A Modular Middleware Flow Scheduling Framework*

Alexandre R.J. François and Gérard G. Medioni
Integrated Media Systems Center
University of Southern California, Los Angeles, CA
{afrancoi,medioni}@iris.usc.edu

ABSTRACT

Immersive, interactive applications require on-line processing and mixing of multimedia data. In order to realize the *Immersipresence* vision, we propose a *generic*, *extensible*, *modular* multimedia system software architecture. We describe here the Flow Scheduling Framework (FSF), that constitutes the core of its middleware layer. The FSF is an *extensible* set of classes that provide basic *synchronization* functionality and composition mechanisms to develop data-stream processing components. In this dataflow approach, applications are implemented by specifying data streams and their path through processing nodes, where they can undergo various manipulations. We describe the details of the FSF data and processing model that supports stream synchronization in a concurrent processing framework. We illustrate the FSF concepts with a *real-time* video stream processing application.

Keywords

Multimedia middleware; dataflow programming; synchronization.

1. INTRODUCTION

We present a Flow Scheduling Framework (FSF) to support generic processing and synchronization of data streams in a modular multimedia system.

Immersive, interactive applications require on-line processing and mixing of multimedia data such as pre-recorded audio and video, synthetic data generated at run-time, live input data from interaction sensors, media broadcast over a non synchronous channel (e.g. the internet), etc. Such applications present numerous challenges researched in many separate fields such as signal processing, computer vision, computer graphics, etc. Besides, several solutions can be developed for a given problem. Independent, partial solutions must therefore be gathered and integrated into working applications. Collecting, understanding and adapting independently developed components is always a challenging exercise. A common platform, providing a unifying data model, would facilitate the development and exploitation of multimedia software components and applications.

Recent multimedia architectures addressing on-line processing include MIT's VuSystem [2] and BMRC's Continuous Media Toolkit [3]. Both systems implement modular dataflow architecture concepts. They are designed primarily for audio and video processing with a strong emphasis on capture and replay aspects, and do not seem to easily scale up to applications involving immersion, interaction and synthetic content mixing. Earlier, De Mey and Gibbs proposed an object-oriented Multimedia Component Kit [4], that implements a framework for rapid prototyping of distributed multimedia applications. Our approach is based on the same concepts: object-oriented, component-based extensible middleware and dataflow-based visual composition of multimedia application. The difference is in our definition of multimedia application: in order to realize the *Immersipresence* vision, we propose the *generic*, *extensible*, *modular* multimedia system software architecture presented in figure 1. A *middleware layer* provides an abstraction level between the low-level services and the applications, in the form of software components. An *application layer* allows to compose and execute multimedia applications using the software components in a dataflow-based, immersive programming environment.

The object of this paper is the description of the Flow Scheduling Framework (FSF) that is the core of the middleware layer. It is an extensible set of classes that provide basic synchronization functionality and composition mechanisms to develop data-stream processing components in the context of the proposed multimedia system architecture. In this dataflow approach, an application is the specification of data streams flowing through processing nodes, where they can undergo various manipulations. The FSF specifies and implements a *common generic data and processing model* designed to support stream synchronization in a concurrent processing framework. This extensible model allows to encapsulate existing data formats and standards as well as low-level service protocols and libraries, and make them available in a system where they can inter-operate.

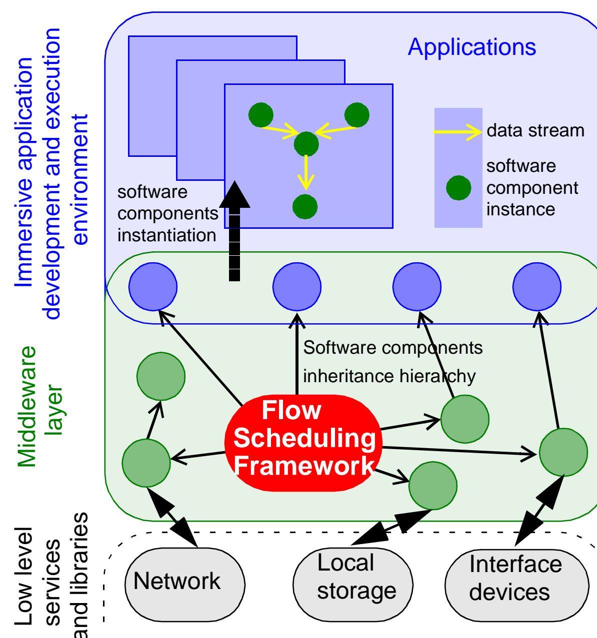

Figure 1. A modular multimedia system software architecture

Copyright © 2000 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or permissions@acm.org

*This research has been funded by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152, with additional support from the Annenberg Center for Communication at the University of Southern California and the California Trade and Commerce Agency.

