A Multimedia Programming Toolkit/Environment *

Raja R. Harinath, Wonjun Lee, Shwetal S. Parikh, Difu Su,
Sunil Wadhwa, Duminda Wijesekera, Jaideep Srivastava
Department of Computer Science, University of Minnesota, Minneapolis, MN 55455

Deepak R. Kenchammana-Hosekote†
IBM Almaden Research Center, San Jose, CA

e-mail: {harinath,wjlee,ssparikh,sdf,wadhwa,wijesek,srivasta}@cs.umn.edu, kencham@almaden.ibm.com

Abstract

This paper provides details and implementation experiences of a multimedia programming language and associated toolkits. The language, a data-flow paradigm for multimedia streams, consists of blocks of code that can be connected through their data ports. Continuous media flows through these ports into and out of blocks. The blocks are responsible for the processing of continuous media data. Examples of such processing include capturing, displaying, storing, retrieving and analyzing their contents. The blocks also have parameter ports that specify other pertinent parameters, such as location, and display characteristics such as geometry, etc. The connection topology of blocks is specified using a graphical editor called the Program Development Tool (PDT) and the geometric parameters are specified by using another graphical editor called the User Interface Development Tool (UIDT). Experience with modeling multimedia presentations in our environment and the enhancements provided by the two graphical editors are discussed in detail.

1 Introduction

The last few years have seen an explosive growth of interest in multimedia systems, leading to its emergence as an independent discipline of study in computer science. Given the strong interest in the systems aspects of multimedia, a number of systems, both prototype and commercial, displaying varying levels of sophistication, have been built, and this trend is on the increase. A common observation is that a lot of effort has been spent, but the end products are not as good as expected, e.g. experience with *Presto* [1], CMT [2], and Macromedia Director [3]. We believe the reason is that the current paradigm for developing multimedia software is not suitable to the task. To draw an analogy from the disciplines of artificial intelligence and databases, before the emergence of Lisp and SQL, respectively, AI programming and database programming were extremely cumbersome tasks where natural paradigms could not be used because of lack of adequate tools and environments. Once the appropriate paradigms were devised, and their corresponding programming languages and software development tools realized, programming in these disciplines was simplified. Programmers could now focus on the real challenges rather than trying to force fit an unsuitable paradigm to the task at hand.

Based on the results of a number of research efforts in multimedia programming and models, we believe that multimedia software development is ready to emerge from its nascent era of ad-hoc programming to one of a more systematic one [2, 3, 4, 5]. There seems to be an emerging consensus that the natural model to think about a multimedia programs is as a directed graph, through which multimedia streams flow. The nodes of the graphs, called *blocks* in *Presto* [1], represent operations that modify streams as they flow through them, while (directed) edges represent the input-output connections between such operations.

Development of a programming paradigm is followed by designing new languages and associated set of development tools. This paper reports those designed and implemented for the multimedia programming language of *Presto*. Based on experiences in software engineering and language design, it is our hope that these tools will make it easier to write *Presto* programs, and consequently guide us in enhancing *Presto*'s programming language and interface. Accordingly, we report our design and experiences that will hopefully contribute to the experience needed for designing a good multimedia programming language.

§2 describes *Presto*'s block-based programming model and its execution environment. §3 describes our toolkit support for *Presto* programmers. §4 describes a comprehensive example constructed using our toolkit and executed on *Presto*. Finally, §5 contains our concluding observations.

*This work is supported by Air Force contract number F30602-96-C-0130 to Honeywell Inc, via subcontract number B09030541/AF to the University of Minnesota.

† Work done while at University of Minnesota

0-8186-8227-2/97 $10.00 © 1997 IEEE

2 *Presto*

Many continuous media (CM) applications can be modeled using the data flow paradigm [5, 6]. In such a paradigm, an application program consists of a collection of data *pipes* that regulate flows of CM streams through functional *blocks*, which encapsulate functions or operations that are performed on CM streams. The pipes and blocks are connected in a directed graph to achieve the overall functionality of the application. Typically, CM data is produced by *source* blocks (camera, disk, microphone, etc.) and consumed by *sink* blocks (display, speaker, etc.). Between source and sink blocks, pipes connect intermediate blocks that perform various processing (image recognition, thresholding, synchronization, etc.) functions. Fig. 1 illustrates an example application.

