both* **A** and **B** to be performed, or only *one*. For the same reasons discussed previously, we choose to interpret (**A || B**) as requiring performance of both **A** and **B**, but potentially allowing actions from each to take place concurrently.

Section 5.3.4 presented a definition of concurrent composition based on the interleaving operator of Section 5.3.3. We base our definition of the UAN concurrency operator on this timetree composition operator. As with other UAN composition operators, though, modifications are necessary to deal with state conflicts. In addition, the usual interpretation of interruptibility in the UAN indicates that uninterruptible tasks disallow other interleaved *or concurrent* activities, so the concurrency operator must observe interruptibility specifiers.

The tripartite activity view of Section 5.3.4 breaks a single activity into three stages: initiation, continuation, and termination. The divisions between these stages are abstract, and do not correspond to specific events within the activity itself. Instead, they simply provide a way to break an activity into parts that can be distributed across other activities. Without this facility, it would be difficult to specify a situation in which an arbitrary number of activities take place while a single activity continues concurrently. For convenience, we choose to interpret initiation and continuation activities as having zero duration. They do not specify "the very first and last part" of an activity; instead, they are abstract *instantaneous* activities serving only as bounds for the continuation activities that represent actual performance.

While this approach allows us to describe concurrent activities spanning other activities, it clouds the issue of state conflicts and state propagation. State configurations at the start and end timestates of an activity arc describe state values before and after the activities labeling the arc are performed. These state configurations are determined by activity definitions (like those in Sections 6.2 and 6.3), by explicit viability conditions, and by propagation from preceding states. But since the new timestates introduced between initiation, continuation, and termination activities represent abstract divisions *within* performance of the original activity, state values at these timestates are indeterminate. (Since initiation and termination activities are instantaneous, it would be reasonable to specify that state component values do not change across them. But this assumption yields no benefits, and it complicates the generation and composition of concurrent specifications, so we avoid it.)

Without knowledge of intermediate state component values, it becomes impossible to detect state conflicts in timetrees resulting from concurrent composition, and so it is impossible to determine which paths are allowable. The virtual state component *(uninterruptible)* provides some information about allowable patterns, since it is constant by definition throughout an uninterruptible sequence, but this is not enough to yield useful results.

At least two assumptions can ameliorate this problem. We have already assumed that state components unmentioned in an activity definition are unaffected by that activity;

this allows state propagation. We can also assume that state components with the same value at the beginning and end of an activity maintain that value throughout the activity. Plainly, though, this is not always true; consider, for example, the value of *mouse-key-state* during the user activity **Mv^**. In such situations, it is impossible to reason about state within the activity until the activity is specified at a lower level of detail (in this example, **Mv M^**). We require that activities be specified at this level of detail before they can be combined concurrently.

We enforce this requirement by introducing a special state value, called *(indeterminate)*, that is not in the range of any state component. (Alternatively, we could add it to the range of each state component, with the understanding that it never appears as an actual state value within a task description.) When we partition an activity, we create an internal timestate between the initiation and termination activities. For state components that are invariant across the activity, we copy their invariant values to this timestate; for state components whose values are changed, we assign the special value *(indeterminate)* at this timestate.

After the interleaving stage of concurrent composition, state propagation will copy these values across other activities that take place concurrently with the continuation of the partitioned activity. We modify the *collapse-VAs* operator of Section 6.5.2, which prunes subtrees rooted at a state conflict, so that it fails upon detecting a conflict between *(indeterminate)* and any other state value. In this way, we can detect an effort to combine activities that both manipulate the same state component, and report an error rather than yielding an incorrect result.

We modify the *partition-activities* operator of Section 5.3.4 to propagate state as described above. We also refrain from partitioning uninterruptible activities; since they cannot be concurrent with any other activities, there is no need to distribute continuations, and we can maintain them in their original form.

partition-activities(*tt*):

```
-- generate new partitioned activities and augment the old activity
   vocabulary
PartAV := { a_initiate, a_continue, a_terminate: a ∈ tt.AV };
tt.AV := tt.AV ∪ PartAV;

-- replace each interruptible activity arc with pair of arcs for initiate and
   terminate, and instantiate new timestates. Copy appropriate state
   component mappings to new timestates.
for each (<startstate, endstate, { act }> ∈ tt.A:
            instate ∉ {internal, end}
            ∀ <endstate, (uninterruptible), instate> ∈ tt.SCM)
    interstate := tsnew( );
    tt.A := tt.A – { <startstate, endstate, { act }> }
            ∪ { <startstate, interstate, { act_initiate }> }
            ∪ { <interstate, endstate, { act_terminate }> };
    tt.TS   := tt.TS ∪ { interstate };
    for each sc_name: (<startstate, sc_name, startval> ∈ tt.SCM ∧
                       <endstate, sc_name, endval> ∈ tt.SCM ∧
                       sc_name ≠ (uninterruptible) )
      if startval = endval
         then tt.SCM := tt.SCM ∪ <interstate, sc_name, startval>
         else tt.SCM := tt.SCM ∪ <interstate, sc_name, (indeterminate)>;
```

Next, we modify the *propagate-continuations* operator so it does not propagate continuations into uninterruptible sequences or virtual activities:

```
propagate-continuations(tt, state):

        -- initialize set of descendants of current node:
        Stateset := ∅;

        -- iterate over each activity arc starting at the current timestate:
        for each <state, endstate, Curlabel> ∈ tt.A

                -- add destination of current arc to Stateset:
                Stateset := Stateset ∪ { endstate };

                -- iterate over arcs starting at endstate:
                for each nextarc: <endstate, nextstate, Nextlabel> ∈ tt.A
                        -- rewrite label of nextarc:
                        if Nextlabel ≠ ∅ ∧ (instate ∉ {internal, end}
                                ∀ <nextstate, (uninterruptible), instate> ∈ tt.SCM)
                            then Auglabel := Nextlabel ∪
                                        { a_continue:    (a_initiate ∈ Curlabel ∨
                                                          a_continue ∈ Curlabel)
                                                        ∧ a_terminate ∉ Nextlabel };
                        -- replace nextarc in tt.A:
                        tt.A := tt.A - nextarc ∪ { <endstate, nextstate, Auglabel> };

        -- recurse on descendants of current node:
        for each tstate ∈ Stateset
                propagate-continuations(tt, tstate);
```

Using these modified operators, we construct $(ta \parallel tb)$ as in Section 5.3.4:

```
        partition-activities(ta);
        partition-activities(tb);
        (ta || tb) := (ta ⇔ tb);
        propagate-continuations((ta || tb), (ta || tb).ts_init);
```

Note that the interleaving operator in the third statement is the version from Section 6.6.7, which incorporates virtual activities.

Finally, to detect attempts to compare indeterminate states, we modify the *collapse-VAs* operator of Section 6.5.2:

collapse-VAs(*tt*, *state*):

> -- *for each virtual activity arc leaving the current timestate, check for state conflicts between its start and end timestates. If there is a conflict and one state is indeterminate, fail with an error. If there is a conflict and both states are determinate, prune off* endstate, *the subtree rooted at it, and the arc leading to it. If there isn't a conflict, recurse on* endstate, *then see if there are any state components whose values are defined in* endstate *but not in* state. *If there are not, remove the arc and combine* state *and* endstate*; if there are, leave the arc unchanged.*

for each *tarc*: <*state*, *endstate*, ∅> ∈ *tt.A*
    if *state-clash(tt, state, tt, endstate)*
        then if $\exists sc_{name}$: (<*state*, sc_{name}, *startval*> ∈ *tt.SCM*
                ∧ <*endstate*, sc_{name}, *endval*> ∈ *tt.SCM*)
                ∧ (*startval = (indeterminate)* ∨
                    endval = (indeterminate))
                ∧ *startval* ≠ *endval*)
            then fail("attempt to compare to an indeterminate state")
            else *prune-state(tt, endstate)*
        else
          collapse-VAs(*tt*, *endstate*);
          if ∄ *name*: (∃ <*endstate*, *name*, sc_{value}> ∈ *tt.A* ∧
                ∄ <*state*, *name*, sc_{value}> ∈ *tt.A*)
            then *replace*(*tt*, *endstate*, *state*);

After state propagation, this version of *collapse-VAs* will report an error if it attempts to prune a branch because of a state conflict where one of the state values is *(indeterminate)* — in other words, if some path's viability depends on a state component whose value is indeterminate due to concurrent activity.

This algorithm might appear to miss one form of conflict. If two different tasks are being composed, and each manipulates the *same* state value, there will be points in the composition at which the internal *(indeterminate)* state value from one task is compared to the *(indeterminate)* state value from the other. At these points, *state-clash* will not report a conflict, even though a conflict exists (the indeterminate values may not be the same).

But in every path through the composed timetree, one of the activities must be initiated first; our method of concurrent composition never yields an activity arc labeled with more

than one initiation activity. When the second activity is initiated during the continuation of the first, the start timestate of that initiation activity is labeled with an actual value for the state component. This inevitably leads to a conflict between an actual value and an indeterminate value, which will cause the composition to fail.

We must note two points about the UAN concurrency operator. First, it represents composition of two tasks that *may* occur concurrently, but does not *require* that any part of one task be concurrent with the other. As with interleaving, **A B** is one acceptable behavior pattern specified by **A || B**. This differs from "mandatory concurrency" in which two activities *must* take place concurrently, as in the cursor-following example of Section 6.3 (Figure 6.4). At present, mandatory concurrency cannot be specified in the UAN, except where it is implicit in the definition of an activity (again, as in cursor-following or rubberbanding feedback).

Second, system and user activity frequently overlap. "Type-ahead," "mouse-ahead," and similar buffered input strategies help to maintain fluid action and response in the face of unpredictable system response time. Our representation of mandatory concurrency can model such behavior, but the UAN ordinarily ignores it, and our definition of the UAN ignores it as well.

6.6.9 Intervals and waiting

The passage of time is implicit whenever sequences of activities take place. While a designer may sometimes view a keypress or the display of an icon as happening instantaneously, these activities actually take some finite time. Some other activities, such as cursor movement, cannot realistically be viewed as instantaneous. And for some activities, such as "double clicking" a mouse button, it is necessary to specify duration explicitly.

The UAN provides explicit indicators for intervals less than or greater than a specified period. For example, a mouse button double click can be specified as **Mv^ (t<n) Mv^** where **n** is the maximum interval between clicks [Hix & Hartson, 1993].

From the perspective of timetrees, it is possible to represent time in at least two compatible ways. We can represent time ("what time is it?") as a component of environmental state, and we can represent the passage of time ("how long?") as an environmental activity, possibly concurrent with other activities.

The state-based representation of time uses an environmental state component with a value at any given timestate equal to the time at which that timestate "occurs" — that is, the time at which the activity leading to that timestate is complete and one of the activities leaving that timestate begins. This implies that timestates are instantaneous, and that time only passes while some activity is taking place. This view is incompatible with the UAN, which (unless otherwise specified) allows arbitrary time to lapse between activities.

To avoid this incompatibility, we can represent intervals explicitly with activity arcs labeled by an environmental activity representing passage of time. In this explicit representation, activity arcs representing UAN actions (user or system) are separated by activity arcs representing elapsed intervals. For example, if the UAN activity **A** ends at time *t*, and the next activity **B** begins at time *u*, the timetree representing the sequence **A B** can be illustrated as in Figure 6.28.

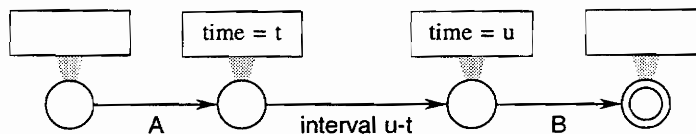


Figure 6.28. Timetree representing **A B** with explicit interval.

Some timetree activities *can* occur instantaneously. The *initiate* and *terminate* activities in the timetree representation of concurrency have zero duration, according to the interpretation we have chosen; the *continue* activity represents actual performance, and *initiate* and *terminate* serve only as instantaneous markers denoting the beginning and end of an activity. Virtual activities do not "occur" at all, strictly speaking, and so no duration is associated with them. Finally, as we stated previously, designers will frequently choose to view primitive actions such as keypresses or highlighting as instantaneous.

This approach provides a way to represent phenomena such as system response time, user reaction time, and even scheduled events. We provide a further example of time representation in Chapter 8.

The definitions of UAN entities and operators presented in this chapter provide explicit details of device representations, temporal relationships, effects on state, and interactions between composition operators, state information, and viability conditions. In the following chapters we will show how these definitions can be expanded to include new input devices, how they can be used to analyze UAN specifications, and directions they indicate for future investigation.