2. OVERVIEW OF THE FSF CONCEPTS

Information is modeled as data *streams*. A stream represents one or several synchronized multimedia objects, i.e. expressed in the same time referential. An application is specified by a number of streams of various origins and the manipulations they undergo as they pass through processing nodes called *cells*. *Intra-stream processes* include formatting, alteration and presentation. A stream can originate from local sources such as local storage or input devices, or can be received from a network. It can also be created internally as the result of the manipulation of one or several other streams. As a stream goes through different cells, it can be altered. After processing, a stream can be stored locally, sent to output devices or sent on a network. *Inter-stream operations* are necessary for synchronization, as time referential transformations require the exchange or re-organization of data across streams. Streams occur in an application as time samples, which can represent instantaneous data, such as samples describing the state of a sensor at a given time, or integrated data describing, at a given time, the state of one or several objects over a certain period of time. Furthermore, the nature of the object(s) represented by a stream dictates the parameters of sample traffic in the stream. While some streams represent a steady flow of uniform time rate samples (e.g. captured frames from a video camera), others represent occasional arrival of samples at random time intervals (e.g. user input). Finally, we make a distinction between *active streams*, that carry volatile information, and *passive streams*, that hold persistent data in the application. For example, the frames captured by a video camera do not necessarily need to remain in the application space after they have been processed. Process parameters however must be persistent during the execution of the application.

3. DATA MODEL

We define a structure that we call *pulse*, as a carrier for all the data corresponding to a given time stamp in a stream. In an application, streams can be considered as pipes in which pulses can flow (in one direction only). The time stamp characterizes the pulse in the stream's time referential, and it cannot be altered inside the stream. Time referential transforms require inter-stream operations.

As a stream can represent multiple synchronized objects, a pulse can carry data describing time samples of multiple individual objects as well as their relationships. In order to facilitate access to the pulse's data, it is organized as a mono-rooted composition hierarchy, referred to as the pulse structure (see figure 3). Each node in the structure is an instance of a *node type* and has a *name*. The node name is a character string that can be used to identify instances of a given node type. The framework only defines the base node type that supports all operations on nodes needed in the processing model, and a few derived types for internal use. Specific node types can be derived from the base type to suit specific needs, such as encapsulating standard data models. Node attributes can be *local*, in which case they have a regular data type, or *shared*, in which case they are subnodes, with a specific node type. Node types form an inheritance hierarchy to allow extensibility while preserving reusability of existing processes.

4. PROCESSING MODEL

We designed a processing model to manage generic computations on data streams. We define a generic cell type, called *Xcell*, that supports basic stream control for generic processing. Custom cell types implementing specific processes are derived from the Xcell type to extend the software component base in the system. In particular, specialized cell types can encapsulate existing standards and protocols to interface with lower-level services provided by the operating system, devices and network to handle for example distributed processing, low-level parallelism, stream presentation to a device, etc.

In an Xcell, the information carried by an active stream (*active pulses*) and a passive stream (*passive pulses*) is used in a process that may result in the augmentation of the active stream and/or update of the passive stream (see figure 2). Each Xcell thus has two independent directional stream channels with each exactly one input and one output. Part of the definition of the Xcell type and its derived types is the process that defines how the streams are affected in the cell. The base Xcell only defines a place-holder process that does not affect the streams.

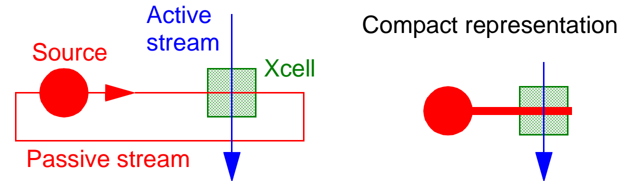

Figure 2. Generic processing unit

4.1 Flow Control

In an Xcell, each incoming active pulse triggers the instantiation of the cell process, to which it is the input data. The process can only read the active pulse data and add new data to the active pulse (i.e. augment its structure). *Existing data cannot be modified.*

Process parameters are carried by the passive stream. A passive stream must form a loop anchored by exactly one source that produces a continuous flow of pulses reproducing its incoming (passive) pulses. The passive pulse arriving at the cell at the same time as the active pulse is used as parameters for the corresponding process instance. The process can access and may update both the structure and the values of the passive pulse.