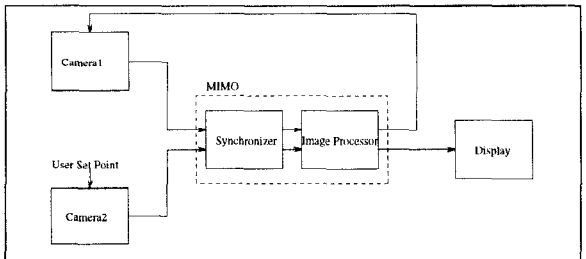


Figure 1: An example application in *Presto*

2.1 Programming Model

Presto supports the construction and execution of distributed multimedia applications from a set of primitive *blocks* via a rudimentary programming language.

A *Presto* primitive *block* is a functional unit that operates on the input data and produces some output data. It consists of a pre-compiled chunk of code that implements its functionality, a set of *data ports* that can be used to pipe in/out CM data, and a set of *parameter ports* that can be used to initialize or change some parameters in the chunk of code that implements the functionality of the block. Current primitive blocks provide the functionality of capture, storage and display of video and audio images, VCR operations on CM streams and control functions provided by and GUI development tool kit such as Motif.

Data ports are used to input/output CM data streams from blocks, and are of two types according to their functionality: *input ports* and *output ports*. Furthermore, when two blocks are connected to each other, depending upon which block is responsible for the data exchange they are categorized as *push* and *pull*. As a consequence of our notion of push and pull, if a push output port is connected to a pull input port, then in order to avoid *losing* data a buffer must be inserted. Conversely if a non-push output port is connected to a non-pull input port then an *activity* block, responsible for grabbing data from the former and stuffing in the latter, must be inserted. Such insertions take place automatically in *Presto*.

Parameter ports are used to communicate parameter values of the functionality encapsulated by blocks, and provide control over CM data propagation through blocks. Currently, *Presto* programs communicate quality of service (QoS) parameters such as rate, frame miss ratios, and file names to store and retrieve CM streams through them. An important class of parameter ports are those that encode the *user visible behavior* of some blocks. They consist of geometric information such as the size and the location (the name of the machine; the window; and the x and y offsets on the window) of display frames, cameras, and control panels (encoded in blocks) such as VCR buttons.

Composite blocks are recursively constructed by connecting appropriate blocks through appropriate ports in their components, where the primitive blocks provide the atomic level of this construction. A composite block with no unconnected data ports is a *Presto* program.

Figure 1 has an example *Presto* program, where `MIMO` is an example of a composite block, which itself does not qualify as a program, because it has exposed data ports. Notice that, as the example illustrates, program graphs can have cycles. Cycles are common in applications that perform feedback control.

CM data exchanged between blocks is in *units* which have a uniform format. Typically, a unit comprises a type, a time stamp, and a raw data segment. A JPEG frame, a set of 1024 8-bit audio samples, etc. are examples of raw data segments.

The following terminology is used in *Presto*. A block with output ports but without input ports is said to be a *source*; a block with input ports but without output ports is said to be a *sink*; and a block with parameter ports to specify its user visible properties is said to be a *user interface block*.

2.2 *Presto* Runtime

Presto programs are written as text files. The *Presto* runtime interprets these programs to construct a session that executes the desired program.

The session is created by identifying all the primitive blocks specified in the program, and connecting them as specified in the program. Since a program can include composite blocks, the process of identification is recursive.

When a session executes, it uses resources such as execution threads, buffer space, disk stubs[1], and communication channels. The *Presto* runtime includes resource managers and schedulers to handle these resources.

Presto provides application transparent distributed execution of block programs by communicating among hosts using ATM and Ethernet connections.