7 Connecting a Device Model to the UAN

In the last chapter we presented a formal definition of the "current" UAN. One strength of the UAN, though, is its extensibility; it is popular in part because different UAN users can change it or add to it, making it more useful for their particular application areas. A formal definition of the UAN that did not allow this process of extension and modification would be of little lasting value. On the other hand, a definition that *does* allow extension will not only support the flexibility that has helped make the UAN so popular; it can also help to maintain consistency, or at least translatability, among various UAN "dialects" that might otherwise drift further and further from any well-defined standard. In this chapter, we present one example of how the timetree model can support extensibility.

One common motive for UAN extension is the introduction of new input devices. Sections 6.1 and 6.2 present definitions for the UAN representation of keys, buttons, and pointing devices, and many new devices are so similar to these that creating new notation for them is trivial. For less familiar devices, it is possible to invent new notation in an ad hoc manner; in fact, in the past there has been no alternative. A systematic and formally defined method for adding new input device representations will help maintain consistency and understandability as the UAN grows to accomodate more interaction devices and styles.

Card, Mackinlay, and Robertson [1990] present a formal and general taxonomy of input devices. Their goal is not only to classify existing devices, but to provide a system that will accomodate, and even suggest, future input device designs. They use this taxonomy to determine various figures of merit for devices based on their classification.

More importantly for our current goal, though, the taxonomy is based on two elements for modeling input devices, a *primitive movement vocabulary* and a set of *composition operators*. These elements provide a system for classifying the user behaviors that manipulate an input device and the values these manipulations can yield. This is the information that UAN representations of input devices require. Since the device model

includes a formal representation of this information, we are able to define formal transformations that map any point in the device space to a corresponding timetree, and the resulting timetree to a set of UAN vocabulary elements.

7.1 The CMR device model

Card, Mackinlay, and Robertson (CMR) postulate that any physical action by which a user manipulates an input device can be broken down into some combination of simple primitive actions, each yielding a one-dimensional "output value." (They specifically exclude input modalities such as speech input or heat-sensing; their model supports only mechanical forces and movements.) In this section, we provide an overview of the device model's primitive movement vocabulary and composition operators.

7.1.1 Primitive movement vocabulary

The CMR input model represents each input device with a six-tuple

<**M**, **In**, **S**, **R**, **Out**, **W**>

M is a *manipulation operator* describing the physical property sensed by the input device. **In** is the *input domain*, the range of values that the manipulation can yield. **S** is the *current state* of the device, a value from the input domain. **R** is a *resolution function* that maps from the input domain to the *output domain*, **Out**. Finally, **W** is a catchall set of device properties which do not fit into the existing categories.

The manipulation operator **M** is one of eight possible primitive operators. Each such operator affects a property described by a combination of three choices: position or force, linear or rotary range, and absolute or relative value. Thus, for instance, an ordinary light switch transduces absolute linear position (up or down), a tuning knob on a radio receiver transduces relative rotary position, and a flat-panel pushbutton transduces absolute linear force (pressure). Each manipulation and property is one-dimensional, and so takes place along (or around) an axis. Figure 7.1 summarizes these properties.

	Linear	Rotary
Position **Absolute** **Relative**	 Position **P** Movement **dP**	 Rotation **R** DeltaRotation **dR**
Force **Absolute** **Relative**	 Force **F** DeltaForce **dF**	 Torque **T** DeltaTorque **dT**

Figure 7.1. Physical properties sensed by input devices. From [Card, Mackinlay, & Robertson, 1990].

The property transduced by an input device has a single value at any given time. This value is **S**, and it must be within the input domain **In**. The CMR model distinguishes "input values," which reflect the physical state of the device, and "output values," which are transmitted to the application using the device. The resolution function **R** translates values from the input domain **In**, reflecting device state, to the output domain **Out**, reflecting the application's view of the device. This distinction between "input" and "output" values is masked in the UAN; we discuss this issue further in Section 7.2.

7.1.2 Composition operators

While the primitive movement vocabulary of the CMR model can represent a wide range of manipulations, many input devices are too complex to be represented by a single tuple with one degree of freedom. A mouse, for example, incorporates two orthogonal one-dimensional movement sensors and one or more buttons, and translates its relative motion into changes in the absolute location of a cursor. The CMR model provides three *composition operators* to support this additional complexity: merge composition, layout composition, and connect composition.

Merge composition combines two devices by taking the cross product of their input domains (and, implicitly, their output domains as well). Thus, for example, the two orthogonal one-dimensional movement sensors of a mouse can be merged to form a single two-dimensional movement sensor.

Layout composition refers to a physical grouping or clustering of devices. In the case of a mouse, the two-dimensional movement sensor and the buttons are layout-composed into a single device.

Connect composition associates two devices by mapping the output domain of one to the input domain of another. In this form of composition the CMR input model views devices as transducers between an input and an output domain, but not necessarily between "a user" and "a system." As a result, for example, the two-dimensional output of a mouse's location sensor can be mapped to the two-dimensional input of a screen cursor. Mouse movements are translated into an output value, which in turn serves as an input to the cursor.

Card, Moran and Newell present a graphical representation for devices within this taxonomy. The individual primitive movements that a device supports are represented as markers on a two-dimensional grid, and composition operators are represented by various styles of lines connecting these markers. For convenience and efficiency, since we discuss only individual devices rather than ranges of devices, we rely on textual descriptions of device compositions.

7.2 The UAN view of devices

The traditional UAN representation of input devices and activities is simpler than the CMR model's representation. The input devices that are part of the canonical UAN — buttons, keys, and pointing devices — were chosen because they were common to the system designs the early UAN was used to represent. Like the rest of the early UAN, the notation for these devices was chosen mainly for the convenience of designers using the UAN.

As a result, these notations — the vocabulary by which the UAN refers to input actions and devices — reflect only the minimal information necessary to capture the designer's intent. Thus, for example, the UAN vocabulary for pressing and releasing a button — **Xv** and **X^** — omits details about preconditions (a button must be "up" before it can be pressed, and "down" before it can be released). As we indicated in Section 6.2.1, this information is sometimes critical. But it would be inconvenient to specify it repeatedly, and so the standard UAN notation leaves it implicit.

While the UAN leaves some information implicit, there are other aspects of device behavior that it does not capture at all, either implicitly or explicitly. The UAN representation of a button captures changes of position (up or down) explicitly, and these changes have implicit viability conditions and effects on state. But the UAN representation says nothing about the button's orientation, the depth to which it must be pushed, or the pressure that must be applied. While such information is important in some designs, it was not important in the design processes to which the early UAN was applied, and so it is not part of the UAN representation.

In fact, the UAN generally expresses input actions in terms of what the CMR device model would call "output values." While **Xv** and **X^** do evoke the pressing and releasing of a key, they abstract away information about position, movement, and pressure. Instead, they manipulate an abstract binary state, reflecting whether a key is "down" or "up." The distinction is clearer in the case of cursor movement via a mouse. The UAN notation for cursor movement refers only to screen locations, and says nothing about the physical movements of a mouse or joystick that control the cursor's position.

Since UAN device representations do not require the detailed information that the CMR primitive movement vocabulary provides, some information from CMR representations is lost in translation to UAN representations. But so long as we do not need to translate in the opposite direction — from UAN representations to CMR representations — this is not a problem.

In Section 7.1.2, we used the example of a mouse to show how the CMR model represents complex devices (via composition operators). We can use this device to compare the UAN's representation strategy to that of the CMR model. For example, a mouse yields a two-dimensional location value. The CMR model represents this by the composition of two one-dimensional transducers; the UAN represents it directly, with cursor-movement actions that yield two-dimensional values. A mouse incorporates both this two-dimensional composite device and one or more buttons. The CMR model represents this as a layout composition; the UAN, instead, treats the buttons and the locator as separate devices, affording different actions and affecting different state components. Finally, the mouse is generally used to manipulate the position of an on-screen cursor. In the CMR model, this is represented by connect composition; in the

UAN, intermediate information about "mouse output" or "cursor input" is not of interest, and the result of mouse movement is expressed directly in terms of cursor movement.

Thus, where the CMR model represents input devices in terms of a primitive movement vocabulary and composition operators, the UAN uses a "flattened" representation requiring considerably less information. The next section presents a technique to generate UAN representations from CMR representations.

7.3 Translation from CMR to UAN device representations

Input device representations in the CMR device model are formed from primitive movement vocabulary tuples and composition operators. To generate UAN representations for these devices, we first transform the CMR representations into timetrees, and then create UAN vocabulary for the actions and state elements that label those timetrees.

7.3.1 Primitive devices and actions

All device descriptions in the CMR model are based on the model's primitive movement vocabulary. An input device with one degree of freedom is described by a six-tuple <**M**, **In**, **S**, **R**, **Out**, **W**>. To create an equivalent UAN representation, we must capture the kind of manipulation (from **M**), the current state of the device (from **S**), and the output domain (from **Out**).

Since the state values and "destinations" of the UAN correspond to the CMR model's "output values," we must represent device state in terms of the image of **S** in the output domain **Out**, rather than the native value of **S** (a value from the input domain). In other words, instead of using **S** as a state value, we must use **R**(**S**), the value obtained by applying the resolution function to **S**. In practice, we do not need to represent either **S** or **R** (or the inverse of **R**) explicitly; instead, we use a state component with a range equal to **Out**. We represent destinations or amounts of manipulations in terms of output domain values, and we keep track of current state in terms of this same domain.

UAN tradition allows the person introducing a new device to make up any name for the actions of that device. While we would like to provide a canonical translation from the

CMR model's classes of manipulation operators to a set of verbs suitable for a UAN representation, the wide variation in possible actions makes it difficult to pick a reasonable set. The chart of Figure 7.2 shows one possible set of names.

	Linear	**Rotary**
Position		
Absolute	**MoveTo(P)**	**TurnTo(R)**
Relative	**MoveBy(dP)**	**TurnBy(dR)**
Force		
Absolute	**Push(F)**	**Torque(T)**
Relative	**Impulse(dF)**	**Wrench(dT)**

Figure 7.2. UAN verbs for manipulation forms.

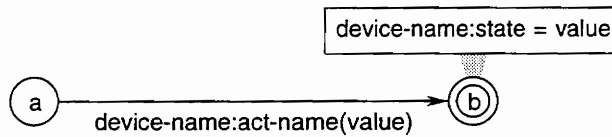
While **MoveTo**, **MoveBy**, **TurnTo**, and **TurnBy** all seem plausible, there are times when an action name should relate more directly to the specific characteristics of a device. For example, calling the actions of a button **MoveTo(bottom)** and **MoveTo(top)** seems clumsy at best. The situation is even worse for **Push**, since a force-sensing device might as easily detect **Pull**; **Stress** would avoid any confounding sense of direction, but seems unduly pejorative. **Impulse** implies transience rather than a change in force, and we have no experience that would help us name changes in relative torque. (For that matter, we have never encountered a relative torque transducer.) Regardless of nomenclature standards, though, it is a simple matter of lexical choice to pick a name for a particular action, and the name chosen has no effect on the rest of the representation.

As new devices are introduced, there may be several devices that share similar manipulation types and state components. To help prevent ambiguity, we suggest that the names of state components and manipulations be tagged with the name of the device. So, for example, a flat panel switch would have a state component *fpswitch:state* with a range of { *off on* } and some initial value (say, *off*); after the user action **fpswitch:Push(on)**, *fpswitch:state* would have the value *on*.

The CMR input device model fails to address one characteristic that is important for some devices. The primitive manipulations of the model do not specify whether a manipulation must change the state of a device. So, for example, the model says nothing about whether it is possible to **fpswitch:Push(on)** a flat panel switch that is already *on*.

As we discussed in Chapter 6, it is sometimes important to keep track of this sort of information; for devices in the existing UAN, we do so by means of viability conditions, expressed by assigning values to state components at start timestates. But since the CMR model does not make this information available, we do not try to create it arbitrarily.

So, given a CMR representation of a primitive input device being set to a particular value, we create a single-arc timetree as follows:



TT: { a, b }

SCV: { < *device-name:state*, **Out** > }

SCM: { < b, *device-name:state*, *value* > }

AV: { *device-name:act-name*(*value*) }

A: { < a, b, { *device-name:act-name*(*value*) } > }

ts_{init}: a

ts_{term}: { b }

Figure 7.3. Timetree for primitive manipulation.

The UAN representation associated with this timetree would be

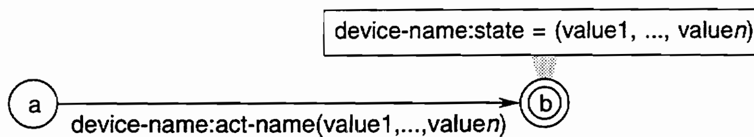
User Action	**Feedback**	**Interface State**
device-name:act-name(value)		device-name:state = value

Note that, as with other UAN device representations, it is possible to represent *value* at a higher level of abstraction. This is frequently more convenient than explicitly representing every possible value. It may also be better to pick a delimiter other than the colon for separating device names from action or component names, since the colon is currently used in conjunction with viability conditions.

