Lecture Notes in Computer Science 1946

Philippe Palanque Fabio Paternò (Eds.)

Interactive Systems

Design, Specification, and Verification

7th International Workshop, DSV-IS 2000
Limerick, Ireland, June 2000
Revised Papers

Springer

Philippe Palanque Fabio Paternò (Eds.)

Interactive Systems

Design, Specification, and Verification

7th International Workshop, DSV-IS 2000
Limerick, Ireland, June 5-6, 2000
Revised Papers

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Philippe Palanque
LIHS University Toulouse 1
Place Anatole France, 31042 Toulouse Cedex, France
E-mail: palanque@univ-tlse1.fr

Fabio Paternò
Consiglio Nazionale delle Ricerche, Istituto CNUCE
Via V. Alfieri 1, 56010 Ghezzano-Pisa, Italia
E-mail: F.Paterno@cnuce.cnr.it

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Interactive systems : design, specification, and verification ; 7th international workshop ; revised papers / DSV-IS 2000, Limerick, Ireland, June 5 - 6, 2000. Philippe Palanque ; Fabio Paternò (ed.). – Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 1946)
ISBN 3-540-41663-3

CR Subject Classification (1998): H.5.2, H.5, I.3, D.2, F.3

ISSN 0302-9743
ISBN 3-540-41663-3 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH
© Springer-Verlag Berlin Heidelberg 2001
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10780903 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 1946
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

试读结束，需要全本PDF请购买 www.ertongbook.com

Preface

The wait for the year 2000 was marked by the fear of possible bugs that might have arisen at its beginning. One additional fear we had during this wait was whether organising this event would have generated a boon or another bug.

The reasons for this fear originated in the awareness that the design of interactive systems is a fast moving area. The type of research work presented at this unique event has received limited support from funding agencies and industries making it more difficult to keep up with the rapid technological changes occurring in interaction technology.

However, despite our fear, the workshop was successful because of the high-quality level of participation and discussion.

Before discussing such results, let us step back and look at the evolution of DSV-IS (Design, Specification and Verification of Interactive Systems), an international workshop that has been organised every year since 1994.

The first books that addressed this issue in a complete and thorough manner were the collection of contributions edited by Harrison and Thimbleby and the book written by Alan Dix, which focused on abstractions useful to highlight important concepts in the design of interactive systems. Since then, this area has attracted the interest of a wider number of research groups, and some workshops on related topics started to be organised. DSV-IS had its origins in this spreading and growing interest. The first workshop was held in a monastery located in the hills above Bocca di Magra (Italy). The event has been held in Italy, France, Belgium, Spain, U.K, Portugal and Ireland, under the auspices of Eurographics, with proceedings regularly published by Springer-Verlag.

After 10 years of research some considerable results have been achieved: we have built a community working on these topics; several projects (European, National, Industrial) have been carried out; various books, journal publications and other related events have been produced; and first industrial products, automatic tools and applications are also appearing based on such approaches.

However, we must admit that interest is growing less quickly than in other areas (Web, mobile communication, usability, ...). The number of new groups working in this area is increasing gradually. One reason is that time-to-market is a crucial factor in industry (and academia!), and consequently more elaborated approaches are less attractive.

To further promote the event and the related topics, we decided to hold it as an ICSE workshop. ICSE is the major international software engineering conference, and we aimed at expounding the topic to this community in order to facilitate interaction and stimulate multidisciplinary approaches and to reach a wider audience. Our proposal was accepted by the ICSE organising committee.

We received 30 submissions from 13 countries. Each paper was reviewed by at least three members of the Programme Committee, and the final selection was made at a meeting held at CHI'2000. Refined versions of less than half of these submissions were selected for inclusion in this book.

The workshop provided a forum for the exchange of ideas on diverse approaches to the design and implementation of interactive systems. The particular focus of this

year's event was on models (e.g., for devices, users, tasks, contexts, architectures, etc.) and their role in supporting the design and development of interactive systems.

As in previous years, we still devoted considerable attention to the use of formal representations and their role in supporting the design, specification, verification, validation and evaluation of interactive systems. Contributions pertaining to less formal representations of interactive system designs and model-based design approaches were also encouraged.