The *Presto* runtime consists of several modules, to handle these various tasks. There is a pre-processor, a block

[1] A disk stub is a channel between the I/O scheduler and an application block.

based-program executor, a session manager, a system resource manager that sits on top of the operating system and network services, and a storage manager for CM streams.

More details on the *Presto* system can be obtained from [1]. Details of the resource scheduling algorithms can be obtained from [7, 8, 9].

3 An Integrated Toolkit for *Presto*

Despite the diverse functionality of its runtime system, one of the major successes of the *Presto* project has been the development and implementation of a block-based programming model. This provides a flexible and modular, *lego-like* building-block approach to developing multimedia applications. In principle, an application builder should be able to access a library of blocks and rapidly prototype the desired application by re-using existing blocks from the library and developing new ones where needed. If desired, the new blocks can be added to the block library.

In order to exploit the ease and power of the *Presto* programming language, we have designed a high-level application development toolkit which can make the task of multimedia application development more palatable. Our choice of the toolkit has been dictated by the needs of *Presto* programmers, which include specifying the blocks and their port structure, specifying the connection structure between appropriate ports, and instantiating values corresponding to parameter ports. It is also immensely helpful to the programmer to be provided with a *verification/analysis* tool that communicates both the errors and system limitations in executing *Presto* programs.

In response to such needs, we have designed a toolkit consisting of three tools, and developed two of them, to run in harmony with *Presto*. They are the *Program Development Tool (PDT)*, the *User Interface Development Tool (UIDT)* and the *Program Analysis Tool (PAT)*.

The PDT allows the blocks and the interconnection structure between them to be specified graphically thus freeing the programmer from the task of specifying a graph structure in a non-graphical language. The UIDT is a tool used to specify the parameters that pertain to the *user visible behavior* of blocks (geometry, size, &c.). The PAT analyses a *Presto* program and verifies that it has no inconsistencies, and that the underlying system has the required resources to execute the program, resulting in significant saving in the *compile–debug–execute* cycle time by partially automating the error-detection mechanisms. Taken together as a well integrated toolkit, they aim to eliminate the drudgery from multimedia application development.

3.1 Issues in Designing an Integrated Toolkit

In developing our toolkit, we have learned from the conventional wisdom gained in the areas of software engineering and user interface design. However, since these areas are not the focus of this paper, we do not provide a detailed survey of the alternative approaches developed. Rather, we list the following principal lessons learned from these areas which are relevant to us: (1) Graphically oriented tools are much better than text oriented ones. (2) The tools must be integrated in an environment such that an application developer can move seamlessly from one to the other. (3) Since early detection of error has high potential benefits, this process should be automated to the extent possible.

Multimedia information management environments are expected to contain a wide collection of hardware and operating system platforms. A serious practical difficulty this poses is of system portability, a problem which is most clearly evident in the case of graphical user interface (GUI) development. For example, a GUI developed in the UNIX environment using X Windows will not run on PCs, while one developed on PCs using Microsoft graphical libraries will not run in the UNIX environment. Hence, we believe that portability of application development tools is a key requirement that must be addressed from the beginning. Towards this, we build GUIs for the proposed tools using libraries such as Java [10] and Tcl/Tk [11] that are portable across a wide variety of platforms.

3.2 Software Architecture

The system architecture pertaining to our development efforts consists of three main components that help prepare and setup the runtime of a distributed multimedia presentation. As detailed in Fig. 2, it consists of a program development tool, a user interface development tool and the *Presto* [12, 13, 14] runtime support.

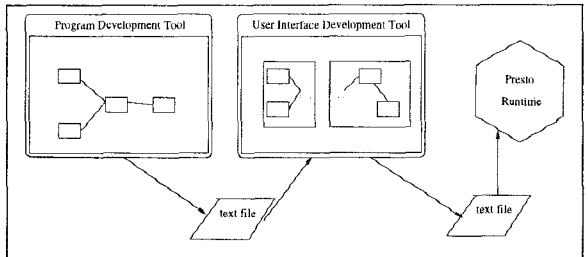


Figure 2: Architecture and Interactions between UI Tools and the Runtime

The program development tool (PDT), developed in Java, is used to construct a block model of a multimedia application by connecting appropriate ports. It can also be used to instantiate some block parameters such as their location.