7.3.2 Composite devices

As with primitive devices, the UAN view of composite devices eliminates much of the detail contained in composite definitions generated by the CMR model. In the cases of merge and connect composition, multiple devices are combined into one, and the UAN represents this single device like any other — by a single state value and activity. In layout composition, the UAN represents the actions and states of the composed devices separately. We reflect their relationship only by a naming convention.

To merge devices, we first pick a new name *device-name* for the merged device. We next create a new name for the manipulation that this device affords. In some cases, where we are combining devices with the same kind of manipulation, the activity name for the new device can be the same as the activity names for its components. For example, if we merge two one-dimensional sliders affording the activity **MoveTo** to form a two-dimensional locator, it makes sense to call the new activity **MoveTo(x,y)**. Finally, we define a state component for the new device, setting its range to the cross-product of the ranges of the component devices.



SCV: { < *device-name:state*, **Out1** × ... × **Out*n*** > }

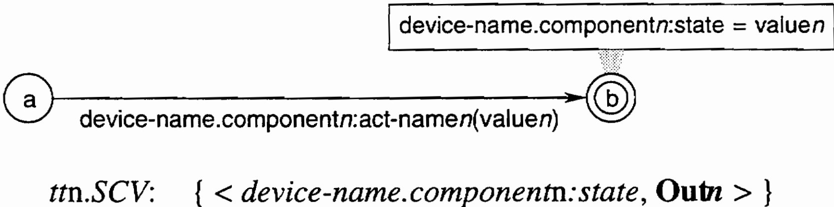
Figure 7.4. Timetree for merge composition.

Figure 7.4 shows the timetree resulting from this definition. Since all components of the timetree except the range of *device-name:state* are apparent from the diagram (as in Figure 7.3), we omit all but *SCV*. The UAN description associated with this timetree is

User Action	Interface State
device-name:act-name(value1, ..., value*n*)	device-name:state = (value1, ..., value*n*)

In layout composition, we retain the separate representations of each component device. The only indication that they are now part of a composite device comes from a new device name prepended to the existing device names. So, for instance, a mouse

composed from a two-dimensional locator and a button would be represented by two state components, *mouse.locator:state* and *mouse.button:state*; these devices would still afford their original actions under the new names **mouse.locator:MoveTo(x,y)** and **mouse.button:MoveTo(position)**.



ttn.SCV: { < *device-name.componentn:state*, **Outn** > }

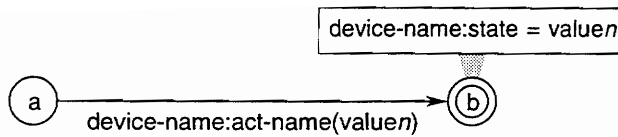
Figure 7.5. Timetree for component device in layout composition.

Figure 7.5 illustrates the form of the timetree for each component device. The UAN for component *n* is

User Action	Interface State
device-name.component*n*:act-name*n*(value*n*)	device-name.component*n*:state = value*n*

For connect composition, we create a new device and manipulation name, but retain only the last composed device's state information. Under the CMR notion of connect composition, the user manipulates a device (say, device1), and the output of device1 is coupled to the input of device2. In effect, the user manipulates device2 indirectly via device1.

According to UAN convention, the new manipulation name should relate to the *output* value of the *second* device, since that is usually what the user perceives. For example, while a conventional mouse locator transduces relative movement — the user moves the mouse a short distance, picks it up, moves it back, and repeats the process — the user perceives control over the *cursor* (or some other object), which has an absolute location on the screen. For this reason, it is sometimes acceptable to use the manipulation name from the second device, rather than creating a new one.



SCV: { < *device-name:state*, **Outn** > }

Figure 7.6. Timetree for merge composition.

Figure 7.6 illustrates the result of this definition, and the UAN for the new device's activity is

User Action	Interface State
device-name:act-name(value*n*)	device-name:state = value*n*

These transformations can all be applied to nested compositions, subject to the constraints of the original CMR model (for example, to connect-compose two devices, the output of the first must be compatible with the input of the second — one cannot connect a one-dimensional torque output to a three-dimensional absolute location input.)

As an example of converting a composite device definition, consider CMR's definition of a mouse.They define a mouse as the layout composition of a cursor locator and three buttons. The cursor locator connects the output of a two-dimensional locator with a screen cursor, and the two-dimensional locator is composed from two merged one-dimensional locators, each sensing relative position. We construct a UAN representation of the mouse piece by piece, as shown in Figure 7.7:

Device	Action	State component	State range
x locator	locator-x:MoveBy(x)	locator-x:state	real
y locator	locator-y:MoveBy(y)	locator-y:state	real
x-y locator	locator-xy:MoveBy(x,y)	locator-xy:state	(real, real)
cursor	cursor:MoveTo(x,y)	cursor:state	(screen-width, screen-height)
cursor-control	cursor-control: MoveTo(x,y)	cursor-control:state	(screen-width, screen-height)
buttons	button-*n*: MoveTo(position)	button-*n*:state	{ up, down }
mouse			
(cursor)	mouse.cursor-control: MoveTo(x,y)	mouse.cursor-control: state	(screen-width, screen-height)
(buttons)	mouse.button-*n*: MoveTo(position)	mouse.button-*n*: state	{ up, down }

Figure 7.7. UAN definitions of devices during successive composition.

The x locator, the y locator, the cursor, and the buttons are all primitive devices. The x locator and y locator are merge-composed to form the x-y locator. This x-y locator is connect-composed with the cursor to form the cursor-control device; the cursor-control device is then merged with the buttons to form the mouse.

The resulting action names and state component names are much more unwieldy than the standard UAN notation for a mouse: **~[x,y]** for cursor movement, **Mv** and **M^** for button actions (presumably **M*n*v** and **M*n*^** would specify multiple buttons). This reflects the UAN's tendency toward convenience over formality. Once new devices are introduced into the UAN, it is likely that designers will find shorthand notations that are more convenient than full, formal names; this is certainly not a problem.

8 Analytic Techniques

Timetrees provide a structure to represent the behavior associated with a human-computer interface. In previous chapters, we defined timetrees and some operations on them, we used timetrees and composition operators to provide a formal definition of the UAN, and we presented a mapping from a device model to the UAN via timetrees. In this chapter, we present some techniques for analyzing the behavior of interactive systems as represented by timetrees. Since it is possible to generate timetrees corresponding to behavioral descriptions, particularly UAN interface specifications, we can apply these techniques to designs as well as implemented systems.

As we stated in Section 2.3.3, analytic techniques that can be applied to behavioral descriptions have great potential value, since behavioral descriptions begin to take shape at a very early point in the development process. Problems that are discovered earlier during development often can be fixed at less expense, and with less danger of introducing design conflicts or inconsistencies.

Much of the material in this chapter focuses on the UAN, because it is central to our other research interests. We describe some UAN analytic techniques that do not explicitly refer to timetrees, but even in these cases, timetrees provide important formal support for such concepts as interface state, feedback, and user action. For this reason, we feel justified in presenting some of these techniques in addition to techniques that use timetrees more directly.

UAN interface descriptions provide a rich target for analysis. Since UAN specifications can capture the interface designer's concept of user behavior from very high to very low levels of abstraction, they are amenable to high-level global analysis and low-level performance prediction or local optimization. Since the UAN is used throughout the development process, analytic evaluation during early stages provides opportunities to prevent problems as they occur, rather than pointing them out after they have become part of the system. Finally, UAN analysis can detect task interactions related to common prefixes (Section 3.2) and feedback conflicts (Section 3.4); other behavioral representations either do not consider these interactions at all, disallow common prefixes

even when they do not cause problems, or require restructuring of designs to factor out common prefixes explicitly.

One of the advantages of the UAN is its expressiveness at both high and low levels of abstraction. At high levels, an entire system can be described in terms of just a few tasks. Each of these can in turn be broken down into a hierarchy of lower-level tasks, until at the lowest level behavior is expressed in turns of primitive actions — actions that cannot be interrupted or interleaved, and are not broken down further within the UAN. (This may seem like a circular qualification — "the UAN can represent low-level actions all the way down to the simplest actions that aren't broken down in the UAN." While the UAN can in principle be used at ever-lower levels of abstraction, we have found in practice that designers almost never want to break activities down further than the usual set of UAN primitives as defined in Section 6.6.7.)

Views of an interface description at different levels of abstraction refer to different kinds of user actions. At a very low level, there may be relatively few distinct actions — move the mouse, press or release a mouse button, press or release a keyboard key. A very slightly higher level might include key combinations, character sequences, and mouse motion with a button down or up. At these levels, distinct actions are determined largely by hardware and interface style, and their range may not vary widely among different designs on the same platform. We refer to this level of abstraction as the "articulatory level" [Hartson & Mayo, 1995].

Still higher levels might represent actions in terms of operations such as *enter-text*, *choose-from-palette*, *toggle-check-box*, and so forth. This is often a more natural level for describing the main functions of an interface; it is quite natural to break an interface description into a very low articulatory level (how to trigger a screen button, how to choose an item from a menu, etc.) and a higher level that assumes knowledge of those basic articulatory skills. The nature of these levels, the boundary between them, and possible other levels above or below them is an area of continued interest to us.

8.1 Global interface analysis

Given behavioral specification techniques capable of representing an entire interface, it is tempting to seek automated ways of characterizing or measuring the interface as a whole. Software engineering research has yielded various metrics for code size and complexity; could it be possible to develop similar metrics for *interface* size and complexity?

A number of research results exist in this area. For example, [Reisner, 1981] uses augmented BNF grammars to represent the linguistic structure of an interface; she measures interface size in terms of vocabulary size and string length, and complexity in terms of the number of productions needed to generate a string of actions. While these measures of size and complexity are easy to determine, it is not clear that they can be expected to predict performance or usability, because there is no obvious relationship between the structure of a BNF grammar for a language and the user's cognitive model of the system.

[Kieras & Polson, 1985] use a GOMS-based representation of the user and a transition network-based representation of the system, and briefly discuss measures of complexity. While this approach attempts to reflect the user's cognitive model of the system, the task structure to do so must be constructed in a separate step from the actual system design, leading to more work and increased chances of inconsistency between the task design and the system design.

More recently, [Byrne, Wood, Sukaviriya, Foley, & Kieras, 1994] present a system to generate GOMS-based representations from UIDE application models. While this reduces some of the work involved in generating a GOMS structure, UIDE models do not contain any information about higher-level tasks, and so higher-level goals and behavior still must be specified separately.

Even the GOMS-based approaches, which attempt to model higher-level user goals and behavior, do not deal well with changes in user behavior over time. For example, research by Rasmussen [Rasmussen, 1983] and others indicates that user mental processes change as the user's familiarity with the system increases; this can have a great

deal of influence on performance, but is not reflected in the Reisner or Kieras and Polson approaches.

While we do not claim that simple global statistics *do* provide interesting usability information, it is possible that various easily measured dimensions of an interface design *might* be correlated with usability. The UAN makes several such measures easily accessible. However, automatic generation of some of these statistics implies a machine-readable, symbol-table-like structure to represent the task hierarchy of a system. In the same way that a compiler can analyze a source program and generate lists of variables and subroutines, or class hierarchies, an automated UAN tool could generate lists of task names and actions, and hierarchies of subtasks. Such a tool is currently under development at Virginia Tech.

8.1.1 Number of tasks

The number of tasks and subtasks in an interface description says something about the complexity of that description, and possibly the complexity of the interface it describes. However, as we have already discussed, this complexity does not always reflect the user's view of the system. If the task structure of the interface description is a good match for the user's mental model, this measure may be of interest, but demonstrating such a match is difficult.

8.1.2 Number of user actions

Unlike tasks, the user actions specified in an interface description necessarily correspond to actions the user performs. It therefore seems plausible that the number of distinct user actions in a description might be correlated with usability, learnability, or cognitive complexity. It is also possible that frequency of a particular user action might be of interest. For example, it might point to problems with action overloading, which can be associated with inconsistency; if the same action is used to perform a number of different tasks, especially in similar contexts, it can lead to confusion (for instance, the overloading of disk ejection onto the drag-to-trash activity in the Macintosh Finder). This issue is confounded with task structure, though, and this might limit the utility of automatic measures.

This measure will also vary widely at different levels of abstraction. At the articulatory level, for example, there is usually a relatively small range of possible actions, and counting them will say more about the hardware platform or basic interaction style (for example, pop-up menus versus menu bars or button bars) than it will say about the larger interface design. A count of higher-level actions may say more about the complexity of a given application.