During the workshop discussion and presentations were grouped according to a set of major topics: Designing Interactive Distributed Systems, Designing User Interfaces, Tools for User Interfaces, Formal Methods for HCI and Model-Based Design of Interactive Systems.

At the end of the sessions participants were split into discussion groups. One aspect that attracted the attention of the participants was the book "What is in the future of software engineering" that was distributed to all ICSE participants: we noticed the complete lack of a chapter addressing human-computer interaction. Thus, we feel that these proceedings also have an additional role: to provide the background information for the missing chapter, that on software engineering for human-computer interaction. This lack underscores how the academic community has not yet completely understood the importance of this subject and the importance of the research area aiming at identifying ergonomic properties and improving the design process so that such ergonomic properties are guaranteed in the software systems produced.

If we consider the HCI map proposed in the HCI curriculum produced by ACM SIGCHI we notice that each component (user, computer, development process, use and context) is evolving very rapidly.

It becomes crucial to identify a design space indicating the requirements, modelling techniques, tools, metrics, architectures, representations and evaluation methods characterising this area.

In addition, the research agenda for this field is dense: it includes extending models to deal with dynamicity (mobile users, ...), develop analysis techniques for making use of the models, more tools for usability evaluation, multi * approaches (multimedia, multi users, multi modal, ...) and end user programming.

We think that the reader will find the material presented in this book useful in understanding these issues, and we sincerely hope it will also prove to be useful in stimulating further studies and improving current practise.

September 2000 Philippe Palanque and Fabio Paternò

Programme Committee

Ann Blandford	University of Middlesex, U.K.
Alan Dix	University of Huddersfield and aQtive Ltd.
David Duce	Oxford Brookes University, U.K.
David Duke	University of Bath, U.K.
Giorgio Faconti	CNUCE-C.N.R., Italy
Miguel Gea	University of Granada, Spain
Nicholas Graham	Queen's University, Canada
Michael Harrison	University of York, U.K.
Robert Jacob	Tufts University, U.S.A.
Chris Johnson	University of Glasgow, U.K.
Peter Johnson	University of Bath, U.K.
Fernando Mario Martins	University of Minho, Portugal
Panos Markopoulos	IPO, University of Eindhoven, The Netherlands
Philippe Palanque (Co-chair)	LIHS, Université Toulouse I, France
Fabio Paternò (Co-chair)	CNUCE-CNR, Italy
Angel Puerta	Stanford University and Red Whale, U.S.A.
Jean Vanderdonckt	Université Catholique de Louvain, Belgium

Sponsoring Organisations

ICSE 2000

The 22nd International Conference on Software Engineering

Limerick, Ireland

Contents

Designing Interactive Distributed Systems

Specifying Temporal Behaviour in Software Architectures for Groupware Systems 1
Timothy N. Wright, T.C. Nicholas Graham (Queen's University) and Tore Urnes (Telenor Research and Development)

Questioning the Foundations of Utility for Quality of Service in Interface Development 19
Chris Johnson (Department of Computing Science, University of Glasgow)

Designing User Interfaces

A Framework for the Combination and Characterization of Output Modalities 35
Frédéric Vernier and Laurence Nigay (CLIPS-IMAG, Grenoble)

Specifying Multiple Time Granularities in Interactive Systems 51
Maria Kutar, Carol Britton and Chrystopher Nehaniv (University of Hertfordshire)

Verifying the Behaviour of Virtual Environment World Objects 65
James S. Willans and Michael D. Harrison (HCI Group, University of York)

Tools for User Interfaces

SUIT – Context Sensitive Evaluation of User Interface Development Tools 79
Joanna Lumsden and Philip Gray (Department of Computing Science, University of Glasgow)

Structuring Interactive Systems Specifications for Executability and Prototypability 97
David Navarre, Philippe Palanque, Rémi Bastide and Ousmane Sy (LIHS, University Toulouse 1)

A Toolkit of Mechanism and Context Independent Widgets 121
Murray Crease, Philip Gray and Stephen Brewster (Department of Computing Science, University of Glasgow)

Formal Methods for Human-Computer Interaction

Integrating Model Checking and HCI Tools to Help Designers
Verify User Interface Properties 135
Fabio Paternò and Carmen Santoro (Instituto CNUCE-CNR)

More Precise Descriptions of Temporal Relations within Task Models 151
Anke Ditmar (University of Rostock)