Using the location information provided by the PDT, the user interface development tool (UIDT) can be used to instantiate *user-visible* behavior of certain program blocks. For example, a button block has user-visible behavior of color, bitmap or label, geometry such as width, height and the displacement of the upper left hand corner of the window panel in which it is placed.

In addition, there is other geometric information such as the size, color and position of the window at local

sites throughout the distributed execution environment. Although these are neither properties of the block programming model nor that of any visible blocks, they contribute to the aesthetic appeal of the demonstration. Hence, the UIDT also specifies other *visibility related* parameters of Presto program executions. The list of such parameters is growing and is expected to encompass all properties that appeal to the aural and visual senses associated with the runtime environment.

As stated, the Presto runtime has a pre-processing stage in which location information of all blocks are derived by using a fragmentation algorithm. In case the program needs to be distributed, the fragmentation algorithm inserts *system blocks* that are necessary to allow distributed execution. These are blocks that facilitate communication, such as network activity.

The PDT, UIDT and the Presto runtime communicate through text files that can be read and written by all three of them. This communication paradigm was chosen mainly because of the independence of each component from the other, the non-real-time nature of interaction between them and the possibility of using or developing alternate tools for each of the components.

Presto programs contain the parameters like position, color and geometry of base local windows on which visible blocks appear, block names and types, number and type of ports that are attached to each block, structure of interconnections between data and parameter ports, initial parameter bindings of parameter ports. Since the user visible behavior of blocks are specified through parameter ports, they do not require any additional support.

3.3 Program Development Tool (PDT)

A block in Presto is a basic unit that accepts some input(s), carries out some transformation on it, and produces some output(s). The functionality of the block is provided by a chunk of code. To support a graphical program development tool, however, each block must also have its unique graphical (iconic) representation, which the developer can use to refer to the block during graphical application construction. This idea has already been incorporated in the Presto system. Fig. 3(a) shows the working of this tool. The application developer creates a program, e.g., *target_recog*, by connecting pre-existing blocks and/or defining new ones. The tool creates a file (*target_recog.prog* in the example) which is used by other tools and passed to the runtime.

The PDT works on a common input and output file syntax. The user gets the PDT to read in an input file, which is graphically displayed on the screen. The PDT then allows the user to make the relevant modifications to this file through its graphical editing capabilities. The capabilities of the tool are described in §3.3.1. The user can modify and save the changes in the same file.

3.3.1 Editing Capabilities. Fig. 3(b) shows a screen capture of a sample run of the intelligence assistant module of the example described later in §4. The module was created by using the Program Development Tool. When the user double clicks on any of the blocks, a pop-up window is displayed which lists the parameters and connections of the particular block. The user can then modify, add or delete any of the parameters or connections. The user can also edit the source file of the particular block from this window. Fig. 3(c) shows a sample pop-up dialog box for a block (*ActivityBlock_p* in this example).

Editing functions provided by the PDT include creating new blocks, changing and viewing block parameters, creating and changing connections between ports of blocks, and editing code. The tool has *undo* facilities to mitigate the effects of editing errors. The tool also supports composite (hierarchically specified) blocks, allowing users to zoom into/out of composite blocks.