8.1.3 Number of entities

At higher levels of abstraction, user actions are typically specified in terms of logical entities in the interface — dialogue boxes, icons, controls, and so forth. A count of the number of such entities present in an interface, or simultaneously available at a given point during system execution, might indicate some aspects of system complexity. UAN specifications usually refer to these entities by name, often as part of a context specification (for example, ~**[OK-button]**); timetrees representing such specifications include groups of state components to capture the visible state of each item. A UAN tool should be able to collect names from such references throughout a task hierarchy and report statistics on the results.

8.1.4 Number of states and classes of state

Many interfaces present various *modes*, or states that associate different behaviors with particular actions. Most systems manifest some kind of internal state, and where that state affects system behavior, conventional wisdom maintains that the interface should make that state information visible. The UAN can represent both visible and invisible interface state, as well as state components not directly associated with the interface. Again, timetrees provide a formal representation of this information.

The number of states an interface can assume and the number of kinds of state present in the interface influence its complexity, although the relationship is by no means simple or straightforward. A tally of different kinds of state, and the number of values each can assume, could form one metric for interface size.

It may also be useful to examine how state components are used in a design. For example, if a state component controls the viability condition of a task, that state

component induces a mode — a task may or may not be available, based on the value of that state component. Again, conventional wisdom indicates that such a state component should have a visible indicator in the interface. In fact, UAN convention suggests using visible aspects of state to specify viability conditions, rather than having some visible item "follow" or "reflect" internal, hidden state. By examining the timetree representation of a task and its viability condition — for example, by checking the values of *item-visible* and *item-highlighted* for an item that must select before performing a task — it is possible to determine whether a task definition follows this convention.

8.2 Timing analysis

In Section 6.6.9, we discussed explicit representation of time in the UAN and in timetrees. Time can be represented both as a component of environmental state and as an environmental activity. We can use both representations in conjunction with UAN/timetree-based behavioral descriptions to explore timing-related issues. We examine two of these issues, user performance prediction and system response specification.

8.2.1 Performance prediction

There is a long tradition of techniques for performance prediction based on keystroke counting, mouse movement, and other articulatory details. Examples of this work include the keystroke model [Card & Moran, 1980] and action grammars [Reisner, 1981]; GOMS and models based on it can also incorporate information at the articulatory level. Since the UAN allows interface specification at this low level, UAN task descriptions can yield the sort of information needed for this kind of analysis, and the timetree model provides a framework for organizing and processing the information. Analysis of a task description can yield the number of keystrokes it requires; in conjunction with a concrete layout of visible entities, it can provide information about mouse movement as well.

For example, to count the number of keystrokes necessary to perform a task, one counts the number of pairs of key-down/key-up actions along a path from the root to a terminal node of a timetree describing that task. If paths to different terminal nodes yield different totals, it indicates that different ways of performing the task require different numbers of keystrokes; in such cases it would be possible to report the minimum or maximum over

all paths, or to obtain more involved statistical results (averages, weighted averages, distributions, and so forth).

Similarly, one can count instances along a path where a key-down/key-up action pair is followed by a mouse action and vice-versa. The number of such patterns is related to the amount of switching between mouse and keyboard during task performance, and such switching tends to slow performance (since homing on a keyboard or mouse takes time, but does not itself advance the performance of a task).

To add an explicit representation of this phenomenon to a timetree, we can insert a separate user action, *home-to-mouse* or *home-to-keyboard*, into each path where such a switch takes place. After empirically determining a representative duration *t* for each of these activities, we can add the environmental activity *interval(t)* to the appropriate activity arcs.

When we defined the semantics of activity arcs labeled with multiple activities, we stated that the ordering of such multiple activities is unspecified. If it is necessary to define the ordering of the activities more explicitly, the arc must be expanded into a timetree in which the details of ordering are made explicit. In Section 6.3, we presented an example of a specification in which two activities actually take place simultaneously.

The special nature of *interval*, marking the passage of time, calls for a different interpretation. Where an arc is labeled with multiple activities, and one of them is *interval*, we state by definition that the argument of *interval* represents the duration of the rest of the activities labeling the arc. So, if an activity arc is labeled with { *home-to-keyboard*, *interval(*1.5 s*)* }, it means that the user performs the activity *home-to-keyboard* and it takes 1.5 seconds.

Given this interpretation, one can take empirically determined or predicted times for user actions and add them to a timetree representation. Once this has been done for every activity (including system activities that require the user to wait), the sum of the intervals along a path yields the time required to perform the associated task.

Adding explicit timing information for mouse-driven travel among targets on a screen is more difficult, because a purely empirical approach to generate the timing information would require separate measurement for each possible pair of sources and destinations. To avoid this enormous task, one can use Fitt's law to *predict* timing instead of using empirical estimates, but this requires additional information that is typically abstracted out of timetree representations. Fitt's Law indicates that the time required to move to a target is proportional to the distance to the target and the inverse of the size of the target; that is, the farther it is to the target, or the smaller the target, the longer it will take to point to it. Given the size and distance of a target, it is possible to estimate the time needed to point to it.

UAN task descriptions typically use contexts of on-screen objects as targets for mouse movement. Usually, though, these contexts remain abstract; instead of defining the exact dimensions and location of an item, UAN descriptions simply refer to **[item]**. ([Chase, Casali, & Hartson, 1992] shows an extension to represent such information explicitly in the UAN.) To obtain information about size and location, it is necessary to refer to screen sketches or prototypes.

Once such sketches or prototypes exist, it is easy to incorporate the necessary information into a timetree. In Section 6.1.2, we represented the state of a display object called *item* with the state components *item-form*, *item-context*, *item-location*, *item-visible*, and *item-highlighted.* We can add new components, *item-size* and *item-location-abs*, to represent size and absolute location as measured from screen designs, and add *cursor-location-abs* to represent absolute cursor location. After each cursor-movement activity of the form **~[item]**, *cursor-location-abs* is updated to equal *item-location-abs.* (Of course, this is only an approximation, since *item-size* is finite and **~[item]** can leave the cursor anwhere within the context of **item**).

Given this information, we can calculate the time for cursor movement within a task by summing the time for cursor movement along a path of the timetree describing the task. (As with keystroke counting, tasks with different paths may yield a range of sums.) For each cursor-movement activity along a path, we can calculate the Euclidian distance between the values of *cursor-location-abs* at the activity's start and end timestates; using

that distance and *item-size*, we can calculate the predicted time to perform the cursor movement.

We can also add an explicit representation of total time to a timetree by defining an environmental state component named *time*. We propagate the value of *time* through the timetree via the following algorithm:

set-time(*state*, *curtime*):
    SCM := *SCM* ∪ { <*state*, *time*, *curtime*> };

    -- *for each arc leaving the current timestate, see if it is equivalent to an arc already encountered. If it is, condense it with the previous arc; if it is not, add it to* Arcset, *or if it is a virtual activity arc, add its end timestate to* Stateset.
    for each *tarc*: <*state*, *endstate*, *Label*> ∈ *A*
        if *interval(t)* ∈ *Label*
            then *set-time*(*endstate*, *curtime* + *t*)
            else *set-time*(*endstate*, *curtime*);

To label each node of the timetree with the total time taken by all the activities leading to that node, apply *set-time* to the root with an initial time value of zero: *set-time*(TS_{init}, 0).

Measures like these can provide some information about expected user performance. To make useful predictions for real-world tasks, though, it is necessary to consider user decision time and other cognitive factors. Some efforts have been made to represent these factors in a manner compatible with the UAN [Sharratt, 1990]. But, as discussed in the previous section, the task structure of a behavioral design does not necessarily reflect the structure of the user's mental processes, and so predictions based on such representations must be suspect.

8.2.2 System response specification

Predicting user performance on a task requires information about both user performance and system performance. In the previous section, we described how timetrees can be used both to represent and to predict some aspects of user performance. The explicit representation of time we presented can also be applied to system activities.

This application can actually carry information in two directions. First, where performance information for system activities already exists, this information can be incorporated into the sort of analysis described in the previous section. If system activities in a task description consist of "widget behaviors" or toolbox functions that already exist and have well-characterized behavior, it should be possible to obtain accurate timing information for them. Since this information is available early in the design process, it can also guide the design away from some performance problems — for example, it might indicate where the system cannot redisplay an object quickly enough to keep up with a dragging operation, or it might identify points at which to change the cursor or provide some other "please wait" indication.

Where system activities involve custom functionality, performance data may not be available until the functions are implemented, perhaps very late in the development process. In this situation, the missing information may make performance analysis impossible. But if there are known constraints on total task performance time, and it is possible to estimate user performance time, it may sometimes be possible to subtract the latter from the former and derive a ceiling on total system activity time. This can provide a real-time constraint on the implementation of system behavior. Since some user performance information can be derived early in the development process, these constraints can be recognized before implementation begins, and system functions can be designed from the beginning to meet their eventual performance requirements.

In this discussion of timing issues we have not considered the possibility of overlap between user and system actions. In practice, it is generally desirable to let the user type ahead or "mouse ahead" while the system is responding. The current UAN does not address this capability at all. While timetrees can certainly represent concurrency between user and system activities — Section 6.3 presented one example — we have not examined the area of buffered user actions. We discuss this further in Chapter 9.

8.3 Local optimization within tasks

In the previous section we discussed extracting low-level information about mouse movement for performance prediction. Some design problems may be reflected in patterns of mouse movement at this level. For example, a drawing program might have

objects that are distributed across a wide screen area, but can only be manipulated by relatively small "handles." Tasks in this system would frequently include cursor movement across a wide range to a relatively small target, and Fitt's Law indicates that for some values of "wide" and "relatively small" usability will suffer.

In some cases, though, simple examination of a UAN description may not point out a problem. For example, the MacDraw drawing program on the Macintosh provides a number of drawing tools invoked by a palette outside the drawing area. One tool allows the user to select an item or items; this selection tool is the default. To draw one graphic object, the user moves to the palette, clicks on a tool icon for that object type, moves back to the drawing area, and draws the object. Once the user draws the object, the system reverts to the selection tool. Thus, to draw several objects of the same type, the user must select the appropriate tool for each new object. The following table shows one possible UAN representation for such an interface fragment.

```
do_draw_tasks:

    ( select(objects) OR
        ( select_object_tool
          create_object_with_selected_tool ) ) *
```

If users often create a series of objects of the same type, this pattern results in a lot of extra mouse movement. But the UAN alone cannot tell us whether this is something the users will often do. More generally, a UAN specification can tell what patterns *can* occur, but cannot tell what patterns most often *do* occur.

Benchmark tasks provide one source for this kind of information. Developers commonly observe users performing "sample tasks" to evaluate usability. Expanding these sample tasks according to the interface's UAN specification can yield sequences of activities corresponding to the developers' concept of "realistic" system use. Through the techniques of Section 8.2, it is possible to analyze keystroke and mouse behavior in these sequences.

In fact, expanding sample tasks through a UAN/timetree description can yield data in the same format as user transcripts — sequences of user actions during the performance of a task. [Siochi and Hix, 1991] discuss the automated analysis of such transcripts. The

basic feature of this work is the detection of repeating patterns of user activity (*maximal repeating patterns*, or *MRP*s), which is precisely the feature of interest in the example above.

Expanding a benchmark task involving repeated object drawing through the UAN of the example above would yield many action sequences in which the user selects a particular tool and then creates an object with that tool. MRP analysis of such an expansion would reveal that this is a frequent pattern, indicating that it might be a good place for improvement; applying Fitt's Law to the mouse movements back and forth between the palette and the drawing area should reinforce this idea. (In fact, MacDraw II adds a feature to address this need: double-clicking on an icon in the tool palette puts the program into a mode in which that tool remains active until another is selected.)

This approach cannot replace actual user testing, but it can provide some of the information user testing would yield, and it has the potential to be much faster and less expensive. It affords the additional advantage that it can take place before the interface has been implemented or even prototyped, as well as later in the development process. We next consider some other examples of analysis early in the development process.

8.4 Applications in early stages of development

In any system development effort, later stages depend on early decisions. Changes in one development area that affect another will increase the effort involved in both parts — for example, late changes to an interface design will require additional implementation effort to rewrite subsystems that had already been implemented, wasting the effort that went into the original implementation. Obviously, it is helpful to detect problems as early as possible in the development process. A number of these problems that can be detected analytically.

8.4.1 Undefined and unused tasks

Given an incomplete task-based design, the problem of finding undefined tasks or tasks that are not referenced is trivial — a simple problem of symbol table management within an automated UAN tool like the one mentioned in Section 8.1. In this case, the symbol table entry for a task would contain its name, a reference to its definition, a list of its

subtasks, and either a list of parent tasks or a reference count. A null reference for the definition would indicate an undefined task; a null list of parent tasks, or a zero reference count, would indicate an unconnected component. (This scenario disallows the existence of recursive task definitions. It is still unclear whether recursion should be allowed in the UAN.)