When the process is completed, the active pulse is sent to the active output. The passive pulse is transmitted on the passive output, down the passive loop, to ultimately reach the source where it becomes the template for generated pulses. Note that the continuous pulsing on passive loops is only a conceptual representation used to provide a unified model. In order to make application graphs more readable, passive loops are represented in compact form as self terminating bidirectional links (see figure 2).

In this model, the processing of different active pulses of the same active stream can occur in parallel, as pulses are received by the cell. However a given active pulse being processed in a cell is not output until the process is complete. Furthermore, a cell cannot alter the time stamp of an active pulse. Time referential transformations, such as buffering or pulse integration, require to make the active pulse data persistent *via* a passive loop, as described later.

4.2 Filtering

In an Xcell derived cell, the process is defined as a function that takes as input data an active pulse and a passive pulse, and may augment the active pulse and/or modify the passive pulse. The process description includes the specification of the associated input and parameter types, i.e. substructures to look for in the active and passive pulses structures respectively. When designing custom cells, corresponding local data types and node types must be defined if not already available. A partial structure type is specified as a *filter* or a composition hierarchy of filters (see figure 3). A filter is an object that specifies a node type, a node name and eventual subfilters corresponding to subnodes. The filter composition hierarchy is isomorphic to its target node structure. When an active pulse is received, the cell process input structure must be identified in the active pulse, and similarly the parameter structure must be identified in the passive pulse before the process can be started (see figure 4). Finding substructures in pulses is called filtering,

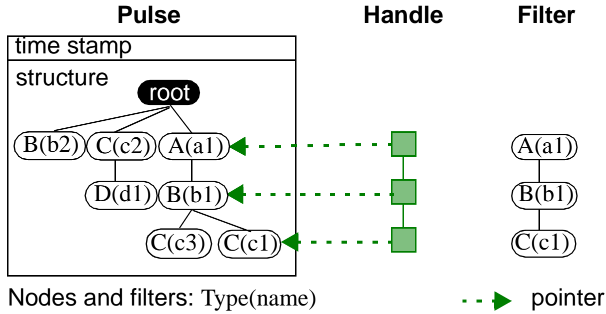

Figure 3. Pulse filtering

and is a subtree matching operation. In order to provide efficient access to relevant data in the pulses, filtering operations return *handles*, that are used in the process for direct access to the relevant structure nodes. A handle is a pointer to a pulse node. Handles can be organized in composition hierarchies isomorphic to filter and node structures. A handle or composition hierarchy of handles is formed as the result of the filtering of a pulse's structure.

The Node names specified in filters to identify instances can be exact strings or string patterns to allow multiple matches. The filtering process thus returns a list of handles, one for each match. The interpretation of multiple matches in the active pulse is part of a specific process implementation. For example, the process can be applied to each match, thus allowing to define operations on objects whose count is unknown at design time.

Filtering efficiency is a major issue especially on active streams, since it must be performed independently on each new active pulse. In traditional dataflow models, data resulting from processes is not accumulated in a structure, but simply output from a processing node. Type checking can therefore occur at application design time, and there is no need to look for the relevant data in the incoming stream. After experimenting with this approach, we found that in multimedia applications, where synchronization is a major constraint, keeping data synchronized in our pulse structure is more efficient than separating data elements for processing and later having to put synchrone samples, distributed in independent storage nodes, back together. Our hierarchical structuration of the pulse data is suitable for efficient implementation of filtering. Furthermore, the filtering approach allows to apply a same process to an undetermined number of objects in an active pulse while preserving the temporal relationship of the object samples. It also makes it possible for several cells on a passive loop to share part or totality of their parameters.

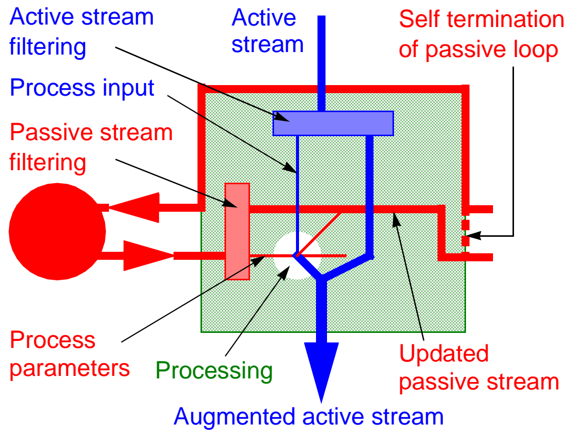

Figure 4. Inside an Xcell

5. APPLICATION COMPOSITION

5.1 Stream Routing

An application is specified by a graph of cells with two independent sets of links: active and passive. Active and passive connections are different in nature, and cannot be interchanged.