3.4 User Interface Development Tool (UIDT)

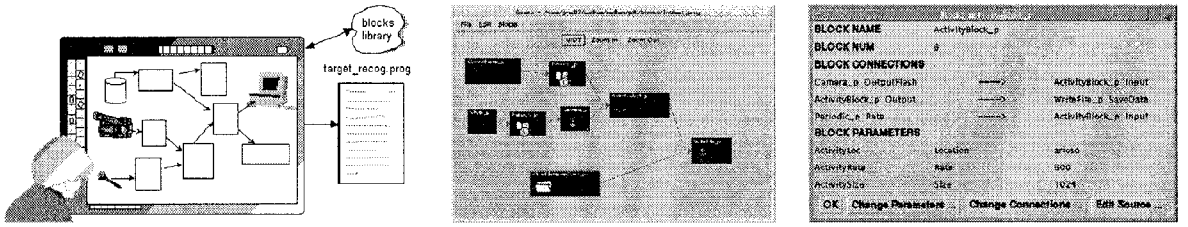
By using the PDT, an application developer can specify what blocks are used in the program and how they are connected with each other. However, there is still the issue of specifying what the user-visible behavior of the application will be, and how the user will interact with it. The purpose of the UIDT is to provide this capability. A key idea in our approach is to introduce the concept of *user visible behavior* of a block, which is described analogous to its interface definition, functionality definition, and graphical representation. When a new block is added to the block library, its user-visible behavior is stored with the object.

As shown in Fig. 4(a), the UIDT takes a program file (e.g. *target_recog.prog*) as input, and creates a palette for the UI developer to design the UI. The visual behavior of each user-visible block of *target_recog* is made available to the UI developer. The UI developer designs the layout of the user interface by modifying the user-visible behavior of the blocks, and adding modes of user interaction with the application (if any). The UIDT modifies the file which is then passed on to the runtime environment of Presto. User Interface design has a substantial amount of human creativity, and we believe our approach will largely eliminate the drudgery while allowing the UI developer to focus on creative aspects.

3.4.1 Design of the UIDT. As a graphical user interface tool, UIDT provides a *WYSIWYG* (What You See Is What You Get) environment.

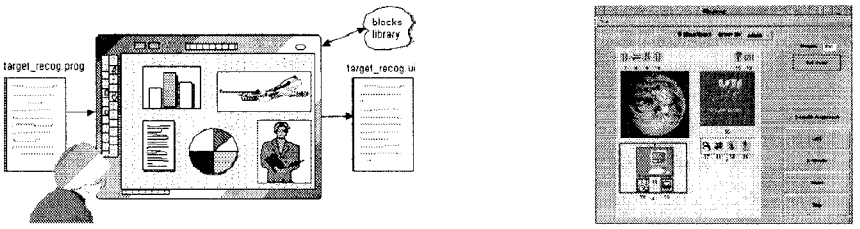
Each Presto program can execute on multiple locations, and each location is given a window of its own, called the *base window*, or *canvas*. All canvases have the same user visible properties, which include offsets on the screen(x and y), size(*width* and *height*), and a title.

The interface elements of user visible blocks are placed on the canvas of the location the block has been assigned



(a) The PDT (b) PDT Main Screen (c) Dialog for ActivityBlock_p

Figure 3: Program Development Tool (PDT)



(a) The UIDT (b) UIDT window

Figure 4: User Interface Development Tool (UIDT)

to. Unlike canvases, different UI Blocks have different user visible properties. Even UI Blocks of the same type have differing user visible properties (e.g. some buttons have bitmaps, some do not). When a new UI Block is created, it may also introduce some user visible properties unknown to the UIDT; the UIDT is able to show and edit these unknown properties.

Some properties are common to every UI Block. They are offsets(x and y), size(*width* and *height*), name and type. In addition, *bitmap* and *label* are two common user visible properties. The UIDT shows a UI Block as a rectangle (if it has a bitmap, shows its bitmap) with a screen representation of the six common properties. If the UI Block has a label, it is also shown. Other user visible properties are shown in text fields.

Geometric properties (x and y offsets, width and height) can be edited by moving and resizing the UI Blocks. Other user visible properties can be edited by changing corresponding text fields. Dialog items can be selected/deselected, and be aligned in various ways.

A sample UIDT window is shown in Fig. 4(b).