Formal Interactive Systems Analysis and Usability Inspection Methods:
Two Incompatible Worlds? 169
Karsten Loer and Michael Harrison
(BAE SYSTEMS Dependable Computing Systems Centre,
University of York)

Model-Based Design of Interactive Systems

Wisdom – A UML Based Architecture for Interactive Systems 191
Nuno Jardim Nunes (Universidade da Madeira,
Unidade de Ciências da Computação) and
João Falcão e Cunha, (Universidade do Porto, GEIN,
Faculdade de Engenharia)

User Interface Declarative Models and Development Environments:
A Survey 207
Paulo Pinheiro da Silva (Department of Computer Science,
University of Manchester)

The Task-Dialog and Task-Presentation Mapping Problem:
Some Preliminary Results 227
Quentin Limbourg, Jean Vanderdonckt, and Nathalie Souchon
(Université catholique de Louvain, Institut d'Administration et
de Gestion)

Indexes

Subject Index 247

Author Index 251

Specifying Temporal Behaviour in Software Architectures for Groupware Systems

Timothy N. Wright[1], T.C. Nicholas Graham[2], and Tore Urnes[3]

[1] University of Canterbury, Private Bag 4800, Christchurch, New Zealand
tnw13@cosc.canterbury.ac.nz
[2] Queen's University, Kingston, Ontario, Canada K7L 3N6
graham@cs.queensu.ca
[3] Telenor Research and Development, P.O. Box 83, N-2007 Kjeller, Norway
tore.urnes@telenor.com

Abstract. This paper presents an example of how software architectures can encode temporal properties as well as the traditional structural ones. In the context of expressing concurrency control in groupware systems, the paper shows how a specification of temporal properties of the semi-replicated groupware architecture can be refined to three different implementations, each with different performance tradeoffs. This refinement approach helps in understanding the temporal properties of groupware applications, and increases confidence in the correctness of their implementation.

1 Introduction

Software architectures traditionally decompose systems into *components* responsible for implementing part of the system, and *connectors* enabling communication between these components. Components implement some part of the system's functionality, while connectors specify the form of intercomponent communication, for example, through method calls or events [28]. We refer to these as structural properties of the architecture.

In synchronous groupware applications, it is not only important to capture *how* components may communicate, but *when*. For example, in a multiuser video annotation system, it is important that all participants see and annotate the same frame [14]. In a shared drawing application, it is important that the drawing operations of participants do not conflict, for example with one person deleting a drawing object that another is moving. As the paper will show, such requirements on sequencing of updates and synchronization of shared state can be expressed as restrictions on when messages can be passed between components involved in an interaction.

This paper investigates how software architectures can specify temporal properties of an application as well as structural ones. From these temporal specifications, a variety of implementations can be derived, embodying different execution properties. This allows an approach where software architectures specify high level temporal properties of implementations, allowing architecture implementers to plug-replace any implementation meeting these properties.

P. Palanque and F. Paternò (Eds.): DSV-IS 2000, LNCS 1946, pp. 1–17, 2001.
© Springer-Verlag Berlin Heidelberg 2001
试读结束，需要全本PDF请购买 www.ertongbook.com

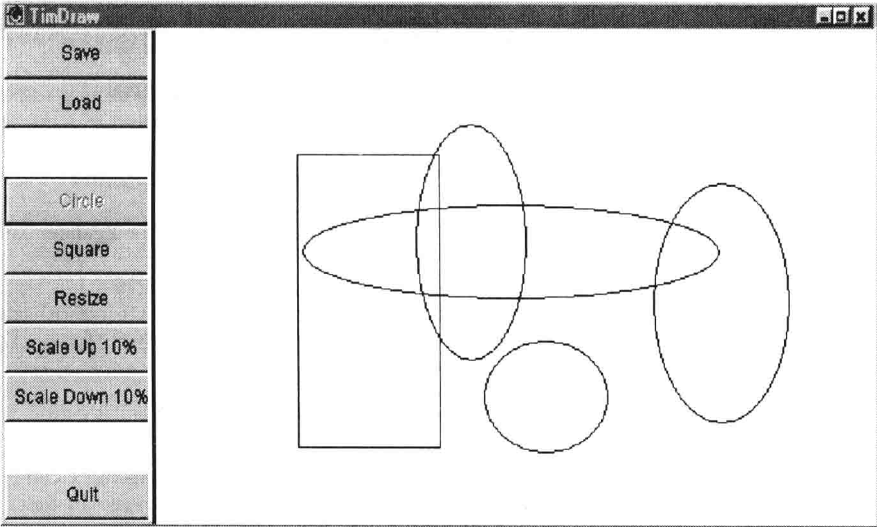

Fig. 1. A Groupware Drawing Program. This program was implemented in Java using the *TeleComputing Developer Toolkit* (TCD) [1].