There is no limitation on the number of cells on a same passive loop. Feed-back can thus be implemented either with cells that directly update their parameters in their process, or with separate cells for processing and update, if the update depends on some subsequent processes, as shown in figure 5. Updated parameters are used in processes as they become available on the corresponding passive loop, thus feed-back loops cannot create any interlock.

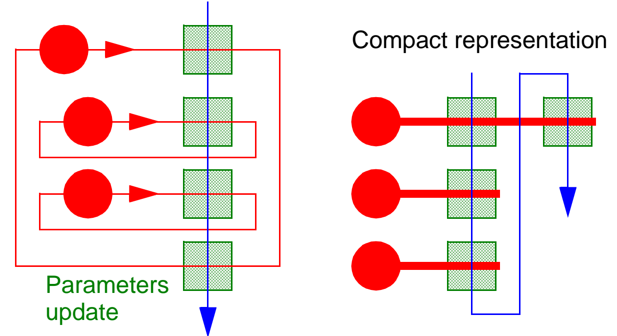

Figure 5. Parameter feed-back on a passive loop

The set of active connections between the cells is a *dependency graph*. A cell using as input the result of another cell must be traversed after the cell on which it depends in order for its process to be executed (otherwise the filtering fails and pulses are transmitted without any process). Independent cells can be traversed in arbitrary order, and their processes can occur in parallel. To take advantage of this parallelism, we define the *Scell* type, which is a stream splitter (see figure 6). An Scell has one active input and two active outputs. Scells transmit incoming input active pulses on both active outputs simultaneously.

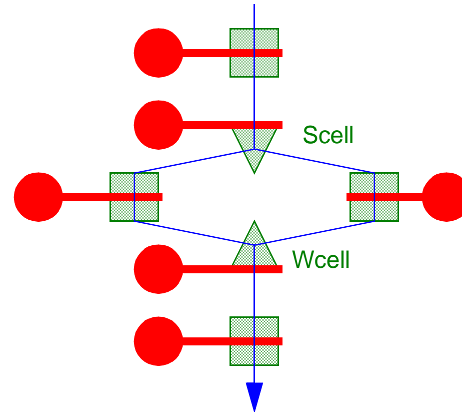

Figure 6. Stream routing for parallel execution

The only local synchronization enforced in the cells is that an active pulse does not reach a given cell before it has been completely processed by the previous cell on the active path. In order to enforce this constraint when a stream follows several parallel paths, we introduce the *Wcell* (see figure 6), which has two active inputs and one active output. A Wcell is needed when a cell depends on several independent cells traversed in parallel: the Wcell waits until it receives the same pulse on both its active inputs before sending it to its active output.

There is no *a priori* assumption on the time taken by cell processes. If the processing time in the upstream cells varies from pulse to pulse, active pulses might reach a given cell out of order. If simple time consistency can be enforced in cell processes, more elaborate synchronization mechanisms are usually required.

5.2 Synchronization

In our framework, synchronization reduces to time referential transformations. As already pointed out, each active stream has its own time referential, to which the pulses' time stamps relate. All passive streams share the same system time referential. The time referential in a stream cannot be altered. Time referential transformations thus require the transfer of pulses across different streams. To do so, active pulses must be stored in a passive loop before they can be reintroduced in a different active stream. Intuitively, if any time reorganization of the active pulses is to occur, they must be buffered. The minimal setup required is presented in figure 7.

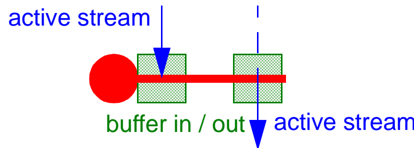

Figure 7. Pulse buffering

Active pulses are buffered, by a specialized cell, to a passive loop in which the pulses describe the content of the buffer at a given time. Buffered pulses may undergo a time transform. Another specialized cell on the same passive loop can retrieve the pulses from the buffer and restitute them in an active stream at the desired rate, determined either by a dedicated timer or by incoming active pulses. This operation may involve resampling.