3.5 Program Analysis Tool (PAT)

When blocks are connected to form block-based programs, the developer must ensure that the port connections are correct, i.e. *push* and *pull* connections are appropriately made, the data types match, and the flow rates (in case of continuous media) are compatible. This is analogous to type checking in programming languages. Experience in programming language design has shown that strongly typed languages lend themselves to better programming discipline as well as less programming errors, since one can perform compile time type checking. Languages without strong typing, provide programmers with greater flexibility but tend to promote lack of programming discipline and thus result in buggy code. The situation is similar with a block-based programming language, and is likely to be exacerbated with the introduction of operation flows in addition to data flows. The ideas behind the proposed PAT are similar. Currently the upper layers of Presto runtime perform the *push/pull* compatibility analysis. We propose to move that functionality into the PAT. In addition, the PAT will perform data-type and operation-type compatibility analysis for port connections in block-based programs. Finally, for data ports through which continuous media flows, the PAT will analyze the compatibility of rate requirements. As shown in Fig. 5, the PAT will take *target_recog.prog* as input and produce *target_recog.err* as output, which can be graphically presented to the programmer.

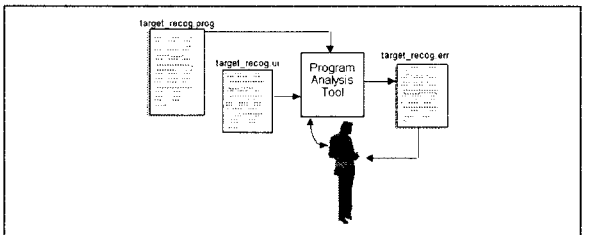


Figure 5: Program Analysis Tool

4 Example: Distributed Target Recognition Application

In order to demonstrate the capabilities of Presto and its integrated toolkit, we provide details of a program that was developed and executed on Presto using the PDT and the UIDT. Our example consists of a hypothetical battle scenario, with a *distributed target recognition (DTR)* application.

4.1 Components of the DTR Example

In the example DTR scenario, live audio/video broadcasts arrive at a command post. They are first analyzed by an *Intelligence Assistant*, whose job is to store the data on a disk to be later analyzed by an *Intelligence Analyst*. The intelligence analyst browses and selects potentially informative segments to be analyzed by the RSTA (the DARPA Reconnaissance, Surveillance and Target Acquisition) image analysis software, and stores it to be viewed by the battlefield *Commander*. The commander can view the annotated clips and take appropriate action based on them. Our encoding of this scenario is captured by three Presto programs, one for each role in the scenario, i.e., Intelligence Assistant, Intelligence Analyst, and Commander.

This application is intended to be distributed across workstations. The application user interface shall include the corresponding UI elements like video display, toggle switches, VCR panels, image captures, listers, sliders, capture buttons, and file name specification blocks.

Fig. 6 shows the block-based Presto programs we created using the PDT for our DTR example.

Intelligence Assistant The output of a camera is digitized and presented to the intelligence assistant. A video spooling capability (i.e. recording the video segment currently being viewed to a named Presto file) is provided by an *ON/OFF* toggle switch.

The intelligence assistant module (Fig. 6(a)), gets the output of a camera and spools it to a named file. The controls *Record* and *Pause* for spooling are provided by the UI block called *ControlPanel_p*. Rate is specified using a block called *Slider_p*. Presto transforms a user program to a system program by locating push/pull incompatibilities and connecting them by inserting special blocks like *ActivityBlock_p* or *Periodic_p*. Between a pull output port of *Camera_p* and a push input port of *WriteFile_p*, an *ActivityBlock_p* is inserted by Presto, which is transparent to the application programmer. The Presto file name is specified to the system by means of the block *WriteFileNameSpec_p*. This is the mechanism by which a raw video input is spooled to a Presto file by the intelligence assistant.

Intelligence Analyst The intelligence analyst module provides full VCR controls [15] on the spooled Presto files which are created by the intelligence assistant's filtered images. The intelligence analyst can view a previously-spooled video file, scan forward or backward, freeze frames, and capture one or more still images to a file. The RSTA program is an image analysis software, which analyzes still images for potential targets and generates annotated images with outlined targets.