As will be shown in the paper, the benefits of this approach are:

- Difficult temporal properties of groupware applications can be treated orthogonally to the application's functionality by embedding these properties in the software architecture;
- Premature commitment to algorithms implementing temporal properties can be avoided, as early design of the system focuses on desired behaviour rather than on algorithms implementing that behaviour;
- The process of specifying properties and refining implementations increases confidence in the correctness of the implementations and provides a clearer understanding of the temporal properties of the application.

In order to demonstrate this approach, we take the example of the implementation of concurrency control in a semi-replicated groupware architecture. We show how concurrency control properties can be encoded in the definition of the semi-replicated architecture itself. Specifically, we treat the problem of ensuring that transactions performed on shared data state are serializable, guaranteeing that operations performed by users do not conflict.

As we shall see in the paper, concurrency control algorithms are complex, and embody trade-offs of degree of consistency versus response time. It is therefore beneficial to separate the specification of the desired concurrency properties of an application from the concurrency control algorithm actually implementing it. To demonstrate this assertion, the paper is organized as follows. Section 2 describes the concurrency control problem in groupware, and introduces a simple groupware drawing tool as an example application. Section 3 introduces the widely used semi-replicated implementation architecture for groupware, and shows how it can be described to possess temporal properties ensuring correct concurrent behaviour. In order to show the flexibility of such a specification, sections 4 through 6 introduce the locking, Eager and adaptive concurrency control algorithms as implementations re-

fined from the semi-replicated architecture. These algorithms have all been implemented as part of the *TeleComputing Developer* (TCD) groupware development toolkit [1].

2 Motivation

To introduce the concurrency control problem and to motivate our approach of encoding temporal properties of applications in the software architecture, we present a simple groupware drawing program. As shown in figure 1, users may draw simple objects such as squares and circles on a shared canvas. Each user's actions are reflected in the canvases of other users in real time. In addition to standard editing operations, users may scale the entire diagram up or down, in increments of 10%.
In the implementation of the drawing program, a shared data structure (or *shared context*) contains the set of drawing objects. Figure 2 shows how operations for resizing and scaling objects are implemented. For example, a resize operation reads the object to be resized from the shared context, changes its size, and saves the object back to the shared context. Similarly, the scale operation scales each of the drawing objects in the shared context.

Figure 2 shows how concurrency problems can arise if two users simultaneously perform a resize and a scale operation. Here, the resize operation is performed while the scale is taking place, partially undoing the effect of the scale. This leaves the diagram in an inconsistent state, where the scale has been applied to all elements except the first. When two user actions lead to an inconsistent result, those actions are said to *conflict*. Concurrency control algorithms are designed to prevent the negative effects of conflicting actions.

2.1 Concurrency Control Styles

Concurrency control algorithms can be roughly divided into two classes – pessimistic and optimistic. Pessimistic schemes guarantee that when a participant in a groupware session attempts to modify the shared artifact, his/her actions will not conflict with the actions of other participants. This guarantee leads to intuitive user interface behaviour, but at the cost of responsiveness. Optimistic approaches, on the other hand, assume that actions will not conflict, and must detect and repair conflicts when they occur.

Resize object "1" to newSize	Scale entire diagram by k%
	n=**getNumberObjects**()
	o1=**getObjectAt**("1")
s=**getObjectAt**("1")	
s.setSize(newSize)	o1.scale(k)
	setObjectAt("1",o1)
setObjectAt("1",s)	o2=**getObjectAt**("2")
	...

Fig. 2. A resize operation conflicting with a scale operation.

Under pessimistic algorithms, update transactions resulting from user actions never fail. One way of achieving this property is to require clients to obtain a lock on the shared context before attempting to process a new user action [22]. This locking may reduce the potential for concurrent execution of clients and introduces networking overhead to obtain locks.