More generally, if two active streams are to be combined in one single active stream (see figure 8), the pulses from both streams must be buffered in a passive loop structure after undergoing a time transform to place them in a common time referential. Synchrone pulses can be retrieved from the buffer and placed in a same active pulse, after an eventual time transformation. This pattern easily generalizes to several active streams being recombined into several different active streams. Although the time transformation, buffering and eventual resampling when de-buffering are generic concepts, the actual operations to carry are data (node) type and cell process dependent, and thus are not part of the framework definition.

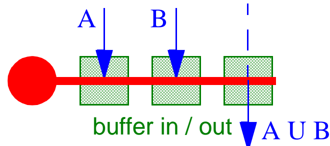

Figure 8. Streams synchronization

6. EXAMPLE APPLICATION

We have implemented a prototype FSF, in the form of a C++ library for the Windows 2000 operating system (the library can easily be ported to other multitasking operating systems). Using this prototype FSF library, we have developed the components needed to build an adaptive color background model-based video-stream segmentation application, nicknamed "blue-screen without a blue screen" (based on [1]). The corresponding application graph is presented in figure 9. The application runs at *thirty* 240x180 frames per second on a dual Pentium III 550 MHz, a frame rate far superior to the one achieved in our previous stand alone implementations. The multithread processing model makes real-time processing possible.

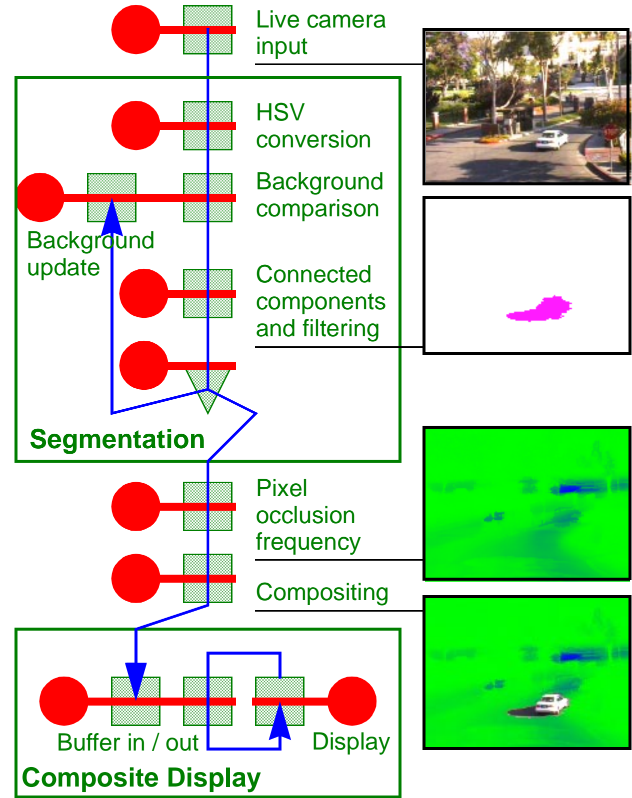

Figure 9. The "blue-screen without a blue screen" application

7. CONCLUSION

We have presented the Flow Scheduling Framework, foundation of the middleware layer of our generic, extensible, modular multimedia system software architecture to support immersive, interactive applications.

The FSF is an *extensible* set of classes that provide basic synchronization functionality and composition mechanisms to develop data-stream processing components in the context of the proposed multimedia system architecture.

We have described the FSF data and processing model to support stream synchronization in a concurrent processing framework, and shown how to build component-based applications in this model. We have demonstrated a prototype FSF implementation with a *real-time* video processing and compositing application.

8. REFERENCES

[1] François A.R.J. and Medioni G.G. Adaptive Color Background Modeling for Real-Time Segmentation of Video Streams. In Proc. Int. Conf. on Imaging Science, Systems, and Technology, pp. 227-232, Las Vegas, NA, June 1999.

[2] Lindblad C.J. and Tennenhouse D.L. The VuSystem: A Programming System for Compute-Intensive Multimedia. IEEE Jour. Selected Areas in Communications, 14(7), pp. 1298-1313, September 1996.

[3] Mayer-Patel K. and Rowe L.A. Design and Performance of the Berkeley Continuous Media Toolkit. In Multimedia Computing and Networking 1997, pp 194-206, Martin Freeman, Paul Jardetzky, Harrick M. Vin, Editors, Proc. SPIE 3020, 1997

[4] de Mey V. and Gibbs S. A Multimedia Component Kit. In Proc. ACM Int. Multimedia Conf. (MM'93), 1993.