In the intelligence analyst module (Fig. 6(b)), still images stored in the Presto file system are captured by *Capture_p*, and displayed by *DisplayCaptureRaw_p* on a display unit through a *ControlPanel_p* block, which implements VCR operations on the screen. The images are analyzed by the RSTA image analysis software, and as a result annotated images are generated. We can also store the annotated images to a file by using the block *UFSWriteFileNameRaw_p*.

Commander The Commander views still images annotated by the RSTA image analysis software. The commander's user interface provides a set of controls for selecting an image to view from among those available.

The commander module (Fig. 6(c)) consists of three basic blocks. It can play back a stored file on screen using *DisplayRaw_p*. *Lister_p* shows the user a file/directory list and can be used to select an annotated still frame from the file system using block *UFSReadFileRaw_p*.

5 Conclusions

In this paper, we have described a data-flow oriented block based programming language suitable for distributed multimedia programming. We have shown an explicit example in detail to exemplify its expressiveness. We have shown that Presto's usability can be enhanced by providing tools that aid in developing block based programs, specifying their user interfaces and verifying their correctness.

In this respect, we have implemented a program development tool (PDT) that specifies the blocks and their interconnection structure. It has been observed that this tool greatly enhances the usability of Presto as a general purpose distributed multimedia testbed. Nevertheless, specifying the user interfaces of Presto programs did not appear to be an easy task in the PDT, and hence we developed another graphical user interface for that purpose, called the user interface development tool (UIDT).

We are currently developing a program analysis tool that will perform static analysis of multimedia programs specifiable in Presto. Future work on Presto also includes the enhancement of tools to incorporate Quality of Service (QoS) specification, its translation into parameters the resource managers can use [16], and analysis of the system's ability to achieve the specified QoS.

References

[1] J. Huang, D. Kenchammana-Hosekote, J. Richardson, and J. Srivastava, "Presto: A Prototyping Environment for Mission Critical Multimedia Applications," in *Proceedings of IEEE Dual Use and Applications Conference*, IEEE, March 1996.