Under optimistic algorithms, update transactions may fail, potentially requiring work to be undone [16]. Optimistic algorithms improve performance by allowing client machines to process user actions in parallel.

Neither pessimistic nor optimistic approaches are suitable for every application. While optimistic approaches may provide better response times for short transactions that are inexpensive to undo [3,29], pessimistic algorithms are preferable in the following three cases:

- *Undo unacceptable:* In some applications, it is impossible to roll back user actions that are retroactively found to conflict with other actions. Examples of such actions include deleting a file or sending an email message.
- *Pessimistic faster:* To be effective, optimistic schemes rely on conflicts being rare, and the cost of undoing operations being inexpensive. Consider the scale operation of figure 2. This operation performs one read and write to the shared context for every drawing object. In a complex drawing with potentially tens or hundreds of objects, the scale operation is likely to conflict with an operation performed by some other user.
- *Optimistic unfair:* In a wide area network, some users may suffer longer latencies than others when accessing parts of the shared context. The actions of these users may be more likely to conflict than the actions of users with lower latency. Fairness may require that users with poor network connections use pessimistic concurrency control.

Concurrency control algorithms therefore embody tradeoffs in the desired behaviour of systems, but all provide the basic property of guaranteeing serializability of transactions carried out by participants in the groupware session. That is, the algorithm should never permit operations to conflict as in the example of figure 2. Our approach is therefore to encode this temporal property of transaction serializability as part of

the definition of the software architecture. We then show how these temporal properties can be implemented by both pessimistic and optimistic algorithms, and by a novel algorithm combining the two. This approach allows us to specify the desired temporal behaviour of the architecture (i.e., transaction serializability) separately from the algorithm used, avoiding premature commitment to a particular concurrency control algorithm.

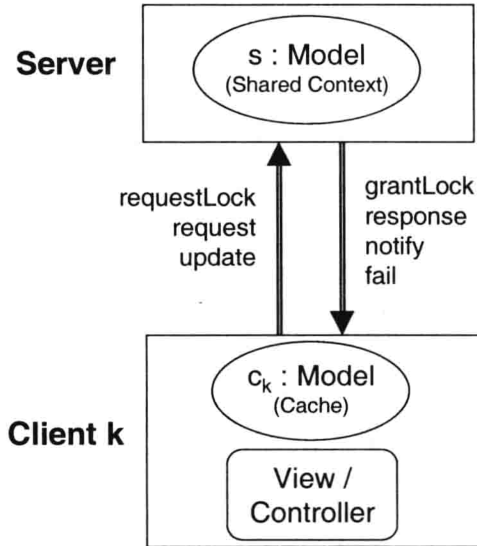

Fig. 3. The semi-replicated implementation architecture for groupware: A shared context is represented on a server machine. Clients contain a cache, a read-only replica of the shared context. Local context does not require concurrency control, and therefore is not represented. Writable replicas of the shared context are assumed to have no concurrency control, and therefore are also not represented.

All of these algorithms have been realized using the Dragonfly [1] implementation of the semi-replicated groupware architecture, in the TCD toolkit. In TCD, we exploit the separation of specification of temporal behaviour from its implementation, allowing concurrency control algorithms to be plug-replaced after the application has been developed.

3 The Semi-Replicated Architecture for Groupware

We model groupware systems using a semi-replicated architecture [15]. Semi-replicated systems are hybrid centralized/replicated systems, where all shared state is represented on a centralized component, some shared state is replicated to the clients, and private state is represented on the clients. Some shared state is replicated in the form of a read-only *client cache*.

Semi-replication is based on the *Model-View-Controller* (MVC) architecture for groupware development [20,15]. In MVC, the shared state underlying each participant's view is located in a *model*, a *controller* is responsible for mapping user actions onto updates to the model, and a *view* is responsible for updating the display in re-

sponse to changes in the model. MVC (and related architecture styles such as PAC* [5]) underlies a wide range of groupware development tools. Despite earlier suspicion that semi-replication is inherently inefficient [21], performance evaluation has shown this architecture to provide excellent response times, even over very wide area networks [29].