One area of UAN research at Virginia Tech involves graphical representations of UAN designs, which also presupposes the existence of a task-hierarchy data structure. A graphical presentation of the structure of an interface design could provide an exceptionally clear and usable indication of missing or unconnected components.

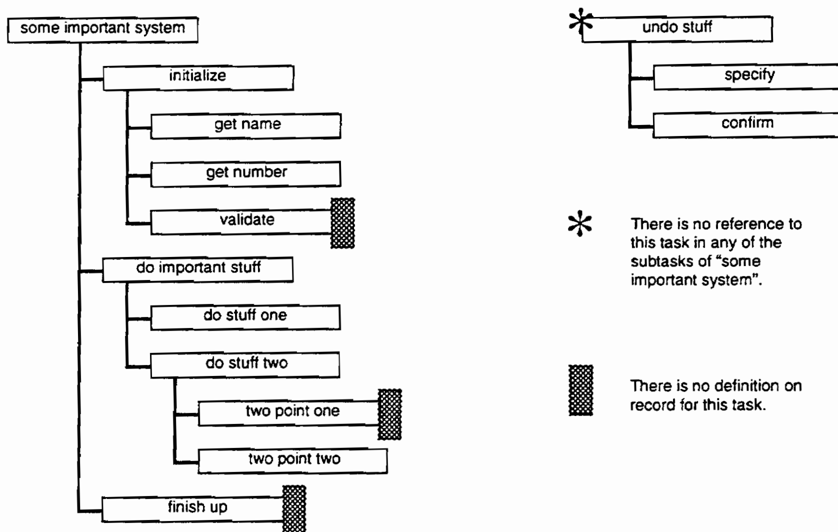


Figure 8.1. A possible graphical representation of an interface design with unconnected and undefined components flagged.

8.4.2 Redundant tasks

When different parts of a system are being developed by separate individuals or groups, duplication of effort is always possible. In task-oriented behavioral design, there may be some tasks that appear at many points within a system. If the developers working on these different parts are truly unaware that they have some subtasks in common, automated analysis alone probably will not be able to solve the problem; different developers might come up with different names, different task structures, and so on. But the notion of a "common subtask" implies some kind of similarity, perhaps in

preconditions, effects on state, or user actions required. All this information should be available in the behavioral specification, and the timetree-based formal definitions of state, viability conditions, and actions represent this information in a form amenable to automatic analysis.

A tool with access to the entire specification should be able to look for tasks that share some of these characteristics. Empirical investigation will be necessary to find out what kinds of similarity are the best indicators of redundancy.

8.4.3 Task design inconsistencies

One form of design inconsistency seems unique to task-based descriptions. It is related to the concept of *interface consistency*, and in fact efforts to generate a consistent interface can actually make this design inconsistency more likely.

[Hix & Hartson, 1993] characterize interface consistency as "the principle of least astonishment":

> If something is done a certain way in an interface (e.g., a task has a specific name, or an icon has a certain appearance[look] or behavior [feel]), users expect the same thing to be done the same way throughout the rest of the interface. Thus, similar things are expected to be done in similar ways. [p. 34]

[Tognazzini, 1992] presents guidelines reflecting a two-sided view of consistency:

- Keep the system's behavior consistent. The same class of object should generate the same type of feedback and resulting behavior, no matter in what part of the program or release of the software they appear.
- Interpret user behavior consistently. Consistent interpretation of user behavior by the system is even more important than consistent system behavior. [p. 139]

Consistency in interpreting user behavior can be elusive where the repertoire of possible behaviors is limited. For example, in an interface design based on windows, icons, and pointing devices, clicking and dragging are the most common actions used to manipulate graphic elements. As a result, multiple functions are overloaded onto similar or identical

user actions. The Macintosh Finder presents one example of this effect: dragging a file icon onto a folder icon causes the file to be copied if the file and folder are on different volumes, but the file is instead only moved if it and the folder are on the same volume. The user cannot tell which volume contains a file or a folder just by looking at their icons, so the result of this dragging action is sometimes surprising.

The problem of overloading a limited set of user actions can become even more acute when separate designers are working on the same part of an interface. The goal of "doing similar things in similar ways" can lead separate designers to develop a set of interface descriptions using the same user behavior to invoke different (but "similar") system functions. This kind of inconsistency is *not* an interface inconsistency, because it is impossible to build an interface that meets this specification. By comparing sequences of user actions among timetrees representing the various task descriptions, it is possible to detect this kind of inconsistency in a UAN specification before the design is passed along to implementors.

A more subtle form of conflict arises as a result of the common-prefix phenomenon in task-based representations. It is perfectly acceptable to present the user with a choice of tasks, some of which start with the same sequence of actions, as long as the tasks eventually diverge to require distinct sequences of user actions. The timetrees of Figures 3.9 and 3.10 illustrate how this divergence takes place among the Select File, Move Icon, and Delete File tasks from the Macintosh Finder.

Conflicts can arise in at least two ways when separate tasks share a common prefix. First, as discussed in Section 3.4, two tasks can specify different system responses to the same user action. This is sometimes precisely the intent of the designer, as in the copy-vs.-move example above. But if no aspect of system or interface state allows the system to choose among the possible alternative behaviors — if, for example, the two tasks are distinguished only by *future* user actions, as in the example of Figures 3.11 and 3.12 — it is impossible to implement an interface satisfying the design.

The second form of conflict arises when two task descriptions specify conflicting values for the same state component within a common prefix. That is, two simultaneously viable tasks may start with the same constellation of state values and the same user

action, but specify different values for one state component as a result of this action. In such a situation, there is no way to determine which value the implementation should assign to the state value in response to the user action; the specification is inherently ambiguous.

In this second circumstance, the definitions of Chapter 6 make it possible to detect conflicts in UAN descriptions. When the tasks in question are composed and the resulting timetree disambiguated, the *replace* algorithm used to condense timestates of redundant arcs will report failure. Conflicts of the first class do not cause failure during composition; instead, they yield a timetree in which multiple system activities branch from a single timestate. While it is easy to scan a timetree for this kind of structure, there are philosophical problems with calling all such occurrences "errors"; we discuss this issue further in Chapter 9.

8.4.4 Coverage of possible action sequences

One class of questions about a design is "what happens if the user does X?" Examination of a behavioral specification can reveal sequences of user actions that lead to undefined behavior — sequences that do not match any part of the specification. But any real system must do *something* in response to such a sequence, whether it indicates an error, ignores the actions, or crashes. Thus, UAN interface descriptions frequently are "incomplete" in the sense that there are sequences of user activity that do not match any part of the task description. This reflects a wider issue pertaining to the scope of the UAN, and we discuss it further in Chapter 9.

In addition, perturbing intended sequences of user actions — for example, "what if the user clicks just off the icon's edge" — may indicate points at which it is too easy for a user to make a serious error. Of course, to identify "serious errors," we must first define the term. Reasonable definitions might include events that make it impossible to achieve the intended task, events that cause persistent nonobvious side effects, or events that cause a change in the system's visible state large enough to disrupt the user's attention to task. This area alone presents a number of research opportunities.

For example, in many Macintosh applications the user can select multiple items by holding down the Shift key while clicking each item. Shift-clicking an unselected item adds it to the selected set; shift-clicking a selected item removes it from the set. In many cases, one can drag the entire selected group by dragging one item within the selection. But if the user accidentally moves just outside the context of an item before starting the drag, all the selected items are deselected. There is no way to undo this deselection; Macintosh interface guidelines specify that, since selection itself does not produce persistent changes, it does not count as an "action" for the purpose of undo/redo management. But multiple selection can take a significant amount of time and cognitive effort, and a single slip can easily undo that work.

Conversely, mousing down outside any object and dragging across the background frequently produces a "selection marquee," allowing the selection of all items within the resulting rectangle. If the user accidentally "misses the background" and starts the drag on an object, the marquee does not appear; instead, the object moves, following the cursor.

In this particular example, examination of the tasks available in the application would reveal that dragging a selected item and dragging across the background produce significantly different effects. It is not immediately clear how the standard behaviors for movement, selection, and dragging could be modified to improve this situation, but it seems it would be valuable to detect it at an early stage of design.

Composition of the timetrees representing tasks in a design can in principle generate a combined representation of all the tasks viable at a particular point in an interface; such a representation presents all the possible actions at a particular point in the design. (In this context, a "point" is a point within a task, corresponding to a particular timestate.) The example of Section 3.2, showing the composition of three small tasks, illustrates this process. By examining such a timetree, it is possible to see which action sequences have been defined across entire higher-level tasks, and to find sequences for which no ensuing system behavior is defined. At the simplest level, one could make a list of the activities labeling the arcs that leave a given timestate; any user action whose viability conditions are met at this timestate, but does *not* appear on one of the arcs, represents a user behavior that is not covered by the specification. This approach potentially can exercise

parts of a design much more quickly than user testing of a prototype, not only because it is subject to automation, but because it does not require repetition of a sequence of activities that leads to the point in question.

Of course, the number of possible user actions can grow very large, and it would be unwieldy to specify them all explicitly in a behavioral specification. In general, task-oriented representations do not explicitly list all possible user actions at each point in a design. Many possible behaviors are either implicit or defined outside the current task description. The current UAN provides little support for implicit or default definitions; adding such support is an area of great interest to us.

8.4.5 Walkthrough and prototype support

While analysis of a design can reveal important information, no analytic procedure can *replace* user testing. But like other forms of testing and verification, user testing early in the development process can reduce overall effort. Timetrees can make it easier to answer certain questions that arise during preliminary user testing.

During formative evaluation of interface designs, design walkthroughs can give users a "feel" for the behavior and appearance of a system even before the creation of a prototype. During these evaluations, users frequently ask questions of the form "what if I do this?" As we discussed in the previous section, timetrees can help answer this question by collecting all the actions that are viable at a given point in a design. Particularly during early stages of task analysis and design, most tasks will be defined only at a relatively high level, omitting detailed actions and special cases; in this situation, timetrees describing the task structure will also be smaller and more tractable for large design subsets.

Once implementation of a prototype begins, the implementors will need information about task interactions. While timetree-based representations remove the burden of common prefix detection and resolution from the dialogue designer, an implementation (even a prototype implementation) sees only a sequence of user actions, and it must operate from an unambiguous representation of possible action sequences and their meanings. Through disambiguation following task composition, timetrees provide such a

representation. This application of timetrees will also be important in efforts to automate translation from behavioral to constructional representations, which we discuss in Chapter 9.

8.5 Task information at run time

In some applications it is valuable to know the user's current task. At one level, this seems trivial — without some level of "understanding" between the user and the system, nothing predictable can happen — but many current systems are designed around objects, actions, and documents, with *no* run-time representation of task information.

With the advent of "intelligent assistants" and other automated agents, this information becomes necessary. For example, an agent may watch what a user is doing and either construct a behavior profile or offer assistance. If the agent does not know all the possible tasks that a user could be performing with a given set of activities, it may offer confusing advice. Timetrees make it possible to associate a list of viable tasks with each node, and thus with each point in a sequence of user actions. In a system designed from the beginning in terms of user tasks, this information can be incorporated into the finished system to facilitate intelligent assistance.

To task-label a timetree, one may either assign a task label to each terminal timestate and then propagate each label back to the root, or label each node in the various timetrees before composition and preserve those labels (as opposed to generating state conflicts). This yields a timetree in which each timestate is labeled with all the current viable tasks.

As an illustration of this task-labeling, consider the Manpulate File task of Section 3.2. This task allows a choice among three subtasks: Select File, Move Icon, and Delete File. All three tasks share a common prefix. The timetree has three terminal timestates, one for each subtask; upon reaching a terminal timestate, there is no ambiguity about which task the user has performed. Figure 3.9 shows the timetree for Manipulate File with its terminal timestates labeled. To task-label this timetree, we assign a task label to each terminal timestate and then propagate each label back to the root. This yields a timetree in which each timestate is labeled with all the current viable tasks, shown in Figure 8.2.