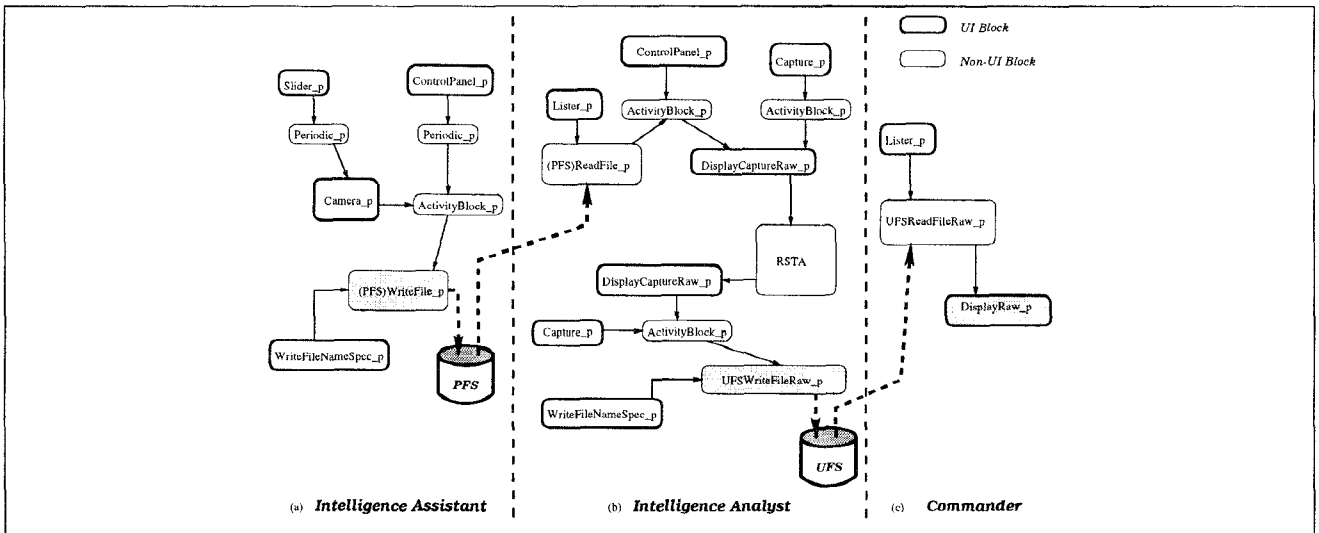


Figure 6: Block-based Programming Diagram for the Distributed Target Recognition Application

[2] K. Patel, *Introduction to the CM Toolkit.* Berkeley Multimedia Research Center, 1995.

[3] J. Thompson and Gottlieb, *Macromedia Director Developer's Guide to Lingo.* 1995.

[4] C. Lindblad, D. Wetherall, and D. Tennenhouse, "The VuSystem: A Programming System for Visual Processing of Digital Video," in *Proceedings of ACM Multimedia 94*, October 1994.

[5] D. Tennenhouse, J. Adam, D. Carver, H. Houh, M. Ismert, C. Lindblad, W. Stasior, D. Wetherall, D. Bacher, and T. Chang, "The ViewStation: A Software-intensive Approach to Media Processing and Distribution," *Multimedia Systems*, vol. 3, pp. 104–115, July 1995.

[6] D. Stewart, R. Volpe, and P. Khosla, "Design of Dynamically Reconfigurable Real-time Software using Port-based Objects," Tech. Rep. CMR-RI-TR-93-11, Dept. of ECE, CMU, 5000, Forbes Avenue, Pittsburgh, PA 15213, July 1993.

[7] J. Huang and D.-Z. Du, "Resource management for continuous multimedia applications," in *Proceedings of the* 15^{th} *Real Time Systems Symposium*, IEEE, December 1994.

[8] J. Huang and P.-J. Wan, "On supporting mission critical applications," in *Proceedings on International Conference on Multimedia Computing Systems*, IEEE, June 1996.

[9] J. Huang, Y. Wang, and D. Kenchammana-Hosekote, "Decentralized end-to-end scheduling for continuous multimedia," in *Proceedings of Workshop on Network and Operating System Support for Digital Audio and Video*, IEEE/ACM, April 1996.

[10] J. Gosling, F. Yellin, and T. J. Team, *The Java Programming Language.* Addison-Wesley, 1996.

[11] J. Ousterhout, *Tcl and the Tk Toolkit.* Addison-Wesley, 1993.

[12] M. Agrawal, D. Kenchammana-Hosekote, A. Pavan, S. Bhattacharya, and N. Vaidyanathan, "High Performance Network Services for Multimedia-Integrated Distributed Control," tech. rep., Honeywell Technology Center, Minneapolis, MN, July 1996.

[13] J. Huang, D. Kenchammana-Hosekote, and J. Wan, "Heterogeneous Distributed Multimedia Information Management for the Infosphere," tech. rep., Honeywell Technology Center, Minneapolis, MN, January 1996.

[14] M. Agrawal, R. Harinath, J. Huang, J. Richardson (Honeywell Technology Center), W. Lee, J. Srivastava, D. Su, S. Wadhwa, and D. Wijesekera (University of Minnesota), "Sonata: Active Views in a Distributed Object-Oriented System," in *Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6) (Appeared in Research Prototype Demonstration Session)*, (Portland, Oregon), August 1997.

[15] D. Kenchammana-Hosekote and J. Srivastava, "I/O scheduling for Digital Continuous Media," *ACM Multimedia Systems Journal*, vol. 5, pp. 213–237, July 1997.

[16] D. Wijesekera and J. Srivastava, "Quality of Service Metrics for Continuous Media," *Multimedia Tools and Applications*, vol. 3, pp. 127–166, September 1996.