Figure 3 shows the elements of this model that are necessary to illustrate how concurrency control properties can be encoded within a software architecture connector. The figure further shows the set of messages allowing the client and server components to communicate. These messages are described in detail in section 3.2.

We assume that no concurrency control is applied to private state represented on clients (since there is no concurrent access to this state), and therefore omit local context from the model. We assume that the client cache is not writable by the client, and therefore can only be updated by the server. We further assume that any replicated state that is writable by the client has no concurrency control associated with it, and therefore need not be included in the model. Despite what may appear to be restrictive assumptions, this model describes the implementation architecture of a wide range of existing groupware development tools. (The following discussion is based on Phillips' survey of groupware development tools and their implementation architectures [24]).

Semi-replicated tools directly implementing this model (or subsets of the model) include Clock [29], TCD [1], Weasel [13], Suite [9], and Promondia [12]. GroupKit [25] is described by the model, as GroupKit environments implement shared state, and GroupKit provides no concurrency control for replicated shared data. Figure 3 also describes systems with replicated state under centralized coordination such as Habanero [6], Prospero [10], Ensemble [23] and COAST [27]. In these systems, a central component is responsible for concurrency control decisions, allowing the shared context to be modeled via a virtual server. Finally, the model describes fully centralized systems such as RendezVous [17], as the trivial case in which there is no replicated data at all.

Systems not described by the model include fully replicated systems using concurrency control algorithms based on roll-backs [8] or operation transforms [11]. Such fully replicated systems include DECAF [23], DreamTeam [26], Mushroom [19] and Villa [4].

Therefore, while this simplified treatment of the semi-replicated architecture does not cover every possible implementation of groupware, it describes a sufficiently large subset of current development tools to be interesting.

3.1 Encoding Concurrency Control in the Semi-replicated Architecture

In order to show how software architectures can encode temporal properties, we first formalize our simplified version of the semi-replicated architecture, and then define its concurrency control properties as restrictions over the treatment of messages.

As shown in figure 3, a groupware system consists of a set of client machines, each containing a cache, and a server machine containing shared state. Clients communicate with the server by issuing *requests* for information and *updates* that modify information. Parameters to requests and updates and responses to requests are all considered to be *values*.

Client and Server Components

We let *Client* $\subset \mathbb{N}$ represent a set of client machines. We define *Update*, *Request* and *Value* to be disjoint sets representing updates and requests made by the view/controller, and values returned by the model as the results of requests. We let $Time == \mathbb{N}$ represent time.

Model

A *Model* stores data. Models are queried via requests. The values of these requests may change over time.

$$Model == Time \times Request \rightarrow Value$$

If $m{:}Model$ we write $m(t)$ to represent $\lambda r \bullet m(t,r)$,the snapshot of the model at time t.

As shown in figure 3, we let $s{:}Model$ represent the shared context, and the family of functions $c_k{:}Model$ represent a cache for each client $k \in Client$. When making requests, clients first consult their cache. If the response has not been cached (i.e., the request is not in the domain of the cache), the shared context is consulted. If used efficiently, a cache can considerably reduce the overhead of network communication [15]. We define a request function rq_k for each client $k \in Client$:

$$
\begin{array}{l}
rq_k : Time \rightarrow Request \rightarrow Value \\
\quad rq_k\,(t,r) == \\
\qquad \text{if } r \in \text{dom}\,(c_k\,(t)) \text{ then} \\
\qquad\quad c_k\,(t,r) \\
\qquad \text{else} \\
\qquad\quad s\,(t,r)
\end{array}
$$

View/Controller

The purpose of an MVC controller is to map user inputs onto updates to the model. In computing an update, the controller makes a set of requests to the model. We formalize the activity of the controller through an *update function,* which computes an update using values obtained from the model:

$$UpdateFn == \text{seq } Value \rightarrow Update$$

An update *transaction* represents the application of an update function to values obtained through a sequence of requests executed at given times. Transactions originate from some client.

$$
\begin{array}{l}
Transaction == \\
\quad Client \times UpdateFn \times \text{seq}\,(Time \times Request)
\end{array}
$$

The view/controller of each client can be thought of as executing a sequence of transactions. When a user performs an action, an update to the shared state is computed, based on values in the cache and shared context. When a client receives notification that the shared context has changed, it computes an update to the display.

试读结束，需要全本PDF请购买 www.ertongbook.com