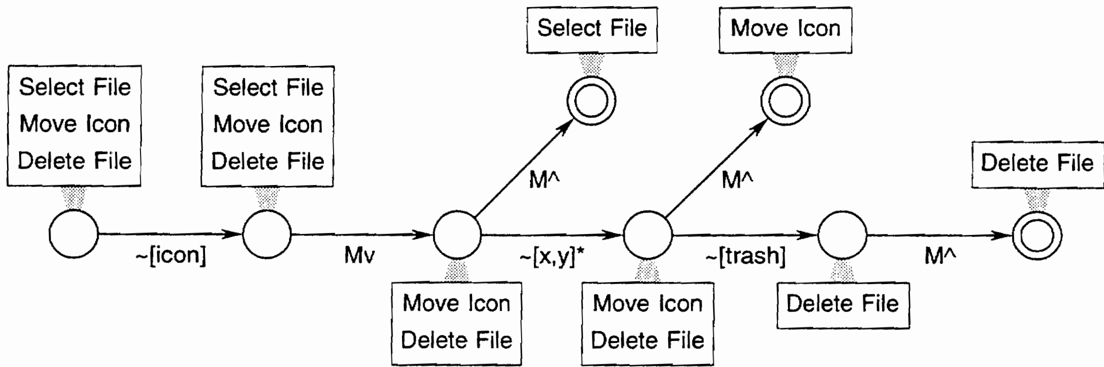


Figure 8.2. A task-labeled timetree.

In a larger system design, it will probably be desirable to augment this kind of task-labeling with information about higher-level tasks as well. In an application that is task-aware, it should be possible to provide assistance or automation based on higher-level user goals as well as immediate, low-level procedures. There are a number of issues involved in carrying this knowledge across from a behavioral, task-oriented design to a constructional implementation; we touch on some of these issues in our discussion of future work, the next chapter.

9 Future Work

The timetree model meets a number of important goals. It provides a set of structures and operators for representing interactive behavior. These structures and operators serve to specify a formal definition for the UAN, they make it possible to connect the UAN to an input device model, and they support a number of techniques for analyzing behavioral specifications.

This model has also succeeded in generating a wide range of new questions and directions for investigation. In this chapter, we briefly outline some of these areas, classifying them into four broad categories: translation between behavioral and constructional representations, philosophical questions about the model itself and the phenomena it represents, desirable extensions to the model, and omissions and difficulties in the UAN that the model has revealed or clarified.

9.1 Translation to constructional representations

Since the UAN first came into existence, automatic generation of interactive systems from UAN descriptions has been an attractive goal. As the UAN matured, our understanding of the difference between the behavioral and constructional realms improved, and it became clear that a "UAN compiler" would be a nontrivial project.

The goal of translation from behavioral specifications to constructional implementations was one of the motivating factors in the development of the timetree model. Initially, we hoped that timetrees themselves might be more or less directly translated into an executable form, much as Jacob's specification diagrams [Jacob, 1986], [Jacob, 1985] could be interpreted by a run-time system. Ironically, it was a similarity between task-based descriptions and the human-computer dialogues of the 1960's that made this difficult.

In older systems where the flow of control through a program dictated the form of the user dialogue, translation between a series of subroutine calls and a series of prompts, inputs, and responses was relatively straightforward. In contemporary systems, patterns

of execution are driven by user choice, and implementations are usually event-driven and asynchronous. Much of the information that was previously hard-coded into a program's procedural flow of control is now represented by state information in program objects.

Task-oriented descriptions, though, still describe *paths through* a system. This information is critically important, because it captures the way people actually use systems; with a goal in mind, they follow a sequence of steps to perform a task, even if the computer follows the steps they choose rather than marching them through a predetermined path. But the gap between this representation and the objects and protocols that constitute modern interactive programs is vast.

We believe that bridging this gap will require tools and techniques that can transform task-based sequence and selection into the states and events of an asynchronous, event-driven program. Since timetrees explicitly model both task structure and state information, we still believe that they can provide the foundation necessary for this translation process. One approach involves explicit translation of task-sequencing information into synthetic state components that can control asynchronous objects, like those of UIDE [Foley, Kim, Kovacevic, & Murray, 1989]. Another approach involves describing the behavior of *constructional* entities in terms of timetrees, and deriving implementations such that the the timetrees describing the behavior of the specification and the implementation are identical. We continue to investigate this area.

Timetrees have already made important contributions to the existing process of manual translation. Most importantly, they have provided a formal definition for the UAN, strengthening a number of areas where the UAN previously lacked rigor or omitted detail.

The explicit representation and resolution of common prefixes provided by timetrees is also important. Approaches like Jacob's state-based executable specifications require the designer to factor out prefixes explicitly; this is a distraction that is not necessary with the UAN. The designer can focus on each task independently, and timetree-based task composition automatically generates a tree to handle the prefixes, or catch conflicts if they exist.

9.2 Philosophical questions

There are at least two outstanding questions about the current timetree model. One regards whether the model is closed under certain operations, that is, whether some manipulations can yield a result that is not a timetree. The other involves the meaning or interpretation of certain timetree structures.

9.2.1 Infinite branching

As we indicated in Section 6.5.2, it is possible to define an infinite timetree which the *collapse-VAs* algorithm will reduce to an infinitely branching timetree of finite depth. Consider a timetree of the form illustrated in Figure 9.1.

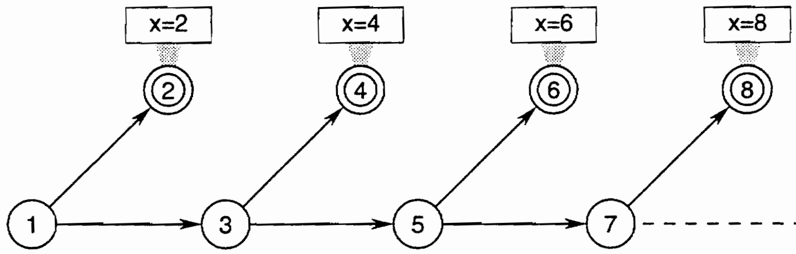


Figure 9.1. Infinite timetree with infinite number of different state values.

In this timetree, all activities are virtual, timestates are labeled with the natural numbers, and every even-numbered timestate has a single state component value assignment, each with a different value. We can define such a timetree formally as follows:

TT:	**N**
SCV:	{ < x, even positive integers > }
SCM:	{ < n, x, n >: n $\in$ even positive integers }
AV:	{ }
A:	{ < n, m, $\varnothing$ >: n $\in$ odd positive integers, m = n + 2 } $\cup$ { < n, m, $\varnothing$ >: n $\in$ odd positive integers, m = n + 1 }
ts_{init}:	1
ts_{term}:	{ n: n $\in$ even positive integers }

collapse-VAs attempts to collapse all the odd-labeled timestates onto the initial timestate, while keeping the even-labeled timestates separate (since they contain conflicting state assignments). Actually, since *collapse-VAs* executes in a depth-first fashion, it does not terminate for an infinite timetree; instead, we must apply it to a depth-limited timetree, then observe the limit as the depth of the source timetree approaches infinity. In this case, the limit yields the following tree:

TT:	{ 1 } ∪ even positive integers
SCV:	{ < x, even positive integers > }
SCM:	{ < n, x, n >: n ∈ even positive integers }
AV:	{ }
A:	{ < 1, m, ∅ >: m ∈ even positive integers }
ts_{init}:	1
ts_{term}:	{ n: n ∈ even positive integers }

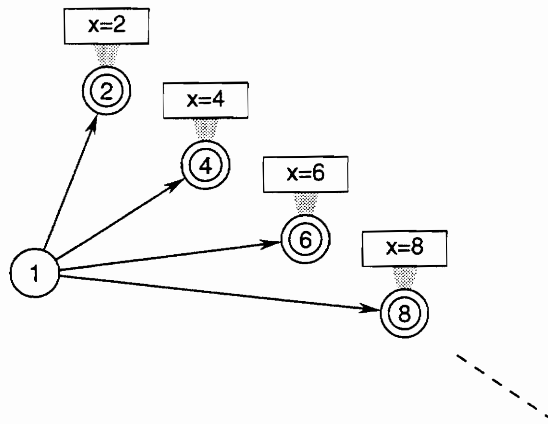


Figure 9.2. Infinitely branching tree generated from timetree of Figure 9.1.

As Figure 9.2 illustrates, the resulting tree is of depth 1, but is infinitely branching. Since the definition of timetrees specifies finite branching, this structure cannot properly be called a "timetree."

We exclude the possibility of infinitely branching timetrees to ensure that the algorithms for composing, disambiguating, abstracting, and analyzing timetrees remain well-defined. Timetrees of infinite *depth* are not uncommon — closure, for example, nearly guarantees

paths of infinite depth — but, since timestates along a path are ordered, depth-limiting allows us to express operations over these trees in terms of limits. There is no satisfactory way to implement "breadth-limiting" without imposing some arbitrary ordering on the inherently unordered paths leaving a single timestate.

As we originally stated, we have not yet encountered a situation in which the composition operators we have defined generate a timetree of this form from a finite specification. Nonetheless, we intend to examine this phenomenon further to determine its significance.

9.2.2 User choice and "system choice"

When several arcs leave a single timestate, it represents a branching of possible paths — in any given path through the specification or system, one path will be followed. In our current interpretation of timetree structures, if several arcs labeled with user actions leave a timestate, it reflects user choice. If several arcs labeled with system actions leave a timestate, it reflects an error in specification. User choice is acceptable, but "system choice" — indeterminate selection of an action by the system — is not. This reflects our user-centered view of the interface.

There are certainly times when systems exhibit indeterminate behavior; sometimes they do so by design. It may be desirable to allow explicit representation of this indeterminacy, just as we have always explicitly represented user choice. At present, timetree composition algorithms allow both kinds of choice. It seems wise to investigate this issue further, instead of high-handedly resolving it by definition.

9.3 Model extensions

We have identified several areas in which extensions to the timetree model may be useful. We first point out some areas of exploration that may clarify which extensions are most valuable; we then consider some individual extensions.

9.3.1 Examination of new interface styles in relation to timetrees

Our work so far has concentrated on the UAN, and most of the UAN specifications we have read and written deal with menu-based or direct-manipulation interfaces. While we

have little experience with newer, more highly interactive interface styles, such as gestural interfaces using a pen or glove, we feel confident that any new interface style will still lend itself to task-oriented development. As long as tasks are useful in describing interfaces, timetrees will be of value in representing those descriptions.

We are particularly interested in gestural interfaces, because it seems reasonable that common prefixes and ambiguity might be more prevalent in such systems.

9.3.2 Extension to multiple users and multiple systems

The traditional formulation of the UAN assumes a single user, a keyboard, a mouse, and a screen. We have shown how to add new devices to the UAN, but we have not addressed the possibility of multiple users and multiple systems.

In principle, timetrees can represent multiple users and systems without difficulty. Most timetree definitions take no note of whether an arc is labeled with user, system, or environmental activity. Where we currently support functions to classify activities and state components into these three categories, we can certainly support additional classifications for multiple users and systems.

Unfortunately, the existence of multiple users and systems implies that concurrency becomes the rule rather than the exception, and timetrees handle general concurrency with some difficulty. To make timetrees useful in this realm, we must find improved ways to deal with concurrency.

9.3.3 More general treatment of concurrency

We have examined various approaches to representing concurrent behavior. Our current approach, breaking activities into initiation, continuation, and termination, suffices for many aspects of concurrency that are common in direct-manipulation interfaces.

It is possible to push concurrency out of timetrees by representing each interacting party with its own timetree and depending on state components for control, or by defining timetrees in which each party interacts with a central, perhaps environmental,

"controller." The first approach can hardly be called a "treatment," and certainly not a "solution"; the second, too, seems only to beg the question.

We are beginning to examine the possibility of incorporating some of the features of Petri nets into an augmented timetree model. Petri nets represent states and transitions explicitly, as do timetrees, but they also allow multiple loci of control. In the current timetree model, the notion of "current state" is mostly implicit; rather than conceiving a system and user jointly hopping from timestate to timestate, we generally think in terms of paths. Adding Petri-like "markings" would enormously increase the complexity of the model, and we have little idea how well it will work.

It is also possible that our current treatment of concurrency is actually the most appropriate one. While Petri nets and other concurrent representations can be used to describe concurrent systems, the possible executions of such systems can still be described by timetrees, using the initiation-continuation-termination view we describe. It may be that new techniques for combining the timetrees of multiple concurrent interactions, in combination with new abstraction operators, will yield an acceptable timetree-based representation of generalized concurrent behavior.

9.3.4 Ranges in state values

Many specifications refer to ranges for state values. This is particularly true for viability conditions. In the current timetree model, state components take on only single values. It is possible to define a state component whose range is the powerset of some other range, allowing that state component to represent any subset of the range; but this is unwieldy for most ranges, and useless in dealing with continuous variables.

If state components can have ranges as values, modifications become a problem — if X is greater than zero at the current timestate, will it be greater than zero after an action that decrements it by one? There are probably other ramifications as well, and so further investigation of this area seems necessary.

9.3.5 Relations among activities and state values

Timetrees themselves make no connection between state component values and activities. State components are defined over timetrees, and each component may have values at any or all timestates. In our definition of the UAN, we adopted conventions that associated state values with activities to represent viability conditions and effects on state, and modified composition algorithms to preserve this information.

There are some situations in which it may be convenient to establish a more direct link between activities and state values. For instance, a number of interface specification systems allow feedback to be defined in terms of constraints — for example, one can specify that a square follows the cursor, while two other squares stay at least an inch away from the first square and each other, and a run-time constraint system will repetitively update the positions of the objects to present the appearance of continuous, constrained motion. The current timetree model can represent such behavior only by breaking it down into its repeated steps. The existence of constraint-based systems demonstrates that these interfaces can be designed *without* forcing the designer into this low-level view.

At a more fundamental level, the timetree model does not provide any explicit representation of causality. By necessity, causes must precede their effects, and so an action that causes another action or sets a state component's value must occur along that action's or timestate's history. But precedence does not imply causality, and there is no way to specify whether one event precedes another by coincidence or by necessity.

Given such knowledge of causality, it is possible to analyze a timetree and *verify* whether causal constraints are satisfied. It may be most useful to develop a separate representation of causality to be used in conjunction with timetrees, rather than incorporating such a representation into the model.

9.3.6 Probable/unlikely choices

Where multiple user activity arcs leave a single timestate, a timetree specifies user choice among the actions. In some situations, it may be useful to represent some knowledge about which activity is most likely or occurs most frequently, based on frequency counts

or other empirical data. This sort of labeling is available in some representations based on GOMS, where it can contribute to performance prediction.

It may be preferable to represent this sort of information without adding to the timetree model. Instead of associating a probability with an activity arc, we can define a state component to represent it, and assign values to this state component at the end timestate of each user activity arc leaving a timestate. This makes it easy to find probability information given a reference to the arc, without requiring an additional arc-labeling component.

9.4 UAN extensions and problems

As we have developed and refined formal definitions for the components of the UAN, we have found several opportunities for improving the UAN's scope and capabilities. These areas suggest some changes that will make it possible to write more complete system specifications, and some that will make it possible to describe behaviors that currently cannot be expressed in the UAN.

9.4.1 Context issues

In Section 6.2.2, we presented definitions for UAN cursor-movement actions in terms of contexts. As we pointed out, current UAN practice frequently omits context-leaving movements; instead, they are implicit in other movement actions. For example, **~X** implies not only moving into the context of **X**, but *leaving any previous* context.

This becomes a problem when contexts can overlap. In such cases, it is possible to enter a new context before leaving an old one. It is also possible to enter or leave multiple contexts simultaneously. A definition that represented context-leaving as part of context-entering would make it impossible to specify these activities.

We have considered this issue at length, and have developed UAN notation for some of these cases. To represent multiple simultaneous contexts, it is necessary to alter the Section 6.2.2 definition of *cursor-location*. Where *cursor-location* now takes on either two-dimensional location values or single context values, it must be able to take on multiple simultaneous context values — in other words, it must represent a *set* of

contexts. This also raises an issue of "allowed values," since *cursor-location* can only take on a particular set-value if the contexts in the set overlap at some point; the physical layout of the contexts will control what sets, and thus what cursor-movement activities, are valid in any particular case.

9.4.2 Completeness of UAN specifications

In informal discussions, a number of practitioners and researchers have asked about "completeness" of UAN specifications. It has been our view that *complete* specification of a system's response to *any possible* user behavior is too difficult — that it requires too much detail, much of it uninteresting. Many aspects of behavior should rightly be abstracted out of a useful task-based description. In fact, some believe that task-based representations can *only* describe certain intended paths through a system, not a system in its entirety.

To the extent that this is true, it presents a major problem. Specifically, in order to implement a robust and reliable system, *someone* must define its response to *every possible* sequence of user activities. The less closely this goal is met, the more likely it is that the system will reward unexpected user behavior with untoward consequences.

At the lowest level, where individual parts of an application interface must deal with any conceivable user action, default responses are probably the most important tool. A rule as simple as "if you don't know what to do with a user action, ignore it" or "if the user hits a key for which no action is defined, beep" can ensure that every user action has a defined result, although it certainly does not ensure that that result is always correct. So far, the UAN does not provide even this level of default support. Usually, such checking is left to the implementor, and implementors can be almost as resourceful as users in making systems do surprising things.

In some circumstances, it may be useful to attach an "otherwise" clause to choice compositions. This might be represented as an "otherwise" branch in a timetree, although this would introduce difficulties in composition and disambiguation; alternatively, the timetree representation of "otherwise" might add branches to a user choice node to handle actions for which the design did not specify a response — in other words, to guarantee

that every possible user action is explicitly represented in the specification. From this most concrete and detailed representation, it will be important to develop more tractable abstractions. Finding more tractable abstractions for timetrees should inform our search for them in the UAN, and vice versa.

The idea of completeness becomes fuzzy at higher levels. At the very highest level, most designs certainly are not expected to explicate every high-level task a user could ever perform with the system. Indeed, the discovery of new tasks, new uses for the "artifacts" of an interface, can be an important part of iterative system refinement [Carroll, Kellogg, & Rosson, 1991]. As users become familiar with the functions a system affords, they find new ways to use these functions. This is a virtue, or at least not a fault.

9.4.3 Arenas and excursions

As one approach for bringing order and modularity to UAN descriptions, we have considered the notion of *arenas* — isolated areas of focus within an interface. Actions associated with one portion of an interface take place within an appropriate arena; actions outside the arena have to do with other parts of the interface, they are described elsewhere in the specification, and they add no complication to the description of the task at hand. Indeed, since actions have different meanings depending on the arena in which they occur, this can be viewed as another approach to defining and managing *modes* within an interface.

Consider an application window on a Microsoft Windows desktop. (We choose Windows for this example to avoid considering a global menu bar.) Tasks associated with the application will involve actions within the window. Actions outside the window — moving outside and clicking, for example — constitute *excursions*; they are a distraction or departure from the task at hand, and it is possible to resume the task at hand as though they never happened. Since excursions usually have nothing to do with the task at hand, they arguably should not be mentioned in that task's dialogue description.

It appears that some excursions should be described in a separate, special area of a dialogue specification. For example, actions to switch between applications in a windowing system are idiosyncratic to particular environments. In some environments,

like the Macintosh or Microsoft Windows, one "activates" a window — indicates that it is the current arena, or focus of the user's attention — by clicking on it. To emphasize this shift in focus, the system brings the window to the foreground. In other environments, just moving the cursor over a window activates it, often *without* bringing it to the foreground (sometimes with puzzling consequences). The existence of environment-wide standards for task switching suggests that excursions associated with this process should be described in some separate specification, to be shared among all applications designed for that environment.

On the other hand, these standards are not always so universal. On the Macintosh, a click inside an inactive window usually just activates that window and brings it to the front. Sometimes, though, it activates the window, brings it to the front, and then causes something to happen. (In the Finder, for example, clicking on an icon in a background window selects that icon, as well as bringing the window to the front.) Interface designers and implementors can choose between these behaviors not only on an application-by-application basis, but on a window-by-window basis within a *single* application. This seems to argue for excursion definitions associated with individual parts of a system design.

Worse yet, there are other situations in which a single action spans arenas. Mouse dragging, for example, often starts in one context and ends in another, and sometimes the destination context is not in the same arena as the source context. It is not clear how arenas can restrict the scope of a task description while still allowing a designer to specify this sort of activity.

9.4.4 Interruptability and task abandonment

The concept of excursions is closely related to the concept of task interruption. The UAN provides facilities for specifying that one task can interrupt another, for indicating points at which a task can be interrupted, and for delimiting sequences of activity that may not be interrupted. In Section 6.6.7, we provided a formal definition of UAN interleavability and interruptibility, clarifying some points that previous descriptions of these phenomena had left ambiguous.

Even with these clarifications, though, the UAN's treatment of interleaving and interruption remains incomplete. Section 6.6.7 describes several types of behavior that designers have tried to express in the past, but are not supported in the current UAN.

More importantly, the current UAN provides very little support for specifying that tasks can be *abandoned* — that an excursion may never return, or that a specific action makes the rest of the task unavailable. This kind of behavior is pervasive in most interfaces, and it is absolutely critical to provide notational and definitional support for it. As the current interpretation of the UAN stands, if a user performs a sequence of actions that match a certain task, but never finishes the task, it is "as though the user never performed the task at all," at least for the purposes of specification. But in reality, systems respond to user actions, state components are assigned values based on those actions, and most importantly, the user is *aware* of having performed those actions. The UAN must provide some way to say that tasks might not be completed, and to specify what happens when they are not.

9.4.5 User-system concurrency

In Sections 9.3.2 and 9.3.3, we discussed support for multiple users and systems, and more general support for concurrency. But there is already pervasive concurrent activity among the user, the system, and the environment. While we have addressed specific forms of interparty concurrency, such as drag feedback (Section 6.3) or the passage of time (Section 8.2.1), we have avoided the general issue of user-system concurrency.

Perhaps the most common form of user-system concurrency involves buffered input. On a heavily loaded command-line interface, the user may be able to type several characters before the system echoes them back; in a direct-manipulation interface, the user often can begin a new action while visual feedback for the last one is still taking place.

Our formal definition of the UAN ignores the possibility that the user will start a new action while the system is completing a preceding action. In principle, it is easy enough to remedy this situation; wherever a user action can overlap a preceding system action, add paths in which the system action and user action are replaced with initiate-continue-

terminate tripartite activities, and allow arbitrary overlapping between the system activity and the user activity.

Of course, the situation becomes more complicated very quickly as we impose limits on the amount of buffering that can take place, timing requirements for feedback, and interactions between delayed system activities and user acton viability conditions. Again, timetrees can in principle address these issues; but we cannot build timetree models of representations for these effects until we *have* representations for these effects. If we want to use the UAN to specify this kind of behavior, we must improve its current ad-hoc approach to input buffering and user-system concurrency.

9.4.6 New forms of task composition

As we illustrated in Section 6.6.7, there are forms of task composition that accurately describe the behavior of real-world systems, but cannot be expressed in the UAN. A systematic expansion of the UAN's task composition operators may allow it to represent any kind of task composition that occurs in the real world; at the very least, we can improve its breadth.

Simple forms of task composition allow a user to perform a set of tasks in sequence (sequential composition), to perform the tasks in arbitrary order (order-independent composition), or to choose one task from the set (choice composition). In addition, it should be possible to specify that *one or more* tasks from the set are executed, either sequentially or in arbitrary order. Right now, this can only be done by explicitly listing the various combinatorial possibilities.

These modalities of choice (do one, do some, do all) should also be available in conjunction with interleaved and concurrent execution. It may even be desirable to allow sequential modalities in conjunction with interleaving and concurrency, perhaps to specify the order in which actions or tasks must begin (as discussed in the previous section).

It should also be possible to specify interleaved or concurrent closure — that is, an arbitrary number of instances of *the same task* executing at the same time (see Section

6.6.7). This raises questions pertaining to multiple instances of state variables, multiple copies of interface objects, and assignment of user actions to task instances. We expect that a formal representation for arenas will allow us to address these issues.

Finally, we have examined a peculiar form of concurrent or interleaved composition which we call "lock-step concurrency." In this form of concurrency, a single sequence of user events advances two or more tasks at the same time. The only clear example we have found so far involves the Macintosh "Key Caps" desk accessory; if this desk accessory is visible while another application is running, keystrokes peform their usual function in the application, but they also are highlighted on the keyboard image in the Key Caps window. The application task and the "Use Key Caps" task advance in response to user keystrokes, each behaving as they would alone, but locked in synchrony with the user's actions. We doubt that this is significant, since we have found no other compelling examples. Still, it puzzles us, and we hope a more general approach to task composition will provide it with a home.

10 Conclusions

Our goal in this research has been to develop a formal model of behavioral phenomena and demonstrate the model's utility. We have developed such a model based on the timetree structure, and we have demonstrated its utility by defining the UAN in terms of timetrees, by relating the UAN to an input device model, and by presenting analytic techniques that timetree-based descriptions allow.

As a formal model, timetrees capture the phenomena important to behavioral interface design and description. These phenomena include user, system, and environmental activity and state; task structures, including composition, temporal relations, and abstractions; and certain task interactions, most importantly the problem of tasks that share a common prefix of user actions and system responses. In addition, while the model does not provide any special representation of relationships between activity and state, it provides a framework for techniques and structures that can represent such relationships.

There are several approaches to validating theoretical models. In some cases, it is useful to prove logical assertions about the model. In other cases, it is more useful to build systems that implement the model. A third approach, the one we have chosen, involves applying the model to solve existing problems.

The model's first application has been the development of a formal definition for the UAN. We have defined UAN entities in terms of state components; user and system actions in terms of activities, viability conditions, and effects on state component values; viability conditions in terms of state component values their effects on task composition; and UAN composition operators in terms of timetree composition operators.

In addition to formal definitions for the various components of the UAN, the model has provided solutions for some outstanding UAN problems. Our definitions of input devices provide explicit representations of viability conditions that had been implicit in previous definitions. Our definitions of interleavability and interruptibility resolve ambiguities in previous definitions, and improve the expressiveness of the existing UAN operators.

Finally, we provide a definition of intervals and waiting that allows explicit reasoning about both absolute and relative time.

To show that the timetree model is not limited to defining UAN components, we have used timetrees to describe techniques for translating device descriptions from a formal input device model into UAN device descriptions. We have shown how timetrees can capture information from the device model that is of interest to behavioral designers, and how that information can be used to generate UAN representations for new devices.

We have presented analytic techniques that can operate on timetrees to yield early information about interface designs. We have described existing performance-prediction metrics that can be applied to timetrees, and thus to UAN descriptions. We have proposed a method for generating sequences of possible user actions from UAN descriptions, and a method to distill useful information from collections of such sequences. We have delineated various problems that can be found during early phases of design, including some forms of redundancy, inconsistency, and incompleteness. Finally, we have described a method to make task information available during the execution of a system, with possible applications for on-line help and intelligent assistance.

While we have shown the utility of the timetree model, it still has some limitations. For example, the timetree representation of concurrent composition captures all possible temporal orderings of concurrent tasks, but provides few tools to reduce and abstract the large, complex structures that result. This limitation is shared by some current behavioral specification techniques, including the UAN. As the timetree model is applied to specification techniques for multi-user systems, where concurrent activity is pervasive, more powerful tools for representing concurrency will be needed.

A related problem involves the model's representation of state component values. In some situations, a specification can state that a value is constrained to a particular range. The timetree model can represent only discrete values at specific timestates for specific components. While some applications of ranges can be represented by virtual activities leading to separate timestates, an explicit representation of value ranges seems more

useful. Such a representation will probably necessitate significant changes to disambiguation, state-conflict checking, and composition operators.

Similarly, as mentioned previously, the model itself does not represent relations among activities and state components; while timetree-based definitions capture the *results* of causal relations among activities and state component values, they provide no explicit representation of these relations. This information is important in many applications, and adding it to the timetree model may prove necessary. This may also help resolve the issue of distinguishing between "system choice" and ambiguous specification in cases where multiple system activities are viable at a single timestate.

To summarize, this work has made the following main contributions to the body of human-computer interaction knowledge:

- a formal model of behavior capturing activity, state, task structure, and interactions among these phenomena
- a formal definition of the UAN, including user and system actions, viability conditions, effects on state, and task composition operators
- improved understanding of some task phenomena, including interleavability, interruptibility, implicit viability conditions, and time
- improved definitions of some UAN composition operators and input representations
- a link between an input device model and the UAN
- techniques for analyzing timetree-based behavioral specifications, including performance prediction, tests for some aspects of usability, tests for some forms of inconsistency and incompleteness, and run-time provision of task information

Current limitations of the timetree model include:

- limited facilities for managing complexity of concurrent composition
- limited power for representing ranges or constraints on state component values
- no explicit representation of causality
- no explicit representation of relations among activities and state components

Beyond these contributions, the timetree model has opened our eyes to even more opportunities for future investigation. There are many ways in which the expressiveness

of the current UAN can be improved, and the timetree model will play an important role in the definition of new UAN constructs and entities; indeed, the model has already provided insights into some UAN topics that were previously obscure. Automated implementation of UAN descriptions remains a long-term goal, and we believe that timetrees will provide an important bridge between the behavioral and constructional realms. Finally, there are areas in which the timetree model itself can be extended, including more powerful techniques for representing concurrency, more flexible representations of state information, and explicit relations among activities and state components. We expect investigation of new interface styles, such as gestural, immersive, and cooperative multi-user interfaces, to help us determine how best to provide these new features.

Bibliography

Byrne, M.D., Wood, S.D., Sukaviriya, P., Foley, J.D., and Kieras, D.E. (1994). Automating Interface Evaluation. In *CHI '94*, Boston, 232-237.

Card, S.K., Mackinlay, J.D., and Robertson, G.G. (1990). The Design Space of Input Devices. In *CHI, 1990*, Seattle, Washington, 117-124.

Card, S.K., and Moran, T.P. (1980). The Keystroke-Level Model for User Performance Time with Interactive Systems. *Commun. ACM, 23*, 396-410.

Card, S.K., Moran, T.P., and Newell, A. (1983). *The Psychology of Human-Computer Interaction*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Carroll, J.M., Kellogg, W.A., and Rosson, M.B. (1991). The task-artifact cycle. In J. M. Carroll (Ed.), *Designing interaction: Psychology at the human-computer interface* (pp. 74-102). New York: Cambridge University Press.

Clarke, E.M., Emerson, E.A., and Sistla, A.P. (1986). Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *TOPLS, 8*(2), 244-263.

Emerson, E.A., and Halpern, J.Y. (1986). "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *JACM, 33*(1), 151-178.

Emerson, E.A., and Srinivasan, J. (1988). Branching Time Temporal Logic. In J. W. de Bakker, W.-P. de Roever, & G. Rozenberg (Ed.), *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency* (pp. 123-172). Berlin: Springer-Verlag.

Foley, J., Kim, W.C., Kovacevic, S., and Murray, K. (1989). Defining Interfaces at a High Level of Abstraction. *IEEE Software, 6*(1), 25-32.

Green, M. (1985). The University of Alberta User Interface Management System. *Comput. Graph., 19*(3), 205-213.

Green, M. (1986). A Survey of Three Dialog Models. *ACM Trans. Graph., 5*(3), 244-275.

Green, T.R.G. (1989). Task Action Grammar, presented at the British Computer Society HCI Specialists Group Day Meeting on Task Analysis, May, London. No proceedings.

Hartson, H.R. (1989). User-Interface Management Control and Communication. *IEEE Software, 6*(1), 62-70.

Hartson, H.R., Brandenburg, J.L., and Hix, D. (1992). Different Languages for Different Development Activities: Behavioral Representation Techniques for User Interface Design. In B. A. Myers (Ed.), *Languages for Developing User Interfaces* (pp. 303-328). Boston: Jones and Bartlett.

Hartson, H.R., and Gray, P. (1992). Temporal Aspects of Tasks in the User Action Notation. *Human Computer Interaction, 7*, 1-45.

Hartson, H.R., and Hix, D. (1989). Toward Empirically Derived Methodologies and Tools for Human-Computer Interface Development. *Int. J. Man-Machine Studies, 31*, 477-494.

Hartson, H.R., Hix, D., and Kraly, T.M. (1990). Developing Human-Computer Interface Models and Representation Techniques. *Software—Practice and Experience, 20*(5), 425-457.

Hartson, H.R., and Mayo, K.A. (1995). A Framework for Precise, Reusable Task Abstractions. In F. Paterno (Ed.), *Design, Specification, Verification of Interactive Systems* (pp. 279-298). Berlin: Springer-Verlag.

Hartson, H.R., Siochi, A.C., and Hix, D. (1990). The UAN: A User-Oriented Representation for Direct Manipulation Interface Designs. *ACM Trans. on Info. Sys., 8*(3), 181-203.

Hill, R. (1987). Event-Response Systems — A Technique for Specifying Multi-Threaded Dialogues. In *CHI+GI Conference on Human Factors in Computing Systems*, Toronto, 241-248.

Hix, D., and Hartson, H.R. (1993). *Developing User Interfaces: Ensuring Usability Through Product & Process*. New York: John Wiley & Sons, Inc.

Hoare, C.A.R. (1985). *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall International.

Hughes, G.E., and Cresswell, M.J. (1968). *An Introduction to Modal Logic*. London: Methuen and Co.

Jacob, R.J.K. (1985). An Executable Specification Technique for Describing Human-Computer Interaction. In H. R. Hartson (Ed.), *Advances in Human-Computer Interaction* (pp. 211-242). Norwood, NJ: Ablex.

Jacob, R.J.K. (1986). A Specification Language for Direct Manipulation User Interfaces. *ACM Trans. Graph., 5*(4), 283-317.

Kieras, D., and Polson, P.G. (1985). An Approach to the Formal Analysis of User Complexity. *Int. J. Man-Machine Studies, 22*, 365-394.

Lamport, L. (1980). "Sometime" is sometimes "not never". In *Seventh ACM Symposium on Principles of Programming Languages*, 174-183.

Milner, R. (1980). *A Calculus of Communicating Systems*. Berlin: Springer-Verlag.

Moran, T.P. (1981). The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems. *Int. J. Man-Machine Studies, 15*, 3-51.

Myers, B.A. (1992). *Languages for Developing User Interfaces*. Boston: Jones and Bartlett.

Olsen, D.R., Jr., and Dempsey, E.P. (1983). Syngraph: A Graphical User Interface Generator. *Comput. Graph, 17*(3), 43-50.

Parnas, D.L. (1972a). On the Criteria To Be Used in Decomposing Systems into Modules. *CACM, 15*(12), 1053-1058.

Parnas, D.L. (1972b). A Technique for Software Module Specification with Examples. *CACM, 15*(5), 330-336.

Parnas, D.L., Clements, P.C., and Weiss, D.M. (1985). The Modular Structure of Complex Systems. *IEEE Trans. Softw. Eng., SE-11*(3).

Payne, S.J., and Green, T.R.G. (1986). Task-Action Grammars: A Model of the Mental Representation of Task Languages. *Human-Computer Interaction, 2*, 93-133.

Reisig, W. (1985). *Petri Nets: An Introduction*. Berlin: Springer-Verlag.

Reisner, P. (1981). Formal Grammar and Human Factors Design of an Interactive Graphics System. *IEEE Trans. Soft. Eng., SE-7*, 229-240.

Sharratt, B. (1990). Memory-Cognition-Action Tables: A Pragmatic Approach to Analytical Modelling. In *Interact '90*, 271-275.

Sibert, J.L., Hurley, W.D., and Bleser, T.W. (1988). Design and Implementation of an Object-Oriented User Interface Management System. In H. R. Hartson (Ed.), *Advances in Human-Computer Interaction* (pp. 175-213). Norwood, NJ: Ablex.

Siochi, A.C., Hartson, H.R., and Hix, D. (1990). *Notational Techniques for Accommodating User Intention Shifts* (TR 90-18). Department of Computer Science, Virginia Polytechnic Institute and State University.

Tognazzini, B."Tog". (1992). *Tog on Interface*. Reading, MA: Addison-Wesley.

Wasserman, A.I., and Shewmake, D.T. (1985). The Role of Prototypes in the User Software Engineering Methodology. In H. R. Hartson (Ed.), *Advances in Human-Computer Interaction* (pp. 191-210). Norwood, NJ: Ablex.

Whiteside, J., Bennett, J., and Holtzblatt, K. (1988). Usability Engineering: Our Experience and Evolution. In M. Helander (Ed.), *Handbook of Human-Computer Interaction* (pp. 791-817). Amsterdam: Elsevier North-Holland.

Yunten, T., and Hartson, H.R. (1985). A SUPERvisory Methodology And Notation (SUPERMAN) for Human-Computer System Development. In H. R. Hartson (Ed.), *Advances in Human-Computer Interaction* (pp. 243-281). Norwood, NJ: Ablex.

Vita

Jeffrey Lynn Brandenburg was born to Harold and Elizabeth Brandenburg on April 13, 1963. He entered Virginia Polytechnic Institute and State University (Virginia Tech) in 1980 and graduated Magna Cum Laude in 1984, with a major in Computer Science and a minor in Psychology. He went on to earn his Master of Science degree at Virginia Tech in 1988, with an implementation of the Graphical Kernel System (GKS) for the then-new Macintosh II. After completing the M.S., he started his doctoral research on models of human-computer interaction with Dr. H. Rex Hartson.

Jeff joined the Dialogue Management System (DMS) project as a programmer in 1983, and was associated with the project in varying capacities for the next twelve years. His work with the DMS group included design and implementation of graphical programming tools; optimization of critical code sections for a high-performance database server; specification, design, and implementation of user-interface design tools; and development and implementation of techniques and tools for interface analysis. Projects outside the DMS group included the GKS implementation for his M.S., a Mac-based graphical front end for a robotics simulation to run on Connection Machines, and support for undergraduate UNIX machines. Jeff's teaching experience includes lab sections on assembly language and Pascal, and classes on UNIX and comparative programming languages.

Jeff is a member of the ACM and SIGCHI. His interests include folk and contemporary acoustic guitar and singing; bicycling; science fiction; chemistry; astronomy; electronics; computer science and practice at all levels from hardware design and microprogramming to high-level behavioral design; various aspects of human mental capabilities; and research at the leading fringes of all aspects of science and technology. He has always been an avid student and follower of scientific and technological progress, and looks forward to continuing participation in that progress.